**NXP Semiconductors**
Reference Manual

Document identifier: vspa3_ism_16au_la1224
Rev. 0, 04/2021

# VSPA-16SP ISA-v3.0 Instruction Set Manual for LA1224

# Contents


## Chapter 1 Preface ..... 10
1.1 Introduction ..... 10
1.2 Variants of VSPA ..... 10
1.3 New features in VSPA ISA-v3.0 ..... 10
1.4 Using this manual ..... 10
1.5 Conventions ..... 11
1.6 Acronyms and abbreviations ..... 12

## Chapter 2 VSPA Architecture Overview ..... 14
2.1 VSPA architecture introduction ..... 14
2.1.1 VCPU introduction ..... 14
2.1.2 IPPU introduction ..... 15
2.1.3 DMA controller ..... 15
2.1.4 AUX DMA ..... 16
2.1.5 IP registers ..... 16
2.1.6 Vector data memory ..... 16
2.1.7 Data Memory Arbitration ..... 16
2.1.8 Debug and trace unit ..... 16

## Chapter 3 VCPU Architecture ..... 17
3.1 Control plane ..... 17
3.1.1 Program memory ..... 18
3.1.2 Program control ..... 19
3.2 Data plane ..... 20
3.2.1 Data memory ..... 22
3.2.2 Data memory pointers ..... 23
3.2.3 Vector register array ..... 25
3.2.4 DMEM address space vs. VRA address space ..... 28
3.2.5 Vector rotate unit ..... 29
3.2.6 VAU operand source registers ..... 29
3.2.7 VAU operand source register muxes ..... 29
3.2.8 Vector arithmetic unit ..... 30
3.2.9 Vector predication ..... 31
3.2.10 VAU destination mux ..... 32
3.2.11 Vector sign capture register (H) ..... 32
3.2.12 Vector NCO ..... 32
3.2.13 Code generation unit ..... 33
3.2.14 Scalar arithmetic & logic unit ..... 33
3.3 Data precision ..... 34
3.3.1 Two's complement conversion ..... 35
3.4 Data types ..... 36
3.5 VCPU GO events ..... 36
3.6 Byte order ..... 37
3.7 IRQ for thread killing ..... 37

## Chapter 4 VCPU Instruction Set ..... 38
4.1 VCPU instruction set overview ..... 38
4.2 Instruction set organization ..... 38

4.2.1 Instruction families..... 38
4.2.2 Instruction formats.....39
4.3 Internal operand generators ..... 40
4.4 VCPU instruction set summary.....40
4.5 System control registers.....103
4.5.1 Vector precision latency.....109
4.6 VCPU condition codes.....109
4.6.1 Multiply condition codes.....111
4.6.2 Divide condition codes.....112
4.6.3 Modulus condition codes.....112
4.7 Data memory pointer instructions.....112
4.8 Data memory load & store instructions.....113
4.8.1 DMEM address generation modes (ptr_mode).....113
4.8.2 Post-modifications of a*X* registers.....113
4.8.3 Special notes.....114
4.9 Vector register array instructions.....130
4.9.1 RAG instructions.....136
4.10 Rotate register instructions.....141
4.10.1 Rotate-register modes.....142
4.11 Extrema instructions.....162
4.11.1 Extrema configuration.....163
4.11.2 Extrema Functionality.....164
4.11.3 Extrema instructions code example.....165
4.12 Vector AU source register instructions.....165
4.12.1 VRA data reads.....167
4.12.2 Data permutation and/or replication.....168
4.12.3 VRA data type conversion.....169
4.12.4 *S0mode* options and detailed description.....172
4.12.5 *S1mode* options and detailed description.....185
4.12.6 *S2mode* options and detailed description.....194
4.13 AU instructions.....202
4.13.1 AU latency.....204
4.13.2 AU instructions code example.....204
4.13.3 Multiply add functionality.....205
4.13.4 Multiply and add with sign conversion.....206
4.13.5 Multiply accumulate functionality.....207
4.13.6 Multiply add with feedback functionality.....209
4.13.7 Decimation in time and frequency butterfly functionality.....210
4.14 Special AU instructions.....226
4.14.1 SAU input and output vector.....227
4.14.2 SAU latency.....228
4.14.3 Special AU instructions code example.....228
4.14.4 Reciprocal functionality.....228
4.14.5 Reciprocal square root functionality.....228
4.14.6 Square root functionality.....229
4.14.7 Pre adder functionality.....229
4.15 Store AU/SAU output instructions.....229
4.15.1 VAU data type conversion.....230
4.16 Vector predication.....237
4.17 GP instructions.....239
4.17.1 GP move instructions.....240
4.17.2 Linear feedback shift register instructions.....243
4.17.3 Floating point generation instructions.....243
4.17.4 Arithmetic instructions.....244
4.18 Hardware loop instructions.....250

4.18.1 Hardware loop control mechanism....250
4.18.2 Overwriting a set.loop instruction....251
4.18.3 Nested hardware loops....252
4.18.4 Early termination of a hardware loop....252
4.18.5 Hardware loop execution constraints....253
4.18.6 Hardware loop legal examples....256
4.18.7 set.loop instruction....256
4.19 Control flow instructions....257
4.19.1 Jump delay slots....258
4.19.2 Compare-and-jump example....258
4.19.3 Back-to-back conditional jumps....258
4.20 Conditional instructions....260
4.20.1 Logical test instruction modifiers....261
4.20.2 Condition code flags....262
4.20.3 Conditional instruction setup time....265
4.21 Numerically controlled oscillator (NCO) instructions....266
4.22 CGU instructions....268
4.22.1 S1Mux add-on for CGU....270
4.22.2 Downlink scrambling subunit....271
4.22.3 Long-sequence uplink scrambling subunit....272
4.22.4 Short-sequence uplink scrambling subunit....272
4.22.5 LTE scrambling subunit (Gold sequence)....274
4.22.6 V.fast/VDSL training sequence subunit....274
4.22.7 V.fast/VDSL quadratic scrambling sequence subunit....275
4.22.8 OVSF unit....275

**Chapter 5 IPPU Architecture....277**
5.1 IPPU overview....277
5.1.1 IPPU SOC level components....277
5.1.2 IPPU features....277
5.2 Inter-vector permutation processing unit....278
5.2.1 IPPU core....278
5.2.2 IPPU operating states....279
5.2.3 IPPU memory access considerations....280
5.2.4 IPPU initialization....280
5.3 IPPU interrupts....281
5.4 IPPU done to VCPU go event....281

**Chapter 6 IPPU Instruction Set....282**
6.1 Size definitions....282
6.2 Hardware definitions....282
6.3 IPPU instructions summary....284
6.4 Load instructions....285
6.5 Load memory index instructions....290
6.6 Store instructions....290
6.7 Set range instructions....292
6.8 Configure bit-reversal, digit-reversal engine instructions....293
6.9 Move register instruction....295
6.10 Load input argument instruction....296
6.11 Compare instruction....297
6.12 Jump instructions....297
6.13 Loop instructions....298
6.14 Done instruction....299

6.15 Clear register instruction....299
6.16 Set/clear element mask register instructions....300
6.17 Add instructions....300
6.18 Advanced features/usage notes....301
6.18.1 Delay slot considerations....301
6.18.2 BR - Bit-reversal....301
6.18.3 BRMR - Bit-reversal-multiple-registers....301
6.18.4 Element-reversal....310
6.18.5 Indirect addressing....312
6.18.6 Vectorized indirect addressing....313

**Chapter 7 DMA Controllers....314**
7.1 Direct memory access unit (DMA)....314
7.1.1 DMA module operation....314
7.1.2 Issuing DMA commands....314
7.1.3 DMA channel arbitration....315
7.1.4 DMA deinterleaving engine....315
7.1.5 Effect of invasive debug on DMA....316
7.1.6 DMA use with FIFOs....316
7.1.7 DMA features not supported....317
7.1.8 Source/destination memory formatting....317
7.2 Auxilary direct memory access unit (AUX_DMA)....320
7.2.1 AUX_DMA module operation....320
7.2.2 Issuing AUX_DMA commands....320

**Chapter 8 Mailboxes....321**
8.1 Mailboxes....321

**Chapter 9 AXI Slave....322**
9.1 AXI slave overview....322
9.2 Memory map....322
9.3 Usage example....322
9.4 VSPA AXI slave flag system....322
9.5 Interface limitations....323

**Chapter 10 Debug and Trace....324**
10.1 Debug....324
10.1.1 VSPA debug block diagram....325
10.1.2 Debug functional description....325
10.1.3 VSPA messages....332
10.1.4 Debug using the DMA FIFOs....347

**Chapter 11 Interrupts....348**
11.1 Interrupts....348

**Chapter 12 Initialization....349**
12.1 Initialization....349

**Chapter 13 VSPA IP Registers.....350**
13.1 Slow read registers.....350
13.2 VSPA register descriptions.....350
13.2.1 VSPA_CCSR memory map.....351
13.2.2 VSPA Hardware Version (HWVERSION).....357
13.2.3 VCPU Software Version (SWVERSION).....358
13.2.4 VCPU System Control register (CONTROL).....359
13.2.5 VSPA Interrupt Enable register (IRQEN).....366
13.2.6 VSPA Source 1 Info (STATUS).....368
13.2.7 VCPU to Host flags register a (VCPU_HOST_FLAGS0 - VCPU_HOST_FLAGS1).....373
13.2.8 Host to VCPU Flags register a (HOST_VCPU_FLAGS0 - HOST_VCPU_FLAGS1).....374
13.2.9 External Go Enable (EXT_GO_ENA).....375
13.2.10 External Go Status (EXT_GO_STAT).....376
13.2.11 VSPA VCPU Illegal Opcode Address (ILLOP_STATUS).....377
13.2.12 VSPA Parameters 0 (PARAM0).....378
13.2.13 VSPA Parameters 1 (PARAM1).....382
13.2.14 VSPA Parameters 2 (PARAM2).....383
13.2.15 VCPU DMEM Size (DMEM0_SIZE).....387
13.2.16 AXI Slave flags register a (AXISLV_FLAGS0 - AXISLV_FLAGS1).....388
13.2.17 AXI Slave Go Enable register a (AXISLV_GOEN0 - AXISLV_GOEN1).....388
13.2.18 Platform Input (PLAT_IN_0).....389
13.2.19 Platform Output (PLAT_OUT_0).....390
13.2.20 Cycle counter MSB register (CYC_COUNTER_MSB).....391
13.2.21 Cycle Counter LSB Register (CYC_COUNTER_LSB).....392
13.2.22 DMEM/PRAM Address (DMA_DMEM_PRAM_ADDR).....393
13.2.23 DMA AXI Address (DMA_AXI_ADDRESS).....394
13.2.24 AXI Byte Count register (DMA_AXI_BYTE_CNT).....395
13.2.25 DMA Transfer Control register (DMA_XFR_CTRL).....396
13.2.26 DMA Status/Abort Control (DMA_STAT_ABORT).....401
13.2.27 DMA IRQ Status (DMA_IRQ_STAT).....403
13.2.28 DMA Complete Status (DMA_COMP_STAT).....404
13.2.29 DMA Transfer Error Status (DMA_XFRERR_STAT).....405
13.2.30 DMA Configuration Error Status (DMA_CFGERR_STAT).....405
13.2.31 DMA Transfer Running Status (DMA_XRUN_STAT).....406
13.2.32 DMA Go Status (DMA_GO_STAT).....407
13.2.33 DMA FIFO Availability Status (DMA_FIFO_STAT).....408
13.2.34 AUX DMA status register (AD_STATUS).....409
13.2.35 Load Register File Control register (Slow read register) (LD_RF_CONTROL).....411
13.2.36 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_0).....413
13.2.37 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_0).....417
13.2.38 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_1).....421
13.2.39 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_1).....425
13.2.40 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_2).....429
13.2.41 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_2).....433
13.2.42 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_3).....437
13.2.43 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_3).....441

13.2.44 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_4)..... 445
13.2.45 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_4)..... 449
13.2.46 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_5)..... 453
13.2.47 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_5)..... 457
13.2.48 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_6)..... 461
13.2.49 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_6)..... 465
13.2.50 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_7)..... 469
13.2.51 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_7)..... 473
13.2.52 VCPU Go Address (VCPU_GO_ADDR)..... 477
13.2.53 VCPU Go Stack (VCPU_GO_STACK)..... 478
13.2.54 AUX DMA program memory address register (AD_PRAM_ADDR)..... 479
13.2.55 AUX DMA AXI address register (AD_AXI_ADDR)..... 480
13.2.56 AUX DMA byte count register (AD_BYTE_COUNT)..... 480
13.2.57 AUX DMA control register (AD_CTRL)..... 481
13.2.58 VCPU Mode 0 (VCPU_MODE0)..... 482
13.2.59 VCPU Mode 1 (VCPU_MODE1)..... 484
13.2.60 VCPU CREG 0 (VCPU_CREG0)..... 485
13.2.61 VCPU CREG 1 (VCPU_CREG1)..... 488
13.2.62 Store Unalign Vector Length (ST_UL_VEC_LEN)..... 489
13.2.63 Vector Predicate Control (VP_CONTROL)..... 491
13.2.64 Extended Condition Control (XC_CONTROL)..... 493
13.2.65 VCPU CREG 2 (VCPU_CREG2)..... 494
13.2.66 RAG set rS0 & rS1 registers (RAG_SET_S0S1_0 - RAG_SET_S0S1_7)..... 496
13.2.67 RAG set rS2, rV & rSt registers (RAG_SET_S2VSt_0 - RAG_SET_S2VSt_7)..... 497
13.2.68 General Purpose Input registers [23 registers] (GP_IN0 - GP_IN22)..... 498
13.2.69 General Purpose Output registers [21 registers] (GP_OUT0 - GP_OUT20)..... 499
13.2.70 VCPU to DQM Trace Small Outbox register (DQM_SMALL)..... 500
13.2.71 VCPU to DQM Trace Large MSB Outbox register (DQM_LARGE_MSB)..... 501
13.2.72 VCPU to DQM Trace Large LSB Outbox register (DQM_LARGE_LSB)..... 502
13.2.73 VCPU to Debugger 32-bit Outbox register (VCPU_DBG_OUT_32)..... 503
13.2.74 VCPU to Debugger 64-bit MSB Outbox register (VCPU_DBG_OUT_64_MSB)..... 504
13.2.75 VCPU to Debugger 64-bit LSB Outbox register (VCPU_DBG_OUT_64_LSB)..... 505
13.2.76 Debugger to VCPU 32-bit Inbox register (VCPU_DBG_IN_32)..... 506
13.2.77 Debugger to VCPU 64-bit MSB Inbox register (VCPU_DBG_IN_64_MSB)..... 507
13.2.78 Debugger to VCPU 64-bit LSB Inbox register (VCPU_DBG_IN_64_LSB)..... 508
13.2.79 VCPU to Debugger Mailbox Status register (VCPU_DBG_MBOX_STATUS)..... 509
13.2.80 VCPU to host outbox message n MSB register (VCPU_OUT_0_MSB - VCPU_OUT_1_MSB)..... 511
13.2.81 VCPU to host outbox message n LSB register (VCPU_OUT_0_LSB - VCPU_OUT_1_LSB)..... 512
13.2.82 VCPU from Host Inbox Message n MSB (VCPU_IN_0_MSB - VCPU_IN_1_MSB)..... 513
13.2.83 VCPU from host inbox message n LSB register (VCPU_IN_0_LSB - VCPU_IN_1_LSB)..514
13.2.84 VCPU to Host Mailbox Status register (VCPU_MBOX_STATUS)..... 514
13.2.85 Host to VCPU Outbox Message n MSB register (HOST_OUT_0_MSB - HOST_OUT_1_MSB)..... 516
13.2.86 Host to VCPU Outbox Message n LSB register (HOST_OUT_0_LSB - HOST_OUT_1_LSB)..... 517

13.2.87 Host from VCPU Inbox Message n MSB (HOST_IN_0_MSB - HOST_IN_1_MSB)..... 518
13.2.88 Host from VCPU Inbox Message n LSB Register (HOST_IN_0_LSB - HOST_IN_1_LSB)519
13.2.89 Host Mailbox Status Register (HOST_MBOX_STATUS)..... 520
13.2.90 IPPU Control register (IPPUCONTROL)..... 521
13.2.91 IPPU Status register (IPPUSTATUS)..... 524
13.2.92 IPPU Run Control register (IPPURC)..... 527
13.2.93 IPPU Arg Base Address register (IPPUARGBASEADDR)..... 528
13.2.94 IPPU Hardware Version (IPPUHWVER)..... 529
13.2.95 IPPU Software Version (IPPUSWVER)..... 530

**Chapter 14 Debug registers..... 532**
14.1 VSPA_DBG register descriptions..... 532
14.1.1 VSPA_DBG memory map..... 532
14.1.2 Global Debug Enable register (GDBEN)..... 536
14.1.3 Debug Run Control register (RCR)..... 538
14.1.4 Debug Run Control Status register (RCSTATUS)..... 539
14.1.5 Trace Control register 1 (TC1)..... 543
14.1.6 Debug Halt Action Control register (HACR)..... 547
14.1.7 Debug Resume Action Control register (RACR)..... 548
14.1.8 Debug Program Trace Sync Action Control register (PTSACR)..... 550
14.1.9 Debug Program Trace Start Action Control register (PSTACR)..... 551
14.1.10 Debug Program Trace Stop Action Control register (PSPACR)..... 553
14.1.11 Debug Data Trace Start Action Control register (DSTACR)..... 555
14.1.12 Debug Data Trace Stop Action Control register (DSPACR)..... 556
14.1.13 Debug Data Trace Message Generation Action Control register (DTMACR)..... 558
14.1.14 Debug WPM Generation Action Control register (WPMACR)..... 559
14.1.15 Debug TSM Control register (TSMCR)..... 561
14.1.16 Debug VSPA Timestamp Counter (Upper) register (TSMCNT_U)..... 563
14.1.17 Debug VSPA Timestamp Counter (Lower) register (TSMCNT_L)..... 564
14.1.18 Debug VSP Architecture Visibility Address Pointer register (RAVAP)..... 565
14.1.19 Debug VSP Architecture Visibility Fixed Data register (RAVFD)..... 573
14.1.20 Debug VSP Architecture Visibility Incrementing Data register (RAVID)..... 574
14.1.21 Debug Verification register (DVR)..... 575
14.1.22 Debug Cross Trigger Out a Action Control registers (CTO0ACR - CTO7ACR)..... 576
14.1.23 Debug Comparator Control and Status register (DC0CS - DC7CS)..... 578
14.1.24 Debug Comparator a Data register (DC0D - DC7D)..... 582
14.1.25 Debug Comparator a Arm Action Control registers (C0AACR - C7AACR)..... 585
14.1.26 Debug Comparator a Disarm Action Control registers (C0DACR - C7DACR)..... 586
14.1.27 Debug Comparator a Trigger Action Control registers (C0TACR - C7TACR)..... 588
14.1.28 Debug to VSP 32-bit Outbox register (OUT_32)..... 590
14.1.29 Debug to VSP 64-bit MSB Outbox register (OUT_64_MSB)..... 591
14.1.30 Debug to VSP 64-bit LSB Outbox register (OUT_64_LSB)..... 592
14.1.31 VSP to Debugger 32-bit Inbox register (IN_32)..... 593
14.1.32 VSP to Debugger 64-bit MSB Inbox register (IN_64_MSB)..... 593
14.1.33 VSP to Debugger 64-bit LSB Inbox register (IN_64_LSB)..... 594
14.1.34 Debugger to VSP Mailbox Status register (MBOX_STATUS)..... 595
14.1.35 Debug Parameter 0 Register (PARAM_0)..... 597
14.1.36 Peripheral ID4 register (PIDR4)..... 598
14.1.37 Peripheral ID5 register (PIDR5)..... 599
14.1.38 Peripheral ID6 register (PIDR6)..... 600
14.1.39 Peripheral ID7 register (PIDR7)..... 601
14.1.40 Peripheral ID0 register (PIDR0)..... 602
14.1.41 Peripheral ID1 register (PIDR1)..... 603
14.1.42 Peripheral ID2 register (PIDR2)..... 604

14.1.43 Peripheral ID3 register (PIDR3)..... 605
14.1.44 Component ID0 register (CIDR0)..... 606
14.1.45 Component ID1 register (CIDR1)..... 606
14.1.46 Component ID2 register (CIDR2)..... 607
14.1.47 Component ID3 register (CIDR3)..... 608

**Appendix A Revision History.....610**

# 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 v3.0. The significant changes in comparison to VSPA ISA v2.0 and v1.0 are described below.

VSPA can be classified based on the number of arithmetic units (AU). This ISM supports VSPA-16SP for LA1224. LA1224 has 1 instance of VSPA 16AU single precision (SP) and 24 bit precision (F24). This means that, VSPA core for LA1224 supports:

- 16 complex MAC operations per clock cycle / single precision
- 32 complex MAC operations per clock cycle / 24 bit precision
- Single precision and 24 bit precision math

## 1.3 New features in VSPA ISA-v3.0

- Backwards Compatibility: The core will boot up in VSPA3 mode. It must be written to VSPA2 mode. This allows software developed on a VSPA2 target SOC to execute with identical performance on an SOC using VSPA3.
- Byte Addressing: All data memory addresses are represented in units of bytes on VSPA3.
- Memory address registers (a0-a19) have been extended to 32 bits.
- 2x 24-bit AU operations per clock: VSPA3 supports twice the number of AU operations per clock when using the F24 AU type.
- Vector predicate support: VSPA3 provides vector predicate registers and new predicate conditional actions.
- 512-bit AXI support on the DMA.
- Banked Data Memory: Data memory now supports up to two banks to allow concurrent access by multiple masters including; VCPU(2)(Ld/St provides 2 accesses to memory in same cycle), IPPU, DMA, AXI-Slave, and Debugger.
- 2X ld.qam/st.llr performance: Reduction in clock cycles for modulation/demodulation.

## 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 controllers
   - *Chapter 7.1*: Includes detailed description of the Direct memory access unit (DMA). Also, lists the features that the DMA does not support.
   - *Chapter 7.2*: Includes detailed description of the Auxilary direct memory access unit (AUX_DMA).
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 14*: 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 IP registers
    - *Chapter 13*: 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 |
| CGU | Code generation unit |
| CREG | System control register |
| DI | Deinterleaving |
| DMA | Direct memory access |
| F24 | 24 bit precision floating point |
| FFT | Fast Fourier transform |
| 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 |

*Table continues on the next page...*

*Table continued from the previous page...*

| | |
|---|---|
| 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 PowerPC e200 which provide access to general peripherals. The architecture is also very suitable as a math co-processor for more general compute platforms.

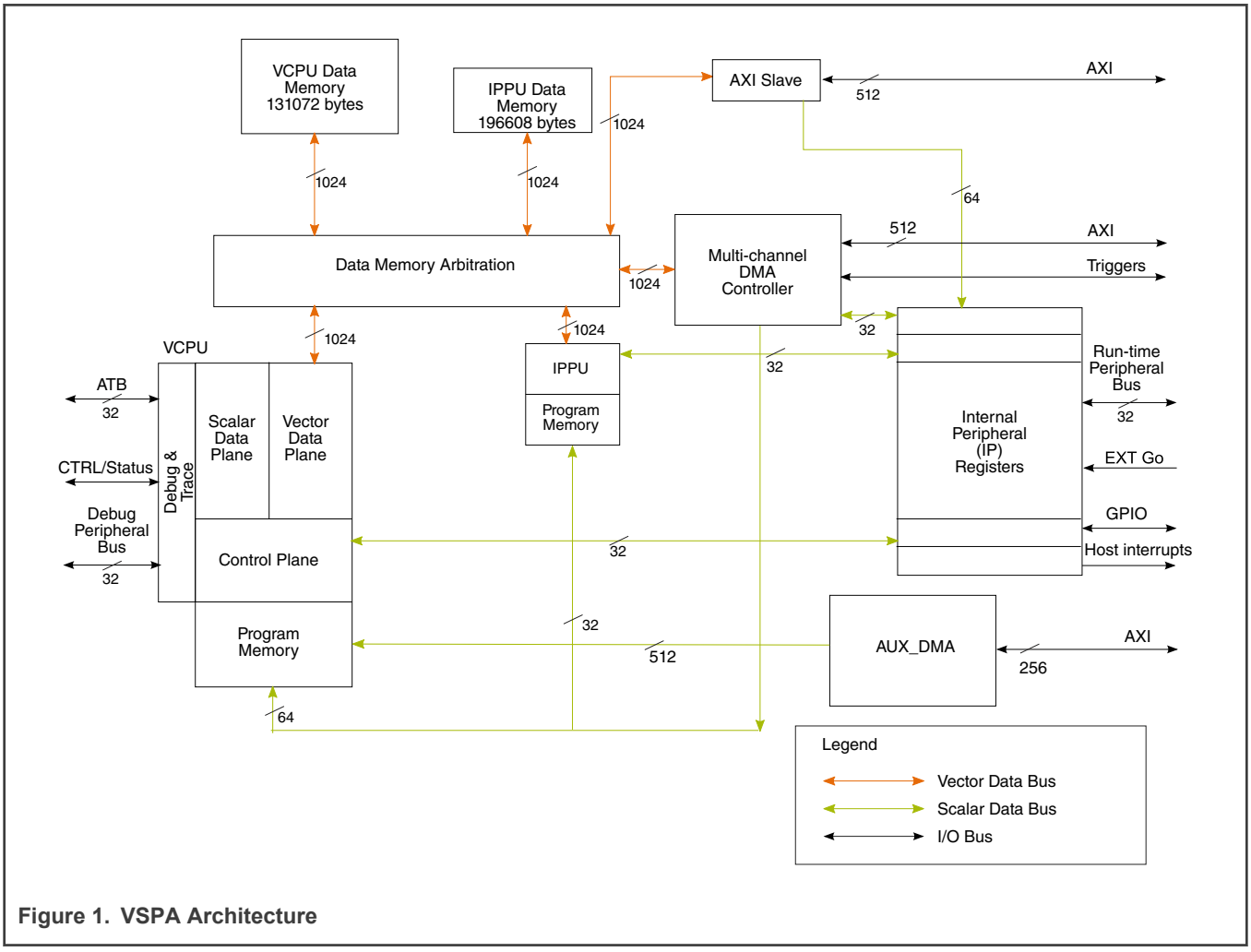



**Figure 1. VSPA Architecture**

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<br>• 64 SP RMAC/RMAD operations per clock cycle<br>• 128 F24 RMAC/RMAD operations per clock cycle<br>• Vector sqrt(\|x\|), 1/x, 1/sqrt(\|x\|) operations<br>• Vector NCO operations<br>• Vector spreading/scrambling code generator for WCDMA<br>• Independent vector compare engine<br>• Multi-port, 8-line vector register array<br>• 4 vector predicate action/capture registers, 128 bits each<br>• Transparent intra-vector permutations on each VRA port<br>• 2 vector rotate units<br>• 256-bit vector sign register | • 32-bit data plane<br>• Full-featured arithmetic and logic unit<br>• 12 scalar registers<br>• Stack pointer<br>• 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 (PowerPC e200) 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 AUX DMA

The Auxiliary Direct Memory Access controller (AUX_DMA) is an special purpose DMA used solely to load VCPU PMEM. The AUX_DMA is activated by IP register writes, and it can be configured and started by either the VCPU or the host (PowerPC e200). The AUX_DMA has its own private AXI bus, so using it does not directly interfere with the VSPA DMA.

Another feature of the AUX_DMA is that it can overlay new program code onto the VCPU PMEM while the VCPU is running, without slowing down or interfering with the operation of the VCPU in any way. This is made possible by using a 512-bit VCPU PMEM read buffer. The VCPU PMEM read buffer is loaded from PMEM whenever the VCPU does an instruction fetch targeting an address that does not already reside in the read buffer. So, the VCPU can do multiple instruction fetches per VCPU PMEM read since the instruction fetches are 64 bits, and the read buffer width is 512-bits. The VCPU has priority access to the PMEM, but the read buffer allows the AUX_DMA to write to PMEM while the VCPU simultaneously fetches instructions from the read buffer.

### 2.1.5 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.

### 2.1.6 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 327680 bytes of memory, allocated as 131072 bytes for the VCPU data memory (VDMEM) and 196608 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 327680 are dependent on the chip level memory integration. Such accesses should be avoided as they may cause the memory to be corrupted.

### 2.1.7 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 |
| ... | VCPU |
| ... | VCPU parallel load/store |
| Lowest | Debug |

### 2.1.8 Debug and trace unit

This unit is responsible for invasive debug (break, stop, step, go, and so on) and non-invasive program and data trace of the VCPU core.

# 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.

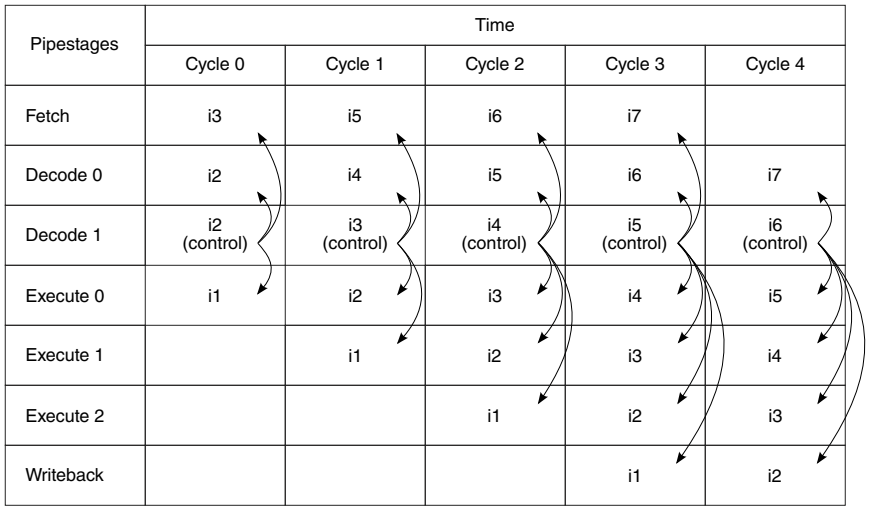



**Figure 3. Pipeline Flow for a VLIW Engine**

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

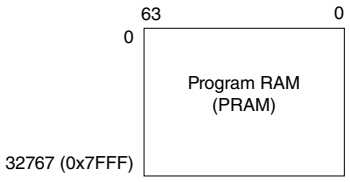



**Figure 4. Program memory address space**

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

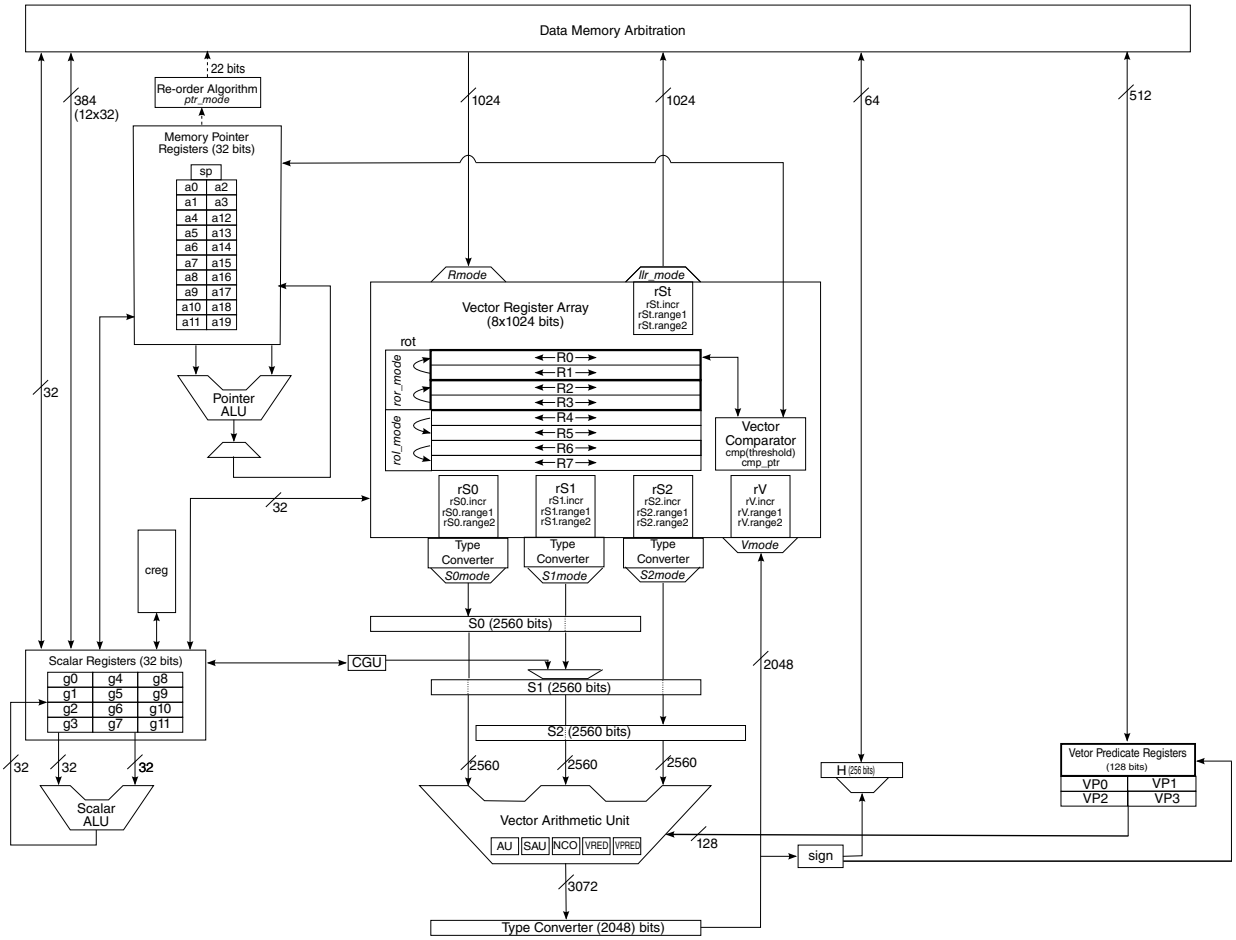



**Figure 5. VCPU Scalar & Vector Data Planes**

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.<br>• The array is addressed as a one-dimensional array in half-word increments.<br>• Each register contains 64 half-words. |
| S0*mode*<br>S1*mode*<br>S2*mode* | VAU Operand Source Register Muxes | Used for intra-vector permutation of data read from the VRA.<br>**NOTE**<br>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*<br>llr8_*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).<br>• Each AU can perform a single complex operation or 4 real operations.<br>• 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:<br>R0R1-combined, R0-only, R1-only, R2R3-combined, R2-only, or R3-only.<br>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. |
| CGU | Code Generation Unit | Generates binary sequences that are useful for WCDMA spreading and scrambling 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:<br>• S0 read-port<br>• S1 read-port |

*Table continues on the next page...*

**Table 3. VSPA Functional Units (continued)**

| Unit | | Descriptions |
|---|---|---|
| | | • S2 read-port<br>• DMEM store read-port<br>• VAU output write-port<br>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.<br>• 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.<br>• Provides storage for up to 4 VAU operations in a single register.<br>• 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 byte in the DMEM. The first DMEM line starts at address 0; the second DMEM line starts at address 128; the third DMEM line starts at address 256 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) byte 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 (2 bytes) and the imaginary part occupies the higher 16 bits (2 bytes).

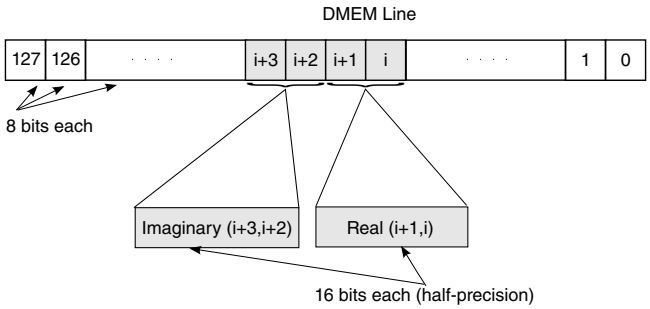



**Figure 6. Data Packing in a DMEM Line**

## 3.2.2 Data memory pointers

DMEM is addressed by pointer registers. Each pointer is 32-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 byte 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.It is capable of doing math (add/sub) on the full 32 bits of the a*X* registers, unless modulo addressing is enabled (affects a0-a3 only).

Instructions which add or subtract immediate data have a maximum of 19 bits (some Op types have smaller fields).

### 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 22-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 |

*Table continues on the next page...*

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

| a*X* Registers | Modulo addressing support | Modulo buffer registers |
|---|---|---|
| a2 | Yes | range_a2 |
| a3 | Yes | range_a3 |
| 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.

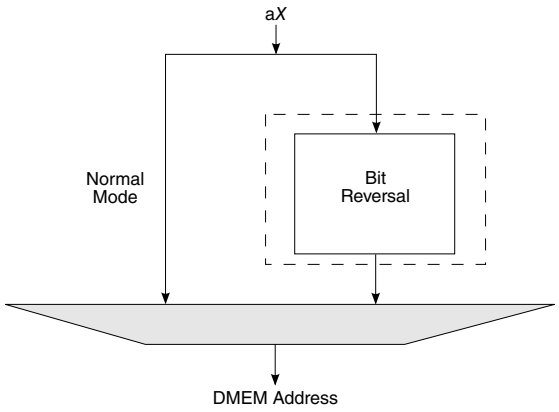



**Figure 7. Pointer reordering algorithm block diagram**

### 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*[20:6], a*X*[1:5]} |
| 64 | a*X*[6:1] | {a*X*[20:7], a*X*[1:6]} |
| 128 | a*X*[7:1] | {a*X*[20:8], a*X*[1:7]} |
| 256 | a*X*[8:1] | {a*X*[20:9], a*X*[1:8]} |
| 512 | a*X*[9:1] | {a*X*[20:10], a*X*[1:9]} |
| 1024 | a*X*[10:1] | {a*X*[20:11], a*X*[1:10]} |
| 2048 | a*X*[11:1] | {a*X*[20:12], a*X*[1:11]} |
| 4096 | a*X*[12:1] | {a*X*[20:13], a*X*[1:12]} |
| 8192 | a*X*[13:1] | {a*X*[20:14], a*X*[1:13]} |
| 16384 | a*X*[14:1] | {a*X*[20:15], a*X*[1:14]} |
| 32768 | a*X*[15:1] | {a*X*[20:16], a*X*[1:15]} |
| 65536 | a*X*[16:1] | {a*X*[20:17], 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.

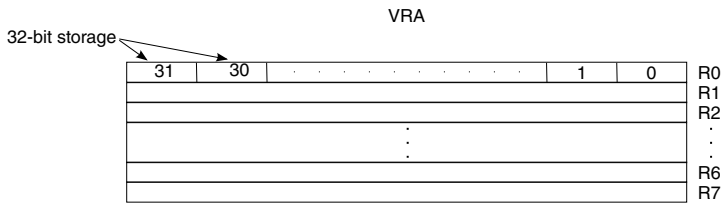



**Figure 8. Data Packing in the VRA**

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.

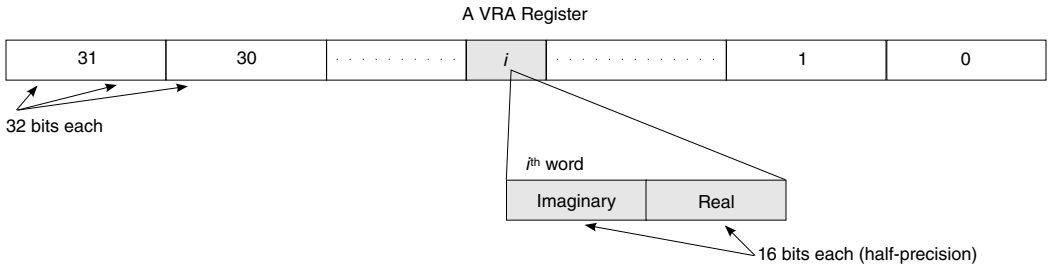



**Figure 9. Data packing in a VRA register line**

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)<br>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)<br>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.

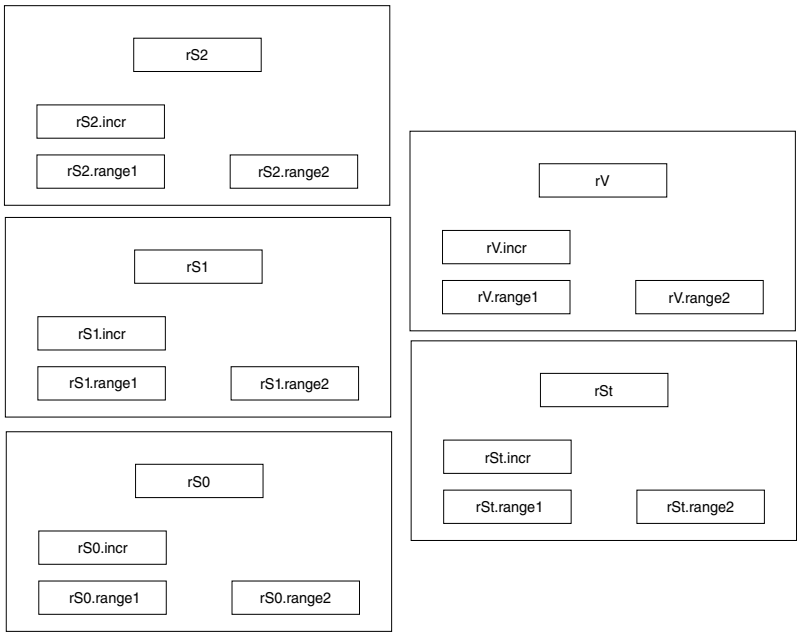



**Figure 10. VRA pointer register sets**

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 byte. The valid address space for DMEM ranges from 0 through 2097151, where each address in this space points to a unique byte.
- 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 21-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.

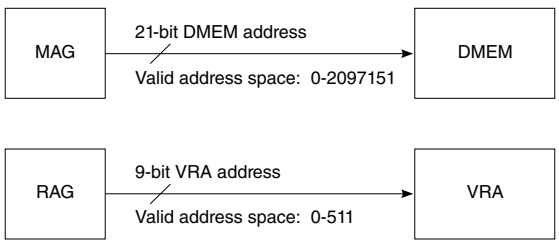



**Figure 11. Address spaces for DMEM and VRA**

### 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:

- Single register shift right for R0, R1, R2, and R3.
- Two-register rotate right for R0R1 and R2R3 where two registers are connected in a rotate right operation.
- Three-register rotate right for R0R1R2 and R0R1R3 where three registers are connected in a rotate right operation.
- Single register shift left for R4, R5, R6, and R7.
- Two-register rotate left for R4R5 and R6R7 where two registers are connected in a rotate left operation.
- Three-register rotate left for R4R5R6 and R4R5R7 where three registers are connected in a rotate left operation.

### 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 2560-bits wide and contains 64 single precision variables or 128 20 bit floating point values. The source registers only store 20 bits, because that is all that is required to prevent loss of precision when converting from fixed 16 or HP. However, internally the VAU uses a 24 bit floating point format.

### 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 24 bit precision, 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. When converting from fixed 16 or HP to real F24, the source register will load data from two VRA registers. In this case the rS0, rS1, or rS2 must point to an even VRA register (R0, R2, R4, or R6), and data will also be taken from the next higher odd R register. 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 in SP mode, and 2 complex operations or 8 real operations in F24 mode. It has a pipeline latency of 4 clock cycles for SP and F24 operations. One Pair of AUs can implement a radix-2 butterfly operation. There are 16 AUs that support single precision and F24 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
- 128 F24 real linear operations per clock cycle
- 32 F24 complex linear 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 and F24 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 and F24 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, three or 18 cycle latency. For specific details on latency see Table 15.

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 Vector predication

The VSPA VCPU provides support for vector predication of real-mode vector instructions. This essentially allows condition bit tests to control on an element by element basis, whether a vector instruction is executed on a given element. If the condition test instruction is not executed, an alternate action will be taken. For example:

```
rmad.vpz; // IF (VP==0) rmad;  ELSE alternate action;
```

This will execute an rmad (real-mode multiply-add) on all of the vector elements that have a corresponding vector predicate mask bit that is zero. For the vector elements that have a corresponding vector predicate mask bit that is one, an alternate output (action) will be generated.

There are a range of alternate outputs that can be chosen, including:

- hold state (only available for rmac and rmac.sau instructions)
- force 0 (S0* 0+ 0)
- force +1 (S0* 0+ 1)
- force -1 (S0* 0+ -1)
- multiply only (S0* S1+ 0)
- add only (S0* 1+ S2)
- forward S0 (S0* 1+ 0)
- forward -S0 (S0* -1+ 0)
- forward S2 (S0* 0+ -S2)
- forward -S2 (S0* 0+ -S2)
- subtract only (S0* 1+ -S2)

These alternate actions are produced by modifying the S1 and S2 inputs to each AU if the predicate fails the test. This produces the same pipeline delay for all AU outputs, regardless of the predicate state.

A vector AU operation can also trigger the capture of predicate condition bits. The per-AU state information that is captured can be AUout == 0, AUout == negative, or, both. For example:

```
rmad.uvp; // do an rmad, and capture flags from the result into the selected VP register(s)
```

The predicate bits are held in 4 VP registers. Each register is 128 bits wide. These registers have write pointers (capture) and read pointers (action) that can auto-increment in various ways, and they can be quickly and efficiently stored to DMEM or loaded from DMEM. They can also be quickly and easily manipulated with common boolean functions AND, OR, XOR, plus clear and 1's complement.

### 3.2.10 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 F24, single precision, depending on the state of *AUprec*. When writing data after a real F24 operation and converting to fixed 16 or hp, two consecutive VRA registers are written. In this case rV must point to an even R register (R0, R2, R4, R6), and the next higher odd R register will also be written. When writing data after a complex F24 operation and converting to fixed 16 or hp, an entire single VRA register will be written.

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.

### 3.2.11 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). This feature is only available when the AUprec is configured for single precision.

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.

### 3.2.12 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.

### 3.2.13 Code generation unit

The Code Generation Unit (CGU) can be used to generate 3GPP/ LTE scrambling sequences, and also to generate scrambling and spreading codes used in W-CDMA. The output bus of the CGU contains 32 parallel complex samples, where each sample is 2 bits wide representing 1, -1, j, and -j. The CGU output can be used to modify the data being loaded into the S1 register. The operation performed for each of the complex elements is

S1 = (S1Mux output) x (CGU output)

Note that because the CGU output can only be 1, -1, j, and -j, the multiplication is simply involving sign inversion and real/imag swapping.

The CGU mode of operation (that is, what sequence to generate) is configured using the 'set.cgu ...' instruction. The supported sequences include:

- LTE scrambling (gold) sequence
- 3G W-CDMA uplink scrambling sequences
- 3G W-CDMA downlink scrambling sequences
- OVSF codes

The CGU output can also be configured to be a product of an OVSF code and one of the scrambling sequences.

### 3.2.14 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, exponent, sign extend, zero extend, min, max, count, floating point reciprocal |
| 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<br>• Between a scalar register and an IP register<br>• Between a scalar register and a pointer storage register<br>• Between a scalar register and a VRA element.<br>• Between scalar registers and operand generator registers (such as the NCO and CGU) |

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, CGU 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.

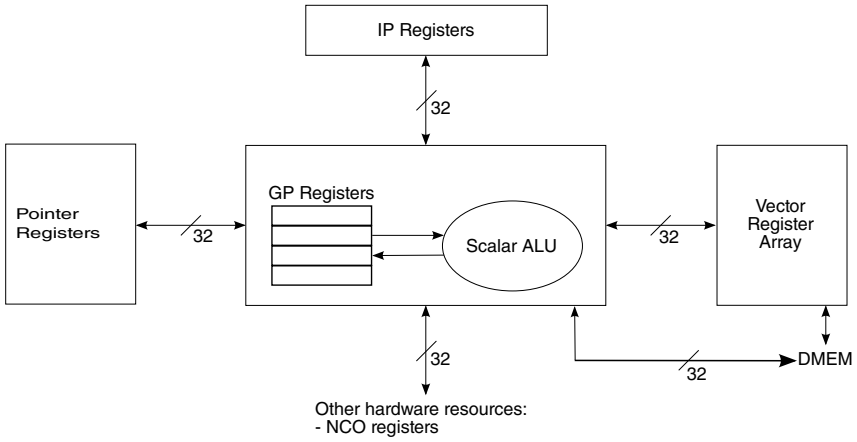



**Figure 12. Scalar ALU connectivity**

## 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 |
| F24 | 24 bit floating 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.

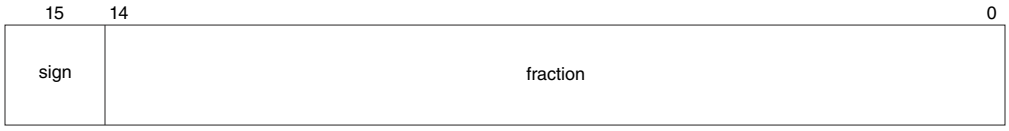



**Figure 13. VSPA Half-fixed data precision**

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 24 bit floating point (F24) representation is shown in Figure 14. It has 1 sign bit, 5 exponent bits, and 18 mantissa bits, not including the implicit 1 in the mantissa.

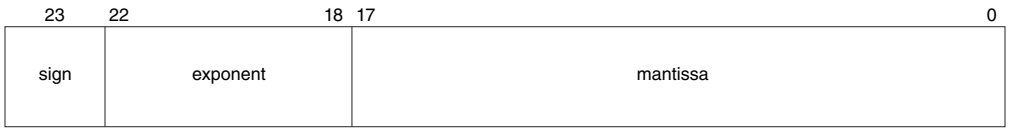



**Figure 14. VSPA 24 bit floating point data precision**

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 |
| Number of (base) digits in the mantissa | t | 24 | 19 | - |
| 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.

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 are 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 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="13">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 - 27</th><th>26 - 24</th><th>23 - 6</th><th>5 - 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="5">OpX</td><td colspan="6">OpD</td><td colspan="2" rowspan="6">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>OpVp</td><td>OpV</td><td>OpVau[1]</td></tr>
<tr><td>Format 2</td><td>0</td><td>1</td><td>0</td><td colspan="4">OpC</td><td colspan="2" rowspan="2">OpV</td></tr>
<tr><td>Format 1</td><td>0</td><td>1</td><td>1</td><td>OpB</td><td colspan="3">OpA</td></tr>
<tr><td>Format 1.1</td><td>0</td><td>1</td><td>1</td><td>OpVp</td><td colspan="3">OpA</td><td>OpV</td><td>OpVau[2]</td></tr>
<tr><td>Halfword</td><td>1</td><td></td><td colspan="2">OpX</td><td colspan="2">OpS-upper</td><td colspan="4">OpS-lower</td></tr>
</table>

1. OpVau instructions may be predicated by the OpVp bit field in format 3
2. OpVau instructions may be predicated by the OpVp bit field in format 1.1

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 1.1 is the same as format 1, with an exception. There is a specific OpB encoding (bits 59:54 will be set to 0b011011) that enables the use of the OpVp field, allowing the OpVau instructions to be predicated.
- 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). It also inclusions the OpVp field allowing the OpVau instructions to be predicated. 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, 1.1, 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[1] | OpVd | OpVrot |
|---|---|---|---|---|---|---|

1. OpVau instructions may be predicated by the OpVp bit field in formats 1.1 & 3

## 4.3 Internal operand generators

The ISA supports some internal operand generation that is useful in communications digital signal processing. Refer Vector NCO and Code generation unit 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. |
| [ * ] | 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.<br>**NOTE**<br>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}<br>See Table 23 for a description of the creg registers. |

*Table continues on the next page...*

**Table 14. VSPA instruction set mnemonic conventions (continued)**

| Convention | Meanings |
|---|---|
| sp | Stack pointer |
| 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} |
| llr8_mode | Bit Log Likelihood Ratio (LLR) compression mode, where llr8_mode∈{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 creg15[0]=0<br>*S0mode*∈{S0hlinecplx, S0fft1, S0fft2, S0fft3, S0fft4, S0fftn} with creg15[0]=1 |
| *S1mode* | *S1mode*∈{S1hlinecplx, S1straight, S1cplx1, S1real1, S1real_conj, S1zeros, S1cplx_conj, S1nco, S1qline, S1i2i1r2r1, S1udfr, S1r2c, S1r2c_conj, S1r2c_im0, S1r2c_re0, S1cgu_i2i1r2r1, S1cgu_hlinecplx, S1cgu_r2c_im0, S1cgu_r2c_re0, S1interp2nr, S1interp2nc} |
| *S2mode* | *S2mode*∈{S2hlinecplx, S2straight, S2cplx1, S2real1, S2zeros, S2i1r1i1r1, S2i1i2r1r2} with creg15[0]=0 |

*Table continues on the next page...*

**Table 14. VSPA instruction set mnemonic conventions (continued)**

| Convention | Meanings |
|---|---|
| | *S2mode*∈{S2hlinecplx, S2fft1, S2fft2, S2fft3, S2fft4, S2fftn} with creg15[0]=1 |
| *Rmode* | *Rmode*∈{normal, zeros, l2l, h2h, h2l_l2h, l2h_h2l, h2l, l2h, replace_h, replace_l, qam} |
| *Vmode* | *Vmode*∈{hlinecplx, even, fftn, fn, fn1, straight} with creg15[2])=0<br>*Vmode*∈{hlinecplx, fft1, fft2, fft3, fft4, fft5} with creg15[2]=1 |
| *S0prec* | *S0prec*∈{half_fixed, half, single} |
| *S1prec* | *S1prec*∈{half_fixed, half, single} |
| *S2prec* | *S2prec*∈{half_fixed, half, single} |
| *AUprec* | *AUprec*∈{padd, f24, single, paddF24} |
| *Vprec* | *Vprec*∈{half_fixed, half, single} |
| *rot_mode* | *ror_mode*∈{<br>R0R1r1, R0R1r2, R0R1r4, R0R1r8, R0R1rND1, R0R1rND2, R0R1rND4,<br>R0r1, R0r2, R0r4, R0r8, R0rND1, R0rND2, R0rND4,<br>R1r1, R1r2, R1r4, R1r8, R1rND1, R1rND2, R1rND4,<br>R2R3r1, R2R3r2, R2R3r4, R2R3r8, R2R3rND1, R2R3rND2, R2R3rND4,<br>R2r1, R2r2, R2r4, R2r8, R2rND1, R2rND2, R2rND4,<br>R3r1, R3r2, R3r4, R3r8, R3rND1, R3rND2, R3rND4} with creg(26) = 0<br>*ror_mode*∈{<br>R0R1R2r1, R0R1R2r2, R0R1R2r4, R0R1R2r8, R0R1rND1, R0R1rND2, R0R1rND4,<br>R0r1, R0r2, R0r4, R0r8, R0rND1, R0rND2, R0rND4,<br>R1r1, R1r2, R1r4, R1r8, R1rND1, R1rND2, R1rND4,<br>R0R1R3r1, R0R1R3r2, R0R1R3r4, R0R1R3r8, R2R3rND1, R2R3rND2, R2R3rND4,<br>R2r1, R2r2, R2r4, R2r8, R2rND1, R2rND2, R2rND4,<br>R3r1, R3r2, R3r4, R3r8, R3rND1, R3rND2, R3rND} with creg(26) = 1<br>*rol_mode*∈{<br>R4R5l1, R4R5l2, R4R5l4, R4R5l8<br>R4l1, R4l2, R4l4, R4l8<br>R5l1, R5l2, R5l4, R5l8<br>R6R7l1, R6R7l2, R6R7l4, R6R7l8<br>R6l1, R6l2, R6l4, R6l8<br>R7l1, R7l2, R7l4, R7l8} with creg(26) = 0<br>*rol_mode*∈{ |

*Table continues on the next page...*

**Table 14. VSPA instruction set mnemonic conventions (continued)**

| Convention | Meanings |
|---|---|
| | R4R5R6I1, R4R5R6I2, R4R5R6I4, R4R5R6I8<br>R4I1, R4I2, R4I4, R4I8<br>R5I1, R5I2, R5I4, R5I8<br>R4R5R7I1, R4R5R7I2, R4R5R7I4, R4R5R7I8<br>R6I1, R6I2, R6I4, R6I8<br>R7I1, R7I2, R7I4, R7I8} with creg(26) = 1 |
| *cgu_mode* | *cgu_mode*∈{dl_sc, dl_sc_singles, ul_sc_long, ul_sc_short, lte_gold} |
| *cgu_reg* | *cgu_reg*∈{dl_sc_x, dl_sc_x_bak, dl_sc_y, ul_sc_x, ul_sc_y, ul_sc_short, ovsf_num,gold_x1, gold_x2, ovsf_dl_sc_y, ovsf_dl_sc_y_bak} |
| *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<br>Load (from memory) instructions<br>Store (to memory) instructions<br>Load/Store instructions<br>Vector Register Array (VRA) instructions<br>Extrema Search Unit instructions<br>Vector Arithmetic Unit Source Register instructions<br>Vector Floating Point Arithmetic Unit instructions<br>Vector Arithmetic Unit Write-Results instructions<br>Vector Sign Register (H) instructions<br>Vector Predicate Register (VP) instructions | | | | | Scalar Arithmetic instructions<br>Logical Operation instructions<br>Scalar Register-Set (g*X*) instructions<br>Loop instructions<br>Scalar Compare instructions<br>Control Flow and State instructions<br>Stack instructions<br>Internal Peripheral Control instructions<br>Operand Generator instructions<br>Miscellaneous instructions |
| Data Memory Pointer instructions | | | | | VSPA instructions quick reference |
| mv | a*U* | Iu22 | 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*. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| 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*. |
| exgS | a*X* | ag*Y* | OpS | 1 | Exchange a*X* & ag*Y* |
| | a*U* | sp | | | Exchange a*U* & sp |
| add | a*U* | a*V*,a*W* | OpA | 1 | Add a*V* and a*W* registers and write the result to a*U*. |
| sub | a*U* | a*V*,a*W* | OpA | 1 | Subtract a*V* from a*W* and write the result to a*U*. |
| addC | a*V* | a*U*,ls19 | OpC | 1 | Add a 19-bit signed immediate value to a*U* and write the result to a*V*. |
| add | a*U* | sp,ls19 | | | Add a 19-bit signed immediate value to stack pointer and write the result to a*U*. |
| addA(.laddr) | a*U*, ls9 | | OpA | 1 | Add a 9-bit signed immediate value to a*U* and write the result to a*U*. If .laddr is included, update is by lines. |
| set.range | a*Y* | ag*X*,lu22 | OpC | 1 | Defines the beginning and ending address for a circular buffer in DMEM accessed by a*Y*.<br>• Beginning address is stored in ag*X*.<br>• Ending address is calculated, ag*X* + l - 1. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | **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 |
| ldA | [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*. |
| ldA(.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). |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| 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. |
| ldB | [ag*X*]+/-ag*Y* | | OpB | 3 | Load vector from [ag*X*], post increment/ decrement ag*X* by ag*Y*. If the ldB instruction is used in parallel with an OpA st instruction it will cause an 1 additional cycle delay in the VCPU program execution if both the load and store access the same DMEM bank. |
| ldB(.laddr) | [ag*X*]+ls5 | | | | Load vector from [ag*X*], post increment ag*X* by ls. If the ldB instruction is used in parallel with an OpA st instruction it will cause an 1 additional cycle delay in the VCPU program execution if both the load and store access the same DMEM bank. |
| 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.x2 | g*Z* | [ag*X*]+/-ag*Y* | | | 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 ag*X* 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.x2 | g*Z* | [ag*X*+/-ag*Y*] | | | 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*. |

*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. |
| ld.u.x2 | 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*. |
| ldS(.laddr) | g*X* | [ag*Y*+ls9] | | | 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*+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 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 g*Z*, otherwise the value is zero extended. The contents of ag*X* 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 g*Z*, 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 |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | extended into g*Z*, 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*. |
| ldbS(.laddr)(.s) | g*Z* | [ag*X*+*l*s] | OpS | 4 | Load byte from [ag*X*+ls] (pre-increment memory pointer) into g*Z*, optionally sign extend. |
| ldbS(.laddr).u(.s) | g*Z* | [ag*X*+*l*s] | | | Load byte from [ag*X*+ls] (pre-increment & update memory pointer) into g*Z*, optionally sign extend. |
| ldbS(.laddr)(.s) | g*Z* | [ag*X*]+*l*s | | | Load byte from [ag*X*]+ls (post increment & update memory pointer) into g*Z*, optionally sign extend. |
| ldb(.s) | g*Z* | [ag*X*+/-ag*Y*] | | | Load byte from [ag*X*+ag*Y*] (pre-increment memory pointer) into g*Z*, optionally sign extend. |
| ldb.u(.s) | g*Z* | [ag*X*+/-ag*Y*] | | | Load byte from [ag*X*+ag*Y*] (pre-increment & update memory pointer) into g*Z*, optionally sign extend. |
| ldb(.s) | g*Z* | [ag*X*]+/-ag*Y* | | | Load byte from [ag*X*]+ag*Y* (post increment & update memory pointer) into g*Z*, optionally sign extend. |
| ldC | [ag*X*]+ls18 | | 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*]+ls18 | | 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*+ls18] | | | 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. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| ldC.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*. |
| 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. |
| ldbC | g*Y* | [ag*X*]+ls18 | | | Load byte from [ag*X*]+ls (post increment & update memory pointer) into g*Z*, zero extend. |
| ldbC | g*Y* | [ag*X*+ls18] | | | Load byte from [ag*X*+ls] (pre-increment memory pointer) into g*Z*, zero extend. |
| ldbC.u | g*Y* | [ag*X*+ls18] | | | Load byte from [ag*X*+ls] (pre-increment & update memory pointer) into g*Z*, zero extend. |
| ldbC.s | g*Y* | [ag*X*]+ls18 | | | Load byte from [ag*X*]+ls (post increment & update memory pointer) into g*Z*, sign extend. |
| ldbC.s | g*Y* | [ag*X*+ls18] | | | Load byte from [ag*X*+ls] (pre-increment memory pointer) into g*Z*, sign extend. |
| ldbC.u.s | g*Y* | [ag*X*+ls18] | | | Load byte from [ag*X*+ls] (pre-increment & update memory pointer) into g*Z*, sign extend. |
| ldhC | g*Y* | [ag*X*]+ls18 | OpC | 4 | 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. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| 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 | OpC | 4 | 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 contents as a sign extended 32-bit value into g*Y*. The |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | 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. |
| ldw | g*X* | lu22 | OpD | 4 | Load a 32-bit scalar from DMEM using an immediate address and store the contents into g*X*. |
| ldh(.s) | g*X* | lu22 | | | Load a 16-bit scalar from DMEM using an immediate address and store the contents into g*X*. |
| ldb(.s) | g*X* | lu22 | | | Load byte from memory address into g*X*. |
| 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. |
| st.uline.llr8_mode | [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 |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | ST_UL_VEC_LEN IP register. The pointer will be post incremented by the number of elements stored. llr mode is used for bit log-likelihood-ratio (LLR) type conversion, read a vector from R[rSt], compress the data by 1/2 by converting the elements to fixed point, then quantize each to 8 bit. |
| 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. |
| stA(.laddr).w | [ag*X*]+ls9 | | | | 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.x2 | [ag*X*]+/-ag*Y* | 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.x2 | [ag*X*+/-ag*Y*] | 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. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| st.u.x2 | [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*. |
| 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*. |
| 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*. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| stbS(.laddr) | [ag*X*+*I*s] | g*Z* | OpS | 1 | Store byte from g*Z* into [ag*X*+Is] (pre-increment memory pointer). |
| stbS(.laddr).u | [ag*X*+*I*s] | g*Z* | | | Store byte from g*Z* into [ag*X*+Is] (pre-increment & update memory pointer). |
| stbS(.laddr) | [ag*X*]+*I*s | g*Z* | | | Store byte from g*Z* into [ag*X*]+Is (post increment & update memory pointer). |
| stb | [ag*X*+/-ag*Y*] | g*Z* | | | Store byte from g*Z* into [ag*X*+ag*Y*] (pre-increment memory pointer). |
| stb.u | [ag*X*+/-ag*Y*] | g*Z* | | | Store byte from g*Z* into [ag*X*+ag*Y*] (pre-increment & update memory pointer). |
| stb | [ag*X*]+/-ag*Y* | g*Z* | | | Store byte from g*Z* into [ag*X*]+ag*Y* (post increment & update memory pointer). |
| 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.<br>**NOTE**<br>Format is real, imaginary. |
| stC | [ag*X*]+Is18 | | 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*]+Is18 | | | | 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*]+Is18 | 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*+Is18] | 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*+Is18] | g*Y* | | | Alternative form of the previous instruction where the address used in the memory access is stored back into the pointer ag*X*. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| stC | [sp+Is18] | 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+Is18] | 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]+Is18 | 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*]+Is18 | g*Y* | OpC | 1 | Store a 16-bit scalar from g*Y* into DMEM using a pointer. Post-increment by a signed, immediate offset. |
| sthC | [ag*X*+Is18] | 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. The contents of ag*X* are not modified by this instruction. |
| sthC.u | [ag*X*+Is18] | 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]+Is18 | 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+Is18] | 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+Is18] | g*Y* | | | Alternative form of the previous instruction where the address used in the memory access is stored back into the stack pointer. |
| stbC | [ag*X*]+Is18 | g*Y* | OpC | 1 | Store byte from g*Z* into [ag*X*]+Is (post increment & update memory pointer). |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| stbC | [ag*X*+Is18] | g*Y* | | | Store byte from g*Z* into [ag*X*+Is] (pre-increment memory pointer). |
| stbC.u | [ag*X*+Is18] | g*Y* | | | Store byte from g*Z* into [ag*X*+Is] (pre-increment & update memory pointer). |
| sth | [ag*X*]+Is9 | I16 | OpC | 1 | Store a 16-bit immediate value into DMEM using a pointer. Post-increment by a signed, immediate offset. |
| stb | [ag*X*]+Is9 | I8 | | | Store immediate byte into [ag*X*]+Is9 (post increment & update memory pointer). |
| 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*]+Is16 | 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 index of the leftmost 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 leftmost written element is given in g*Z*. |
| st.low | [ag*X*]+Is16 | 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 leftmost written element is given in g*Z*. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| st.high | [ag*X*]+ag*Y* | Iu8 | | | 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 rightmost written element is given by Iu8. |
| st.high | [ag*X*]-ag*Y* | Iu8 | | | 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 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. Post-increment by a signed, immediate 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 by the contents of ag*Y*. The index of the rightmost written element is given by value 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 rightmost written element is given by value in g*Z*. |
| st.high | [ag*X*]+Is16 | 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 index of the rightmost written element is given by value in g*Z*. |
| stb | Iu22 | I8 | OpD | 1 | Store immediate byte into Iu22(memory pointer). |
| sth | Iu22 | I16 | | | Store immediate half word into Iu22(memory pointer). Only upper 21 bits of byte addr is in the macro instruction, byte address must be hword aligned. Modify existing instruction to new encoding & increase pointer size. |
| stw | Iu22 | I32 | | | Store immediate word into Iu22(memory pointer). Only upper 20 bits of byte addr is in macro the instruction, byte address must |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | be word aligned. Modify existing instruction to new encoding & increase pointer size. |
| stb | Iu22 | g*X* | | | Store byte from g*X* into memory address. |
| sth | Iu22 | g*X* | | | Store a 16-bit scalar from g*X* to an immediate address in DMEM. |
| stw | Iu22 | g*X* | | | Store a 32-bit scalar from g*X* to an immediate address in DMEM. |
| Load/Store instructions | | | | | VSPA instructions quick reference |
| stld | [a0]+/-ag*X*, [a1]+/-ag*Y* | | OpA | 3 | Store a vector to DMEM using A0 as base pointer and load a vector from DMEM using A1 as base pointer. A0 is post modified by the contents of ag*X* and A1 is post modified by the contents of ag*Y*. The stld instruction will cause an 1 additional cycle delay in the VCPU program execution if both the load and store access the same DMEM bank. |
| stld.laddr | [a0]+/-a*Z*, [a1]+Is9 | | | | Store a vector to DMEM using A0 as base pointer and load a vector from DMEM using A1 as base pointer. A0 is post modified by the contents of a*Z* and A1 is post modified by the signed, immediate offset number of lines. The offset register, a*Z*, is restricted to a4-a11. The stld instruction will cause an 1 additional cycle delay in the VCPU program execution if both the load and store access the same DMEM bank. |
| stld.laddr | [a0]+Is9, [a1]+/-a*Z* | | | | Store a vector to DMEM using A0 as base pointer and load a vector from DMEM using A1 as base pointer. A0 is post modified by the signed, immediate offset number of lines and A1 is post modified by the contents of a*Z*. The offset register, a*Z*, is restricted to a4-a11. The stld instruction will cause an 1 additional cycle delay in the VCPU program execution if both the load and store access the same DMEM bank. |
| Vector Register Array (VRA) instructions | | | | | VSPA instructions quick reference |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| 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*<br>x ∈ {0,2,4,6} | | | | For modulation order M, take a M/16 line of memRead bus (64M bits), transform it into 2 full lines, and write it into R*x* and R*x +1*, where x ∈ {0,2,4,6}. The transformation implements QAM modulation. |
| ld.el_rev | R*x* | | | | Load a vector from memRead bus into Rx with element reverse |
| 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. |
| 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. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | **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 |
| 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 |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| 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. |
| setA.VRAptr | rS2rS1rS0 | Iu3S2, Iu4S1, Iu4S0 | OpA | 1 | Initialize Vector Register Array pointers rS0, rS1, rS2. |
| setA.VRAptr | rStrVrS1 | Iu3St, Iu4V, Iu3S1 | | | Initialize Vector Register Array pointers rS1, rV, rSt. |
| setA.VRAptr | rStrV | Iu3St, Iu4V | | | Initialize Vector Register Array pointers rV, rSt. |
| setA.VRAptrIP | Iu5 | Iu4 | | | Initialize Vector Register Array pointers from IP register. |
| setB.VRAptr | r*X* | Iu9 | OpB | 1 | Initialize a Vector Register Array pointer. |
| setB.VRAptr | rS2rS1rS0 | Iu3S2, Iu4S1, Iu4S0 | OpB | 1 | Initialize Vector Register Array pointers rS0, rS1, rS2. |
| setB.VRAptr | rStrVrS1 | Iu3St, Iu4V, Iu3S1 | | | Initialize Vector Register Array pointers rS1, rV, rSt. |
| setB.VRAptr | rStrV | Iu3St, Iu4V | | | Initialize Vector Register Array pointers rV, rSt. |
| setB.VRAptrIP | Iu5 | Iu4 | | | Initialize Vector Register Array pointers from IP register. |
| 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*.<br>**NOTE**<br>Format is start, wrap. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| set.VRArange2 | r*X* | Iu9, Iu9 | | | Initialize the range registers (start and wrap addresses) of the 2nd buffer for r*X*.<br>**NOTE**<br>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. |
| 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*:g*X*+1:g*X*+2:g*X*+3 | | | Fill the entire vector with the contents of g*X* through g*X*+3. |
| mv.q | [rV] | g*X*:g*X*+1:g*X*+2:g*X*+3 | | | Move the contents of g*X* through g*X*+3 into 4 full words in Rx. |
| fill.d | [rV] | g*X*:g*X*+1 | | | Fill the entire vector with the contents of g*X* through g*X*+1. |
| mv.d | [rV] | g*X*:g*X*+1 | | | Move the contents of g*X* through g*X*+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*. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| set.rot | | (lt_mode), (rt_mode) | | 0 | Configure the vector rotation unit.<br>**NOTE**<br>lt_mode and rt_mode are independent. |
| mv | R*x* | R*y* | | 2 | Move the full vector Ry into Rx.<br>**NOTE**<br>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 |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| 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 up to 2048. The search engine uses rS2 to read vector data from the VRA. In 'all' mode, N must be greater 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 result into either or both a memory pointer and scalar register. This is a multi-cycle instruction that depends on the number of elements(N) 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. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | 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. |
| | 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*<br>*S1prec*<br>*S2prec*<br>*AUprec*<br>*Vprec* | | OpB | 1<br>1<br>1<br>3<br>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 |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| rmad(.uvp)(.vpnz \| .vpz) | (AUout) | | OpVau | 4 | Multiply and add.<br>.uvp, .vpnz, .vpz are the vector predicate extensions for the OpVau instructions.<br>• .uvp: Update Vector Predicate - update the VP registers as configured by the VP_CONTROL register.<br>• .vpz: Execute the associated OpV instruction on an element if the VP register element bit is zero, otherwise, execute the alternate action specified by the VP_CONTROL register.<br>• .vpnz: Execute the associated OpV instruction on an element if the VP register element bit is one, otherwise, execute the alternate action specified by the VP_CONTROL register. |
| | (VP) | | OpVau | 5(SP) | |
| rmad.sau(.uvp)(.vpnz \| .vpz) | (AUout) | | OpVau | 4(SP) | Multiply and add, using SAU result as S1 operand.<br>.uvp, .vpnz, .vpz are the vector predicate extensions for the OpVau instructions.<br>• .uvp: Update Vector Predicate - update the VP registers as configured by the VP_CONTROL register.<br>• .vpz: Execute the associated OpV instruction on an element if the VP register element bit is zero, otherwise, execute the alternate action specified by the VP_CONTROL register.<br>• .vpnz: Execute the associated OpV instruction on an element if the VP register element bit is one, otherwise, execute the alternate action specified by the VP_CONTROL register. |
| | (VP) | | OpVau | 5(SP) | |
| rmac(.uvp)(.vpnz \| .vpz) | (AUout) | | OpVau | 4(SP) | Multiply and accumulate.<br>.uvp, .vpnz, .vpz are the vector predicate extensions for the OpVau instructions.<br>• .uvp: Update Vector Predicate - update the VP registers as configured by the VP_CONTROL register.<br>• .vpz: Execute the associated OpV instruction on an element if the VP |
| | (VP) | | OpVau | 5(SP) | |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | register element bit is zero, otherwise, execute the alternate action specified by the VP_CONTROL register.<br>• .vpnz: Execute the associated OpV instruction on an element if the VP register element bit is one, otherwise, execute the alternate action specified by the VP_CONTROL register. |
| rmac.sau(.uvp) (.vpnz \| .vpz) | (AUout) | | OpVau | 4(SP) | Multiply and accumulate, using SAU result as S1 operand.<br>.uvp, .vpnz, .vpz are the vector predicate extensions for the OpVau instructions.<br>• .uvp: Update Vector Predicate - update the VP registers as configured by the VP_CONTROL register.<br>• .vpz: Execute the associated OpV instruction on an element if the VP register element bit is zero, otherwise, execute the alternate action specified by the VP_CONTROL register.<br>• .vpnz: Execute the associated OpV instruction on an element if the VP register element bit is one, otherwise, execute the alternate action specified by the VP_CONTROL register. |
| | (VP) | | OpVau | 5(SP) | |
| cmad | | | OpVau | 4(SP) | 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(.uvp)(.vpnz \| .vpz) | (AUout) | | OpVau | 4(SP) | Multiply and add then feedback result to multiplier.<br>.uvp, .vpnz, .vpz are the vector predicate extensions for the OpVau instructions. |
| | (VP) | | OpVau | 5(SP) | |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | • .uvp: Update Vector Predicate - update the VP registers as configured by the VP_CONTROL register.<br>• .vpz: Execute the associated OpV instruction on an element if the VP register element bit is zero, otherwise, execute the alternate action specified by the VP_CONTROL register.<br>• .vpnz: Execute the associated OpV instruction on an element if the VP register element bit is one, otherwise, execute the alternate action specified by the VP_CONTROL register. |
| dif.sau | | | OpVau | 4(SP) | 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 | | | OpVsau | 2 | Iterate numerically controlled oscillator. |
| padd | | | OpVsau | 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. |
| wr.fft2 | | | | | Write AU output into the VRA. Perform shift function at WbMux for FFT 2nd stage. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| 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.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. |
| 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 optional condition is true, then sum the 64 least significant bits in the H register then add to the contents of g*X* and store |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | 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 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 |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | 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.<br>**NOTE**<br>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.<br>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.<br>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, 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*+ls9] | H | | | 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 |
|---|---|---|---|---|---|
| st | I19 | H | | 1 | Store a sign vector from H to an immediate address in DMEM.<br>**NOTE**<br>lsb of I19 is ignored if legacy_mem_addr=0. |
| Vector Predicate Register (VP) instructions | | | | | VSPA instructions quick reference |
| ldm | VPmask(Iu4) | [ag*X*-*vN*] | OpS | 4 | Load predicate register(s)from DMEM, with pre-read modification of ag*X* by 0,1,2, or 4 VP register widths, corresponding to *vN* = 0, 1, 2 and 4, respectively. |
| stm | [ag*X*]+(Iu2) | VPmask(Iu4) | | 1 | Store predicate register(s) to DMEM, with post-write modification of ag*X* by 0,1,2, or 4 VP register widths. |
| and | Vp*Z* | Vp*X*, Vp*Y* | OpS | 1 | Logically AND any 2 predicate registers, and write the result to any predicate register. |
| or | Vp*Z* | Vp*X*, Vp*Y* | | | Logically OR any 2 predicate registers, and write the result to any predicate register. |
| xor | Vp*Z* | Vp*X*, Vp*Y* | | | Logically XOR any 2 predicate registers, and write the result to any predicate register. |
| mv | H | VPx(Iu1) | OpS | 1 | Move a VP pair into H (pair is VP[1:0] or VP[3:2]). |
| mv | VPx(Iu1) | H | | | Move H into a VP pair (pair is VP[1:0] or VP[3:2]). |
| com | VPmask(Iu4) | | OpS | 1 | 1's complement of predicate register(s) (invert all bits of the selected predicates). |
| clrm | VPmask(Iu4) | | OpS | 1 | Clear predicate register(s). |
| Scalar Arithmetic instructions | | | | | VSPA instructions quick reference |
| addS(.ucc)(.cc) | g*Z* | g*X*,g*Y* | OpS | 1 | If optional condition is true, then add g*X* to 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 |
|---|---|---|---|---|---|
| | | | | | **NOTE**<br>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 78. |
| 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 78.<br>**NOTE**<br>Use of sp is allowed. sp can replace g*Z*. |
| addS(.ucc).s | (g*Z*) | g*Z*,Is16 | | | 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 78.<br>**NOTE**<br>Use of sp is allowed. sp can replace g*Z*. |
| addD(.ucc)(.cc) | g*X* | g*Y*,I32 | OpD | 1 | If optional condition 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 78. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | **NOTE** Use of sp is allowed. |
| subS(.ucc)(.cc) | g*Z* | g*X*,g*Y* | OpS | 1 | If optional condition 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 78. **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 78. **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 78. **NOTE** Use of sp is allowed. sp can replace g*Z*. |
| subD(.ucc)(.cc) | g*X* | g*Y*,l32 | OpD | 1 | If optional condition 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 |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | condition codes based on the result of the operation. See Table 78.<br>**NOTE**<br>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*.<br>**NOTE**<br>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 optional condition 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*. |
| 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 optional condition 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 optional condition is true, then g*Z* = floor (g*X*/g*Y*).<br>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*. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| 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 optional condition is true, then g*Z* = rem(g*X*/g*Y*).<br>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*. |
| 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. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| 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 g*X* by 2^n, where the integer exponent n is stored in g*Y*.<br>**NOTE**<br>The value in g*Y* 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 g*Y* 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*. |
| sign | g*X* | g*Y* | OpS | 1 | Move the SP sign into g*Z* from g*X*. |
| rotr | g*X* | g*Y*,*I* | | | Rotate right from g*Y* into g*X* by I number of bits. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| rotl | g*X* | g*Y*,*I* | | | Rotate left from g*Y* into g*X* by I number of bits. |
| rotr(.cc) | g*Z* | g*X*, g*Y* | | | If optional condition is true, rotate right from g*X* into g*Z* by g*Y*[0:5] number of bits. |
| rotl(.cc) | g*Z* | g*X*, g*Y* | | | If optional condition is true, rotate left from g*X* into g*Z* by g*Y*[0:5] number of bits. |
| mant(.cc) | g*Z* | g*X* | | | If optional condition is true, move the SP mantissa into g*Z* from g*X*. |
| exp(.cc) | g*Z* | g*X* | | | If optional condition is true, move the SP exponent into g*Z* from g*X*. |
| sexth(.cc) | g*Z* | g*X* | | | If optional condition is true, sign extend 16-bit value into g*Z* from g*X* - g*Z*={{16{g*X*[15]}},g*X*[15:0]}. |
| sextb(.cc) | g*Z* | g*X* | | | If optional condition is true, sign extend 8-bit value into g*Z* from g*X* - g*Z*={{24{g*X*[7]}},g*X*[7:0]}. |
| zexth(.cc) | g*Z* | g*X* | | | If optional condition is true, zero extend 16-bit value into g*Z* from g*X* - g*Z*={{16{0}},g*X*[15:0]}. |
| zextb(.cc) | g*Z* | g*X* | | | If optional condition is true, zero extend 8-bit value into g*Z* from g*X* - g*Z*={{24{0}},g*X*[7:0]}. |
| min | g*Z* | g*X*, g*Y* | OpS | 1 | Move the smaller of g*X* & g*Y* into g*Z*. |
| max | g*Z* | g*X*, g*Y* | | | Move the larger of g*X* & g*Y* into g*Z*. |
| abs(.cc) | g*Z* | g*X* | | | If optional condition is true, move the absolute value of g*X* into g*Z*. g*Z*=\|g*X*\|. g*X* & g*Z* contain integer values. |
| cntl | g*Z* | g*X* | | | Count leading 0's in g*X*, return result in g*Z*. |
| cnto | g*Z* | g*X* | | | Count the number of 1's in g*X*, return result in g*Z*. |
| cntz | g*Z* | g*X* | | | Count the number of 0's in g*Y*, return result in g*X*. |
| sum1 | g*Z* | g*X* | | | Alternate mnemonic for cnto. Count the number of 1's in g*X*, return result in g*Z*. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| mac(.cc) | g*Z* | g*X*, g*Y* | OpS | 4 | Multiply and accumulate. If optional condition is true, multiply g*X* with g*Y* and add to the result in g*Z*. g*Z*, g*X* and g*Y* contain integer values. |
| 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 optional condition 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 optional condition 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 optional condition is true, clear bit number Iu5 from g*X* and write result to g*Z*. g*X* is not changed. |
| and(.cc) | g*Z* | g*X*,g*Y* | OpS | 1 | If optional condition is true, then perform the logical AND of g*X* with g*Y* and store the result in g*Z*.<br>**NOTE**<br>Use of sp is allowed. |
| 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.<br>**NOTE**<br>Use of sp is allowed. |
| andD(.cc) | g*X* | g*Y*,I32 | OpD | 1 | If optional condition is true, then perform logical AND of long immediate with g*Y*and store the result in g*X*.<br>**NOTE**<br>Use of sp is allowed. |
| or(.cc) | g*Z* | g*X*,g*Y* | OpS | 1 | If optional condition is true, then perform the logical OR of g*X* with 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 |
|---|---|---|---|---|---|
| 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 optional condition 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 optional condition 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 optional condition is true, then perform the logical XOR of g*X* with g*Y* and store the result in g*Z*. |
| bxor(.cc) | g*Z* | g*X*,Iu5 | | | If optional condition is true, toggle bit Iu5 in g*X*, write result to 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 optional condition 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 optional condition 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*. |
| sr(.cc).s | g*Z* | g*X*,g*Y* | | | If optional condition 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 optional condition 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*. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| lfsr | (g*X*) | g*X*,g*Y* | OpB | 1 | Linear feedback shift register operation. Shift-left the contents of g*X* by one bit position, then replace its least significant bit with the bit-wise XOR of g*Y*. |
| 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 (g*X*) instructions | | | | | VSPA instructions quick reference |
| mv(.cc) | g*Z* | g*X* | OpS | 1 | If optional condition is true, then move g*X* to g*Z*. |
| exg(.cc) | g*Z* | g*X* | OpS | 1 | If optional condition is true, exchange g*Z* and g*X*.<br>**NOTE**<br>Use of SP is allowed as a source operand. |
| 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*. |
| mvD(.cc) | g*X* | I32 | OpD | 1 | If optional condition is true, then move an immediate scalar into g*X*. |
| mvh.s(.cc) | g*Z* | g*X* | OpS | 1 | If optional condition is true, then sign extend the least significant 16 bits of g*X* and move to g*Z*. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| 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 optional condition is true, then move current pc to g*Z*. |
| mv(.cc) | g*Z* | quot | | | If optional condition is true, then move the quotient result of the previous modulo operation to g*Z*. |
| mv(.cc) | g*Z* | rem | | | If optional condition 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.<br>• 1st argument is loop iteration count.<br>• 2nd argument is size. |
| setC.loop | Iu16 | | | | Set the number of iterations using an immediate number. |
| setC.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*. |
| setB.loop | Iu10 | | OpB | 2 | 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* | | | | Set the number of iterations using the contents of ag*X*. If ag*X* = 0, the number of iterations will be 65536. |
| setB.loop | ag*X*,ls5 | | | 2 | 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.<br>**NOTE**<br>Assembler directive to indicate end of the loop. |
| Scalar Compare instructions | | | | | VSPA instructions quick reference |
| cmp | a*U*,lu22 | | OpC | 2 | Compare a data memory pointer with an immediate number.<br>**NOTE**<br>The second cycle in 'cmp' is used to set up a conditional instruction. |
| cmpD(.cc) | g*Y*,l32 | | OpD | 1 | If optional condition is true, then compare g*Y* with an immediate scalar.<br>**NOTE**<br>Use of sp is allowed. |
| cmpS.z | g*Z*,lu16 | | OpS | 1 | Compare g*Z* with a short unsigned immediate scalar. |
| cmpS.s | g*X*,ls16 | | | | Sign extend an immediate short scalar, then compare it with g*X*. |
| cmp(.cc) | g*X*,g*Y* | | | | If optional condition is true, then compare 2 scalar registers. Similar to subtract but without a destination register. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | **NOTE** Use of sp is allowed. |
| Control Flow and State instructions | | | | | VSPA instructions quick reference |
| jmp(.cc) | g*X* | | OpC | 3 | If optional condition is true, then jump to the address in g*X*. |
| jsr(.cc) | g*X* | | | | If optional condition is true, then jsr to the address in g*X*. |
| jmp(.cc) | Iu25 | | | | If optional condition is true, then jump to an immediate address. |
| jsr(.cc) | Iu25 | | | | If optional condition is true, then jsr to an immediate address. |
| jmp(.cc) | [pc+Is25] | | | | If optional condition is true, then jump to a PC relative (PC+Is25) address. |
| jsr(.cc) | [pc+Is25] | | | | If optional condition is true, then jsr to a PC relative (PC+Is25) address. |
| jmp(.cc) | [pc+g*X*] | | | | If optional condition is true, then jump to a PC relative (PC+g*X*) address. |
| jsr(.cc) | [pc+g*X*] | | | | If optional condition is true, then jsr to a PC relative (PC+g*X*) address. |
| loop_break(.cc) | Iu25 | | OpC | 3 | If optional condition 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 optional condition 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 | lu4 | OpS | 1 | Move an immediate number into the condition codes. |
| setS.creg | creg | lu4 | OpS | 1 | Initialize the Control register (creg) with an immediate scalar. |
| setB.creg | creg | lu4 | 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 | lu22 | OpS | 1 [3] | Initialize the stack pointer. **NOTE** Only upper 20 bits of byte address is in macro instruction, byte address must be word aligned. |
| 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 optional condition is true, then move contents of g*X* to the stack pointer. |
| | g*Z* | sp | | | If optional condition is true, then move the stack pointer to g*Z*. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| stS | [sp+Is10] | g*X* | OpS | 1 [2,3] | Push g*X* onto the stack at an immediate offset from the current stack pointer.<br>**NOTE**<br>The stack pointer does not change. |
| stS | [sp]+Is10 | g*X* | | | Push g*X* onto the stack, then post-increment the stack pointer by an immediate number. |
| st | [sp+Is10] | H | | 2[2,3] | Push H onto the stack at an immediate offset from the current stack pointer.<br>**NOTE**<br>The stack pointer does not change. |
| st | [sp]+Is10 | H | | | Push H onto the stack, then post-increment the stack pointer by an immediate number. |
| ldS | g*X* | [sp+Is10] | OpS | 4 [3] | Pop g*X* using an immediate offset from the current stack pointer.<br>**NOTE**<br>The stack pointer is not updated with the new address. |
| ldS.u | g*X* | [sp+Is10] | | | Pop g*X* using an immediate offset from the current stack pointer.<br>The stack pointer is updated with the new address. |
| ld | H | [sp+Is10] | | | Pop H using an immediate offset from the current stack pointer.<br>**NOTE**<br>The stack pointer is not updated with the new address. |
| ld.u | H | [sp+Is10] | | | Pop H using an immediate offset from the current stack pointer.<br>The stack pointer is updated with the new address. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| 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.<br>**NOTE**<br>The stack pointer does not change. |
| 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.<br>**NOTE**<br>The stack pointer does not change. |
| stm | [sp+*l*] | r*X* | OpS | 1 | Store an ordered list of RAG registers on the stack at an immediate offset from the current stack pointer. Each RAG stored uses 3 words of memory. All registers (increment, pointer, wrap2, wrap1, start2 & start1) of a RAG are stored.<br>**NOTE**<br>The stack pointer does not change. |
| ldm | r*X* | [sp+*l*] | | 4 | Load an ordered list of RAG registers from the stack at an immediate offset from the current stack pointer. Each RAG is loaded from 3 words of memory. All registers (increment, pointer, wrap2, wrap1, start2 & start1) of a RAG are loaded.<br>**NOTE**<br>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.<br>The stack pointer is post-modified such that it points to the start of the next available (empty) full DMEM line. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| 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.<br>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 | I32 | | OpD | 1 | Store an immediate scalar to the stack. |
| Internal Peripheral Control instructions | | | | | VSPA instructions quick reference |
| mvip | Iu9 | I32 | OpD | 1 | Move an immediate scalar to an IP register.<br>• 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 | Iu9 | g*X*,I32 | | | Move a subset of a scalar register to an IP register.<br>• An immediate integer bit mask is used to specify the source data in g*X*.<br>• 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.<br>• An immediate integer bit mask is used to specify the source data in the IP register. |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | • 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.<br>• An integer bit mask in g*Y* is used to specify the source data in g*X*<br>• 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 | Iu9 | g*X* | | | Move a scalar register to an IP register.<br>• 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.<br>• An integer bit mask in g*Y* is used to specify the source data in the IP register.<br>• 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 | g*X* | Iu9 | | 1 | Move an IP register to a scalar register.<br>• 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 | [g*Z*] | g*X*(,g*Y*) | | 1 | Move into IP register given by index in g*Z*, g*X* is data, optional g*Y* is mask (if not provided use implicit mask of 0xffffffff). |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| 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 g*Y* 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 g*X*[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 g*X*[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 g*X* 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 g*X* mask |
| btstip | Iu9 | Iu5 | OpS | 1 | Test bit at position Iu5 in IP register<br>**NOTE**<br>This instruction cannot be used on a slow IP register. |
| btstip | Iu9 | g*X* | | 1 | Test bit at position g*X*[0:5] in IP register<br>**NOTE**<br>This instruction cannot be used on a slow IP register. |
| Operand Generator instructions | | | | | VSPA instructions quick reference |
| set.nco | {radix2, singles, normal}, Iu10, Is32 | | OpD | 3 | Initialization of the vector NCO generator mode and frequency parameters. Order of arguments is nco_k, nco_freq.<br>NCO frequency in radix 2 mode needs to be configured as a signed 32 bit number. That is, the frequency specified should be in the range [($2^{31}$-1), $2^{31}$-1]. The case of frequency being equal to -$2^{31}$ will be equivalent to frequency being equal to 0. |
| mv | nco_k | Iu11 | OpB | 3 | Set the vector NCO generator (nco_k-parameter) to an immediate scalar. Note, |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | only affects least significant 11 bits of the half word register. |
| add | nco_k | ls11 | | | Offset the vector NCO generator (nco_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*. |
| mv | cgu_reg | g*X* | OpB | 1 | Move a parameter from a scalar register to a cgu register.<br>**NOTE**<br>Restrictions exist on the destination register. See Table 66. |
| mv | g*X* | cgu_reg | | 1 | Move a cgu parameter into a scalar register.<br>**NOTE**<br>Restrictions apply on the source register See Table 66. |
| set.cgu | cgu_mode \| 0x200 \| 0x400 | | | 1 | CGU output can be modified with conjugate and/or ovsf subunit by adding 0x200 and/or 0x400 to the cgu_mode. These values should be defined as macros so that the code would look as follows.<br>`.set conj, 0x200` |

*Table continues on the next page...*

**Table 15. VCPU instructions summary (continued)**

| OpCode | Dest | Source | Family | Cycles | Description |
|---|---|---|---|---|---|
| | | | | | `.set conj, 0x400`<br>`set.cgu cgu_mode \| ovsf \| conj;`<br>`set.cgu cgu_mode \| ovsf;`<br>`set.cgu cgu_mode \| conj;` |
| set.cgu | cgu_mode | | | 1 | Configure the code generator unit operating mode. |
| 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.

### Instructions by family

**Table 16. OpA instructions**

| OpA | | | OpA | | |
|---|---|---|---|---|---|
| nop | | | set.br | ag*X* | fft_size |
| addA(.laddr) | a*U* | ls9 | setA.VRAptr | r*X* | lu9 |
| ldA(.laddr) | [ag*X*]+ls9 | | mvA.VRAptr | r*X* | ag*Y* |
| stA(.laddr) | [ag*X*]+ls9 | | mvA.VRAptr | ag*Y* | r*X* |
| stA(.laddr).w | [ag*X*]+ls9 | | setA.VRAincr | r*X* | ls9 |
| ld(.br) | [ag*X*]+/-ag*Y* | | mvA.VRAincr | r*X* | ag*Y* |
| st | [ag*X*]+/-ag*Y* | | mvA.VRAincr | ag*Y* | r*X* |

*Table continues on the next page...*

**Table 16. OpA instructions (continued)**

| OpA | | | OpA | | |
|---|---|---|---|---|---|
| st.uline | [ag*X*] | | mvA.VRArange1 | r*X* | ag*Y* |
| st.br | [ag*X*]+/-ag*Y* | | mvA.VRArange2 | r*X* | ag*Y* |
| st.llr_mode | [ag*X*]+/-ag*Y* | | mvA.VRArange1 | ag*Y* | r*X* |
| st.w(.br) | [ag*X*]+/-ag*Y* | | mvA.VRArange2 | ag*Y* | r*X* |
| add | a*U* | a*V*, a*W* | stld | [a0]+/-ag*X*, [a1]+/-ag*Y* | |
| sub | a*U* | a*V*, a*W* | stld.laddr | [a0]+/-a*Z*, [a1]+ls9 | |
| st.uline.llr8_mode | [ag*X*] | | stld.laddr | [a0]+ls9, [a1]+/-a*Z* | |

**Table 17. 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] |
| 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* |
| setB.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 | | | | | |

*Table continues on the next page...*

**Table 17. OpB instructions (continued)**

| OpB | | | OpB | | |
|---|---|---|---|---|---|
| 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* |
| mvB.VRAincr | ag*Y* | r*X* | mvB.VRArange2 | ag*Y* | r*X* |
| mv.h | [rV] | g*X* | | | |
| fill.q | [rV] | g*X* : g*X*+1 : g*X*+2 : g*X*+3 | ldB | [ag*X*]+/-ag*Y* | |
| mv.q | [rV] | g*X* : g*X*+1 : g*X*+2 : g*X*+3 | ldB(.laddr) | [ag*X*]+ls5 | |
| fill.d | [rV] | g*X* : g*X*+1 | set.cgu | cgu_mode \| conj \| ovsf<br>See Table 15 | |
| mv.d | [rV] | g*X* : g*X*+1 | set.cgu | cgu_mode | |
| mv.w | g*X* | [rS0] | mv | cgu_reg | g*X* |
| | | | mv | g*X* | cgu_reg |

**Table 18. OpC instructions**

| OpC | | OpC | | |
|---|---|---|---|---|
| nop | | ldhC | g*Y* | [sp]+ls18 |
| set.loop | lu10, lu16 | ldhC.s | g*Y* | [sp]+ls18 |

*Table continues on the next page...*

**Table 18. OpC instructions (continued)**

| OpC | | | OpC | | |
|---|---|---|---|---|---|
| setC.loop | lu16 | | ldhC | g*Y* | [sp+ls18] |
| setC.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*, lu22 | sthC | [ag*X*+ls18] | g*Y* |
| set.range | a*Z* | ag*X*, g*Y* | sthC.u | [ag*X*+ls18] | g*Y* |
| mv | a*U* | lu22 | sthC | [sp]+ls18 | g*Y* |
| addC | a*V* | a*U*, ls19 | sthC | [sp+ls18] | g*Y* |
| add | a*U* | sp, ls19 | sthC.u | [sp+ls18] | g*Y* |
| cmp | a*U*, lu22 | | sth | [ag*X*]+ls9 | l16 |
| swi(.cc) | lu16 | | stb | [ag*X*]+ls9 | l8 |
| ldC | [ag*X*]+ls18 | | stbC | [ag*X*]+ls18 | g*Y* |
| ldC | g*Y* | [ag*X*]+ls18 | stbC | [ag*X*+ls18] | g*Y* |
| ldC | g*Y* | [ag*X*+ls18] | stbC.u | [ag*X*+ls18] | g*Y* |
| ldC | ag*Y* | [sp+ls18] | st.low | [ag*X*]+ag*Y* | lu8 |
| ldC.u | g*Y* | [ag*X*+ls18] | st.low | [ag*X*]-ag*Y* | lu8 |

*Table continues on the next page...*

**Table 18. OpC instructions (continued)**

| OpC | | | OpC | | |
|---|---|---|---|---|---|
| ldC.u | ag*Y* | [sp+ls18] | st.low | [ag*X*]+ls16 | lu8 |
| ldbC | g*Y* | [ag*X*]+ls18 | st.low | [ag*X*]+ag*Y* | g*Z* |
| ldbC.s | g*Y* | [ag*X*]+ls18 | st.low | [ag*X*]-ag*Y* | g*Z* |
| ldbC | g*Y* | [ag*X*+ls18] | st.low | [ag*X*]+ls16 | g*Z* |
| ldbC.s | g*Y* | [ag*X*+ls18] | st.high | [ag*X*]+ag*Y* | lu8 |
| ldbC.u | g*Y* | [ag*X*+ls18] | st.high | [ag*X*]-ag*Y* | lu8 |
| ldbC.u.s | g*Y* | [ag*X*+ls18] | st.high | [ag*X*]+ls16 | lu8 |
| ldhC | g*Y* | [ag*X*]+ls18 | st.high | [ag*X*]+ag*Y* | g*Z* |
| ldhC.s | g*Y* | [ag*X*]+ls18 | st.high | [ag*X*]-ag*Y* | g*Z* |
| ldhC | g*Y* | [ag*X*+ls18] | st.high | [ag*X*]+ls16 | g*Z* |
| ldhC.s | g*Y* | [ag*X*+ls18] | | | |
| ldhC.u | g*Y* | [ag*X*+ls18] | | | |
| ldhC.u.s | g*Y* | [ag*X*+ls18] | | | |

**Table 19. OpD instructions**

| OpD | | | OpD | | |
|---|---|---|---|---|---|
| done | | | | | |
| set.VRAptr | rS0iu9, rS1iu9, rS2iu9, rViu9, rStIu3 | | lfsr | g*X* | g*Y*, I32 |
| set.VRAincr | rS0is9, rS1is9, rS2is9, rVis9, rStIs3 | | xorD(.cc) | g*X* | g*Y*, I32 |
| set.nco | {radix2, singles, normal}, lu16, ls32 | | mpyD(.cc) | g*X* | g*Y*, I32 |
| mv.h | [rV] | ls16 | | | |
| mv.w | [rV], | REALis16, IMAGis16 | | | |
| st | [ag*X*]+ls15 | REALis16, IMAGis16 | mvip | lu9 | I32 |
| push | I32 | | clrip | lu9 | I32 |

*Table continues on the next page...*

**Table 19. OpD instructions (continued)**

| OpD | | | OpD | | |
|---|---|---|---|---|---|
| mvD(.cc) | g*X* | I32 | setip | Iu9 | I32 |
| addD(.ucc)(.cc) | g*X* | g*Y*, I32 | mvip | Iu9 | g*X*, I32 |
| | | | mvip | g*X* | Iu9, I32 |
| subD(.ucc)(.cc) | g*X* | g*Y*, I32 | ldm | a*X*, a*Y*, …, a*Z*, g*X*, g*Y*, ..., g*Z* | [sp+Is16] |
| | | | stm | [sp+Is16] | a*X*, a*Y*, …, a*Z*, g*X*, g*Y*, ..., g*Z* |
| andD(.cc) | g*X* | g*Y*, I32 | stb | Iu22 | I8 |
| orD(.cc) | g*X* | g*Y*, I32 | sth | Iu22 | I16 |
| floatx2n | g*X* | g*Y*, I32 | stw | Iu22 | I32 |
| cmpD(.cc) | g*Y*, I32 | | stb | Iu22 | g*X* |
| ldb | g*X* | Iu22 | sth | Iu22 | g*X* |
| ldh | g*X* | Iu22 | stw | Iu22 | g*X* |
| ldw | g*X* | Iu22 | | | |

**Table 20. OpS instructions**

| OpS | | | OpS | | |
|---|---|---|---|---|---|
| setS.creg | creg | Iu4 | exp(.cc) | g*Z* | g*X* |
| mvip | Iu9 | g*X*, g*Y* | or(.cc) | g*Z* | g*X*, g*Y* |
| mvip | Iu9 | g*X* | sexth(.cc) | g*Z* | g*X* |
| mvip | [g*Z*] | g*X*, g*Y* | sextb(.cc) | g*Z* | g*X* |
| mvip | [g*Z*] | g*X* | bset(.cc) | g*Z* | g*X*,Iu5 |
| mvip | g*X* | Iu9, g*Y* | xor(.cc) | g*Z* | g*X*, g*Y* |
| mvip | g*X* | Iu9 | bxor(.cc) | g*Z* | g*X*, Iu5 |
| mvip | g*X* | [g*Z*], g*Y* | | | |
| mvip | g*X* | [g*Z*] | bclr(.cc) | g*Z* | g*X*, g*Y* |
| bclrip | Iu9 | Iu5 | zexth(.cc) | g*Z* | g*X* |

*Table continues on the next page...*

**Table 20. OpS instructions (continued)**

| OpS | | | OpS | | |
|---|---|---|---|---|---|
| bclrip | Iu9 | g*X* | zextb(.cc) | g*Z* | g*X* |
| clrip | Iu9 | g*X* | bclr(.cc) | g*Z* | g*X*,Iu5 |
| bsetip | Iu9 | Iu5 | | | |
| bsetip | Iu9 | g*X* | mvh.s(.cc) | g*Z* | g*X* |
| setip | Iu9 | g*X* | mv(.cc) | g*Z* | g*X* |
| sr | g*X* | g*Y*,*I* | exg(.cc) | g*Z* | g*X* |
| sr.s | g*X* | g*Y*,*I* | mv(.cc) | g*Z* | pc |
| sign | g*X* | g*Y* | mv(.cc) | g*Z* | quot |
| rotr | g*X* | g*Y*, *I* | mv(.cc) | g*Z* | rem |
| sl | g*X* | g*Y*, *I* | mv | h | VPx(Iu1) |
| rotl | g*X* | g*Y*, *I* | mv | VPx(Iu1) | h |
| btst | g*Y*, *I* | | min | g*Z* | g*X*, g*Y* |
| btstip | Iu9 | Iu5 | max | g*Z* | g*X*, g*Y* |
| btstip | Iu9 | g*X* | | | |
| ldS | g*X* | [sp+Is] | | | |
| ldS.u | g*X* | [sp+Is] | not(.cc) | g*Z* | g*X* |
| stS | [sp+Is] | g*X* | com | VPmask(Iu4) | |
| stS | [sp]+Is | g*X* | abs(.cc) | g*Z* | g*X* |
| ld | H | [sp+Is] | log | g*Z* | g*X* |
| ld.u | H | [sp+Is] | clr.VRA | g*X*, g*Y* | |
| st | [sp+Is] | H | ff1 | g*Z* | g*X* |
| st | [sp]+Is | H | fns | g*Z* | g*X* |
| ldS(.laddr) | g*X* | [ag*Y*+Is] | cntl | g*Z* | g*X* |
| ldS(.laddr).u | g*X* | [ag*Y*+Is] | sum1 | g*Z* | g*X* |
| ldS(.laddr) | g*X* | [ag*Y*]+Is | cnto | g*Z* | g*X* |

*Table continues on the next page...*

**Table 20. OpS instructions (continued)**

| OpS | | | OpS | | |
|---|---|---|---|---|---|
| stS(.laddr) | [ag*Y*+ls] | g*X* | cntz | g*Z* | g*X* |
| stS(.laddr).u | [ag*Y*+ls] | g*X* | ff0to1 | g*Z* | g*X* |
| stS(.laddr) | [ag*Y*]+ls | g*X* | ff1to0 | g*Z* | g*X* |
| ld | H | [ag*Y*+ls] | mv | sp | lu22 |
| ld.u | H | [ag*Y*+ls] | clr.g | *I* | |
| ld | H | [ag*Y*]+ls | clrm | VPmask(lu4) | |
| ldm | VPmask(lu4) | [ag*Y*-lu2] | sl(.cc) | g*Z* | g*X*, g*Y* |
| st | [ag*Y*+ls] | H | rotl(.cc) | g*Z* | g*X*, g*Y* |
| st.u | [ag*Y*+ls] | H | and(.cc) | g*Z* | g*X*, g*Y* |
| st | [ag*Y*]+ls | H | mant(.cc) | g*Z* | g*X* |
| stm | [ag*Y*]+(lu2) | VPmask(lu4) | ld | H | *I* |
| mvS | ag*X* | ag*Y* | st | *I* | H |
| mvS | a*U* | sp | mv | cc | lu4 |
| mvS | sp | a*V* | mvS.z | g*Z* | lu16 |
| exgS | ag*X* | ag*Y* | mvS.s | g*Z* | ls16 |
| exgS | a*U* | sp | cmpS.z | g*Z*, lu16 | |
| nop | | | cmpS.s | g*Z*, ls16 | |
| ld | g*Z* | [ag*X*+/-ag*Y*] | div.z | (g*Z*) | g*Z*, lu16 |
| ld.u | g*Z* | [ag*X*+/-ag*Y*] | div.s | (g*Z*) | g*Z*, ls16 |
| ld | g*Z* | [ag*X*]+/-ag*Y* | mod.z | (g*Z*) | g*Z*, lu16 |
| st | [ag*X*+/-ag*Y*] | g*Z* | mod.s | (g*Z*) | g*Z*, ls16 |
| st.u | [ag*X*+/-ag*Y*] | g*Z* | mpyS.z | (g*Z*) | g*Z*, lu16 |
| st | [ag*X*]+/-ag*Y* | g*Z* | mpyS.s | (g*Z*) | g*Z*, ls16 |
| ld.x2 | g*Z* | [ag*X*+/-ag*Y*] | rdiv.z | (g*Z*) | g*Z*, lu16 |
| ld.u.x2 | g*Z* | [ag*X*+/-ag*Y*] | rdiv.s | (g*Z*) | g*Z*, ls16 |

*Table continues on the next page...*

**Table 20. OpS instructions (continued)**

| OpS | | | OpS | | |
|---|---|---|---|---|---|
| ld.x2 | g*Z* | [ag*X*]+/-ag*Y* | rmod.z | (g*Z*) | g*Z*, lu16 |
| st.x2 | [ag*X*+/-ag*Y*] | g*Z* | rmod.s | (g*Z*) | g*Z*, ls16 |
| st.u.x2 | [ag*X*+/-ag*Y*] | g*Z* | rsub.z | (g*Z*) | g*Z*, lu16 |
| st.x2 | [ag*X*]+/-ag*Y* | g*Z* | rsub.s | (g*Z*) | g*Z*, ls16 |
| ld | H | [ag*X*+/-ag*Y*] | addS(.ucc).z | (g*Z*) | g*Z*, lu16 |
| ld.u | H | [ag*X*+/-ag*Y*] | addS(.ucc).s | (g*Z*) | g*Z*, ls16 |
| ld | H | [ag*X*]+/-ag*Y* | subS(.ucc).z | (g*Z*) | g*Z*, lu16 |
| st | [ag*X*+/-ag*Y*] | H | subS(.ucc).s | (g*Z*) | g*Z*, ls16 |
| st.u | [ag*X*+/-ag*Y*] | H | ldhS(.laddr)(.s) | g*Z* | [ag*X*+ls] |
| st | [ag*X*]+/-ag*Y* | H | ldhS(.laddr).u(.s) | g*Z* | [ag*X*+ls] |
| popm | ag | | ldhS(.laddr)(.s) | g*Z* | [ag*X*]+ls |
| pushm | ag | | sthS(.laddr) | [ag*X*+ls] | g*Z* |
| popm | a | | sthS(.laddr).u | [ag*X*+ls] | g*Z* |
| pushm | a | | sthS(.laddr) | [ag*X*]+ls | g*Z* |
| ldm | r*X* | [sp+*I*] | ldh(.s) | g*Z* | [ag*X*+/-ag*Y*] |
| stm | [sp+*I*] | r*X* | ldh.u(.s) | g*Z* | [ag*X*+/-ag*Y*] |
| and | H | | ldh(.s) | g*Z* | [ag*X*]+/-ag*Y* |
| or | H | | sth | [ag*X*+/-ag*Y*] | g*Z* |
| and | Vp*Z* | Vp*X*, Vp*Y* | sth.u | [ag*X*+/-ag*Y*] | g*Z* |
| or | Vp*Z* | Vp*X*, Vp*Y* | sth | [ag*X*]+/-ag*Y* | g*Z* |
| xor | Vp*Z* | Vp*X*, Vp*Y* | mac(.cc) | g*Z* | g*X*, g*Y* |
| andS(.z) | (g*X*) | g*X*, I16 | | | |
| orS | (g*X*) | g*X*, ls | | | |
| xorS | (g*X*) | g*X*, ls | | | |
| mpy(.cc)(.s) | g*Z* | g*X*, g*Y* | | | |

*Table continues on the next page...*

**Table 20. OpS instructions (continued)**

| OpS | | | OpS | | |
|---|---|---|---|---|---|
| | | | ldb(.s) | g*Z* | [ag*X*+/-ag*Y*] |
| div(.cc)(.s) | g*Z* | g*X*, g*Y* | stb | [ag*X*+/-ag*Y*] | g*Z* |
| | | | ldbS(.laddr)(.s) | g*Z* | [ag*X*+ls] |
| mod(.cc)(.s) | g*Z* | g*X*, g*Y* | ldbS(.laddr).u(.s) | g*Z* | [ag*X*+ls] |
| sr(.cc)(.s) | g*Z* | g*X*, g*Y* | ldbS(.laddr)(.s) | g*Z* | [ag*X*]+ls |
| sr(.cc)(.s) | g*Z* | g*X*, g*Y* | ldbS(.laddr)(.s) | g*Z* | [ag*X*]+ls |
| rotr(.cc) | g*Z* | g*X*, g*Y* | stbS(.laddr) | [ag*X*+ls] | g*Z* |
| addS(.ucc)(.cc) | g*Z* | g*X*, g*Y* | stbS(.laddr).u | [ag*X*+ls] | g*Z* |
| | | | stbS(.laddr) | [ag*Y*]+ls | g*Z* |
| add(.cc) | g*Z* | g*X*, H | stbS(.laddr) | [ag*X*]+ls | g*Z* |
| add(.cc) | g*Z* | H, H | ldb.u(.s) | g*Z* | [ag*X*+/-ag*Y*] |
| add(.cc) | g*Z* | H, g*Y* | ldb(.s) | g*Z* | [ag*X*]+/-ag*Y* |
| subS(.ucc)(.cc) | g*Z* | g*X*, g*Y* | stb.u | [ag*X*+/-ag*Y*] | g*Z* |
| | | | stb | [ag*X*]+/-ag*Y* | g*Z* |
| cmp(.cc) | g*X*, g*Y* | | | | |

**Table 21. OpV instructions**

| OpVr | | OpVsx | | OpVau/OpVsau | OpVd | OpVrot |
|---|---|---|---|---|---|---|
| ld.normal | R*x* | rd | S0 | rmad(.uvp) (.vpnz\|.vpz) | wr.hlinecplx | rol |
| ld.h2h | R*x* | | S1 | rmad.sau(.uvp) (.vpnz\|.vpz) | 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(.uvp) (.vpnz\|.vpz) | wr.fft2 | |
| ld.l2h | R*x* | | | rmac.sau(.uvp) (.vpnz\|.vpz) | wr.fft3 | |

*Table continues on the next page...*

**Table 21. OpV instructions (continued)**

| OpVr | | OpVsx | | OpVau/OpVsau | 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 | | |
| clr | R*x* | | | mads.sau | | |
| bin2num | R*x* | | | maf(.uvp)<br>(.vpnz\|.vpz) | wr.fftn | |
| ld.qam | R*x*<br>x ∈ {0,2,4,6} | | | mafac | wr.fn | |
| | | | | clr.au | wr.fn1 | |
| ld.el_rev | R*x* | | | dif.sau | | |
| | | | | rcp | | |
| | | | | rrt | | |
| | | | | srt | | |
| | | | | nco | | |
| | | | | padd | | |

**Table 22. OpZ instructions**

| OpZ | OpZ |
|---|---|
| nop | loop_begin |
| rts | break |

## 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 23.

**Table 23. CREG Fields**

| index | Field Description | Width | Reset Value |
|---|---|---|---|
| 0 | Reserved | -- | -- |

*Table continues on the next page...*

**Table 23. CREG Fields (continued)**

| index | Field Description | Width | Reset Value |
|---|---|---|---|
| 1 | Reserved | 1 bit | 0 |
| 2 | Reserved | -- | -- |
| 3 | VAU output mode control (0=normal; 1=sign; 2=binary)<br>Mode 0 - normal: when written back to VRA, it returns normal values in specified precision<br>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)<br>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)<br>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<br>bit 0: initial state bit (Z bit)<br>bit 1: sign capture enable bit (0=no capture; 1=capture)<br>bit 2: auto-increment enable (0=no auto-increment; 1=auto-increment)<br>See Vector sign capture register (H) for details. | 3 bits | 000 |
| 13 | HPfixed scale-by-2 switch<br>bit 0: Vprec control<br>bit 1: S1prec control<br>(0=no scaling; 1=scale-by-2)<br>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<br>bit 0: controls Sx-mux mode<br>bit 1: write enable mask for bit 0 (1=enable)<br>bit 2: controls V-mux mode<br>bit 3: write enable mask for bit 2 (1=enable)<br>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 |

*Table continues on the next page...*

**Table 23. CREG Fields (continued)**

<table>
<tr><th>index</th><th>Field Description</th><th>Width</th><th>Reset Value</th></tr>
<tr><td>16</td><td>Fractional interpolator numerator constant (0 -15). Restriction on numerator: D<N<2D, where N is the numerator constant and D is the denominator constant.</td><td>4 bits</td><td>0000</td></tr>
<tr><td>17</td><td>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.</td><td>4 bits</td><td>0000</td></tr>
<tr><td>18</td><td>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</td><td>4 bits</td><td>0000</td></tr>
<tr><td>19</td><td>SP VAU output width switch<br>(0=Single line; 1=Two lines)</td><td>1 bit</td><td>0</td></tr>
<tr><td>20</td><td>SP VAU output lane switch<br>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.<br>Note: only applicable if creg(19)=0<br>Vprec=SP: 0,2=0: 1023; 1,3=1024: 2047</td><td>2 bits</td><td>00</td></tr>
<tr><td>21</td><td>Reserved</td><td>2 bits</td><td>00</td></tr>
<tr><td>22</td><td>vspa_mode<br><table><tr><th>creg22[1:0]</th><th>VSPA Operating mode</th></tr><tr><td>0b00</td><td>VSPA2 mode, supports half word (16-bit) memory access resolution</td></tr><tr><td>0b01</td><td>Reserved, not supported</td></tr><tr><td>0b10</td><td>VPSA3 mode, supports byte (8-bit) memory access resolution</td></tr><tr><td>0b11</td><td>Reserved, not supported</td></tr></table></td><td>2 bits</td><td>10</td></tr>
<tr><td>23</td><td>order_g<br>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).</td><td>3 bits</td><td>000</td></tr>
<tr><td>24</td><td>order_i<br>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).</td><td>3 bits</td><td>000</td></tr>
<tr><td>25</td><td>sau_mode<br>Controls what SAU instructions are available:</td><td>2 bits</td><td>00</td></tr>
</table>

*Table continues on the next page...*

**Table 23. CREG Fields (continued)**

<table>
<tr><th>index</th><th>Field Description</th><th>Width</th><th>Reset Value</th></tr>
<tr><td></td><td>
<table>
<tr><th>creg25[1:0]</th><th>available SAU instructions</th></tr>
<tr><td>0b00</td><td>srt, rrt, rcp, nco, padd</td></tr>
<tr><td>0b01</td><td>Reserved</td></tr>
<tr><td>0b10</td><td>Reserved</td></tr>
<tr><td>0b11</td><td>Reserved</td></tr>
</table>
There must be 1 instruction between changing creg25 and an SAU instruction using the new mode.
</td><td></td><td></td></tr>
<tr><td>26</td><td>rotate3_en<br>Controls what rotate instructions are available:
<table>
<tr><th>creg26[1:0]</th><th>Three register rotate right</th><th>Three register rotate left</th></tr>
<tr><td>0b00</td><td>Not available</td><td>Not available</td></tr>
<tr><td>0b01</td><td>Available</td><td>Not available</td></tr>
<tr><td>0b10</td><td>Not available</td><td>Available</td></tr>
<tr><td>0b11</td><td>Available</td><td>Available</td></tr>
</table>
</td><td>2 bits</td><td>00</td></tr>
<tr><td>27</td><td>elRev<br>Selects element size for VCPU instruction ld.el_rev and IPPU ld_mode er (Element Reverse) & brmr (Bit Reverse Multiple Registers)
<table>
<tr><th>creg27[1:0]</th><th>Load vector register with element reversal</th></tr>
<tr><td>0b00</td><td>Reverses elements of 16 bits</td></tr>
<tr><td>0b01</td><td>Reverses elements of 32 bits</td></tr>
<tr><td>0b10</td><td>Reverses elements of 64 bits</td></tr>
<tr><td>0b11</td><td>Undefined</td></tr>
</table>
</td><td>2 bits</td><td>00</td></tr>
<tr><td>28</td><td>signOpS0<br>S0 sign operation<br>signOpS0[3:2]<br>• 0b00 No sign operation<br>• 0b01 Sign operation on Real element</td><td>4 bits</td><td>0000</td></tr>
</table>

*Table continues on the next page...*

**Table 23. CREG Fields (continued)**

| index | Field Description | Width | Reset Value |
|---|---|---|---|
| | • 0b10 Sign operation on Imag element<br>• 0b11 Sign operation on Both element<br>signOpS0[1:0]<br>• 0b00 Absolute value<br>• 0b01 Negative value<br>• 0b10 Change sign<br>• 0b11 -1 if element < 0, else +1 | | |
| 29 | signOpS1<br>S1 sign operation<br>signOpS1[3:2]<br>• 0b00 No sign operation<br>• 0b01 Sign operation on Real element<br>• 0b10 Sign operation on Imag element<br>• 0b11 Sign operation on Both element<br>signOpS1[1:0]<br>• 0b00 Absolute value<br>• 0b01 Negative value<br>• 0b10 Change sign<br>• 0b11 -1 if element < 0, else +1 | 4 bits | 0000 |
| 30 | signOpS2<br>S2 sign operation<br>signOpS2[3:2]<br>• 0b00 No sign operation<br>• 0b01 Sign operation on Real element<br>• 0b10 Sign operation on Imag element<br>• 0b11 Sign operation on Both element<br>signOpS2[1:0]<br>• 0b00 Absolute value<br>• 0b01 Negative value<br>• 0b10 Change sign<br>• 0b11 -1 if element < 0, else +1 | 4 bits | 0000 |
| 31-254 | Reserved | | |

*Table continues on the next page...*

**Table 23. CREG Fields (continued)**

<table>
<tr><th>index</th><th>Field Description</th><th>Width</th><th>Reset Value</th></tr>
<tr><td>255</td><td>Used for setting multiple fields in a single instruction based on a 4-bit immediate index value<br>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.
<table>
<tr><th>index</th><th>1</th><th>3</th><th>13</th><th>13</th><th>15</th><th>15</th><th>19</th></tr>
<tr><th>bit#</th><th>0</th><th>1:0</th><th>1</th><th>0</th><th>2</th><th>0</th><th>0</th></tr>
<tr><th>CREG State</th><th>VDPW</th><th>AUOM</th><th>HPFS</th><th>HPFV</th><th>FFTV</th><th>FFTS</th><th>ALLAU</th></tr>
<tr><td>0</td><td>0</td><td>00</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>00</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>2</td><td>0</td><td>00</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>3</td><td>0</td><td>00</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>4</td><td>0</td><td>00</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>5</td><td>0</td><td>00</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>6</td><td>0</td><td>00</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>7</td><td>0</td><td>00</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>8</td><td>0</td><td>00</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>9</td><td>0</td><td>00</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>10</td><td>0</td><td>00</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>11</td><td>0</td><td>00</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>12</td><td>0</td><td>00</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>13</td><td>0</td><td>00</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>14</td><td>0</td><td>00</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>15</td><td>0</td><td>00</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
</table>
VDPW - Vector data path width (wide/narrow)<br>AUOM - AU output mode control (normal, sign, bin)<br>HPFS - Half fixed scale by 2 for S1<br>HPFV - Half fixed scale by 2 for V<br>FFTS - FFT mode control bits for S<br>FFTV - FFT mode control bits for V<br>ALLAU - All AU write back enable (for single mode)</td><td>N/A</td><td>N/A</td></tr>
</table>

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 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 24. 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 24. 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 |
| 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*, Iu22 | + | + | + | + | 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 |

*Table continues on the next page...*

**Table 24. Instructions affecting the CC bits (continued)**

| Instruction | N | Z | V | C | Update description |
|---|---|---|---|---|---|
| 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 |
| 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 24 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 25. Set Range Registers**

| | |
|---|---|
| set.range a0<br>set.range a1<br>set.range a2<br>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 26.

**Table 26. 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 Iu22 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* + Iu22 - 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 27. 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**

The instruction 'mv Rx Ry' also uses the same internal data register as temporary storage during the move operation. This instruction should not be located in either of the 2 instructions immediately following any load which has an implicit destination and it should not be placed between a load requiring an explicit destination and the subsequent ld.Rmode instruction.

**Table 27. 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 ag*Y*. 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.x2 | g*Z* | [ag*X*]+/-ag*Y* | 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. |

*Table continues on the next page...*

**Table 27. Summary of load instructions (continued)**

| Load Instructions | Dest | Source | Description |
|---|---|---|---|
| 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.x2 | g*Z* | [ag*X*+/-ag*Y*] | 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.x2 | 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*. |
| 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*. |
| 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 g*Z*, 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*]+*ls* | 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 g*Z*, otherwise the value is zero extended. |
| ldhS(.laddr)(.s) | g*Z* | [ag*X*+*ls*] | 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 g*Z*, otherwise the value is zero extended. The contents of ag*Y* are not modified by this instruction. |

*Table continues on the next page...*

**Table 27. Summary of load instructions (continued)**

| Load Instructions | Dest | Source | Description |
|---|---|---|---|
| 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 g*Z*. If .s is included, the 16 bit value is sign extended into gZ, otherwise the value is zero extended. |
| ldb(.s) | g*Z* | *I* | Load byte from memory address into gZ. |
| ldbS(.laddr)(.s) | g*Z* | [ag*X*+*I*s] | Load byte from [agX+Is] (pre-increment memory pointer) into gZ, optionally sign extend. |
| ldbS(.laddr).u(.s) | g*Z* | [ag*X*+*I*s] | Load byte from [agX+Is] (pre-increment & update memory pointer) into gZ, optionally sign extend. |
| ldbS(.laddr)(.s) | g*Z* | [ag*X*]+*I*s | Load byte from [agX]+Is (post increment & update memory pointer) into gZ, optionally sign extend. |
| ldb(.s) | g*Z* | [ag*X*+/-ag*Y*] | Load byte from [agX+agY] (pre-increment memory pointer) into gZ, optionally sign extend. |
| ldb.u(.s) | g*Z* | [ag*X*+/-ag*Y*] | Load byte from [agX+agY] (pre-increment & update memory pointer) into gZ, optionally sign extend. |
| ldb(.s) | g*Z* | [ag*X*]+/-ag*Y* | Load byte from [agX]+agY (post increment & update memory pointer) into gZ, optionally sign extend. |
| ldm | VPmask(Iu4) | [ag*X*-*vN*] | Load predicate register(s) from DMEM, with pre-read modification of ag*X* by 0,1,2, or 4 VP register widths, corresponding to *vN* = 0, 1, 2 and 4, respectively. The effective memory address is truncated for alignment to the size that matches the number of predicate registers being loaded. The MSB of VPmask corresponds to VP3; the LSB corresponds to VP0. The supported values of the VPmask field are:<br>• VP0 (VPmask = 0x1)<br>• VP1 (VPmask = 0x2)<br>• VP2 (VPmask = 0x4)<br>• VP3 (VPmask = 0x8)<br>• VP0-VP1 (VPmask = 0x3)<br>• VP2-VP3 (VPmask = 0xC)<br>• VP0-VP3 (VPmask = 0xF) |
| ldC | [ag*X*]+Is18 | | Load a vector from DMEM using a pointer, then post-increment or post-decrement by a signed immediate offset. |

*Table continues on the next page...*

**Table 27. Summary of load instructions (continued)**

| Load Instructions | Dest | Source | Description |
|---|---|---|---|
| ldC | g*Y* | [ag*X*]+ls18 | 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*+ls18] | 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*+ls18] | 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+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. |
| ldbC | g*Y* | [ag*X*]+ls18 | Load byte from [ag*X*]+ls (post increment & update memory pointer) into gZ, zero extend. |
| ldbC | g*Y* | [ag*X*+ls18] | Load byte from [ag*X*+ls] (pre-increment memory pointer) into gZ, zero extend. |
| ldbC.u | g*Y* | [ag*X*+ls18] | Load byte from [ag*X*+ls] (pre-increment & update memory pointer) into gZ, zero extend. |
| ldbC.s | g*Y* | [ag*X*]+ls18 | Load byte from [ag*X*]+ls (post increment & update memory pointer) into gZ, sign extend. |
| ldbC.s | g*Y* | [ag*X*+ls18] | Load byte from [ag*X*+ls] (pre-increment memory pointer) into gZ, sign extend. |
| ldbC.u.s | g*Y* | [ag*X*+ls18] | Load byte from [agX+ls] (pre-increment & update memory pointer) into gZ, sign extend. |
| 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*. |

*Table continues on the next page...*

**Table 27. Summary of load instructions (continued)**

| Load Instructions | Dest | Source | Description |
|---|---|---|---|
| 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 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. |
| ldbC | g*Y* | [sp]+ls18 | Load byte from [sp]+ls (post increment & update memory pointer) into gZ, zero extend. |
| ldbC | g*Y* | [sp+ls18] | Load byte from [sp+ls] (pre-increment memory pointer) into gZ, zero extend. |
| ldbC.u | g*Y* | [sp+ls18] | Load byte from [sp+ls] (pre-increment & update memory pointer) into gZ, zero extend. |
| ldbC.s | g*Y* | [sp]+ls18 | Load byte from [sp]+ls (post increment & update memory pointer) into gZ, sign extend. |

*Table continues on the next page...*

**Table 27. Summary of load instructions (continued)**

| Load Instructions | Dest | Source | Description |
|---|---|---|---|
| ldbC.s | g*Y* | [sp+ls18] | Load byte from [sp+ls] (pre-increment memory pointer) into gZ, sign extend. |
| ldbC.u.s | g*Y* | [sp+ls18] | Load byte from [sp+ls] (pre-increment & update memory pointer) into gZ, sign extend. |
| 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. |
| ldB | [ag*X*]+/-ag*Y* | | Load vector from [ag*X*], post increment/decrement ag*X* by ag*Y* |
| ldB(.laddr) | [ag*X*]+ls5 | | Load vector from [ag*X*], post increment ag*X* by ls |

#### OpB loads

The OpB load instruction (ldB) may not be used in parallel with any OpA instruction (ldA/ld/ld.br) which reads from DMEM. The ldB instruction should not use the same pointer register as a parallel OpA store instruction. In addition, the OpB load does not support MAG ranging. If an OpB load instruction in parallel with an OpA store instructions access the same DMEM bank, then the macro instruction will take 1 extra cycle to complete so both accesses are finished before the VCPU continues.

#### OpA stld instructions

MAG ranging is not supported on A1 for the stld instruction. The OpA stld.laddr instruction is restricted to using a4-a11 as the offset register. If the load and store operation of an OpA ldst instruction access the same DMEM bank, then the macro instruction will take 1 extra cycle to complete so both accesses are finished before the VCPU continues.

### 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 28.

**NOTE**

The instruction 'mv Rx Ry' uses the same data multiplexer as the store instructions during the move operation. This instruction should not be located in the same macro instruction as any store.

**Table 28. 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 |

*Table continues on the next page...*

**Table 28. Data write width for st instructions (continued)**

| Store Instruction | Address Source | Data Source | Width of store to VRA |
|---|---|---|---|
| stA(.laddr).w | [ag*X*]+Is9 | VRA | 32 bits |
| st | [ag*X*]+/-ag*Y* | g*Z* | 32 bits |
| st.x2 | [ag*X*]+/-ag*Y* | g*Z* | 32 bits |
| st | [ag*X*+/-ag*Y*] | g*Z* | 32 bits |
| st.u | [ag*X*+/-ag*Y*] | g*Z* | 32 bits |
| st.x2 | [ag*X*+/-ag*Y*] | g*Z* | 32 bits |
| st.u.x2 | [ag*X*+/-ag*Y*] | g*Z* | 32 bits |
| stS(.laddr) | [ag*Y*]+*Is* | g*X* | 32 bits |
| stS(.laddr) | [ag*Y*+*Is*] | g*X* | 32 bits |
| stS(.laddr).u | [ag*Y*+*Is*] | 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*]+*Is* | g*Z* | 16 bits |
| sthS(.laddr) | [ag*Y*+*Is*] | g*Z* | 16 bits |
| sthS(.laddr).u | [ag*Y*+*Is*] | g*Z* | 16 bits |
| stb | *I* | g*Z* | 8 bits |
| stbS(.laddr) | [ag*X*+*I*s] | g*Z* | 8 bits |
| stbS(.laddr).u | [ag*X*+*I*s] | g*Z* | 8 bits |
| stbS(.laddr) | [ag*X*]+*I*s | g*Z* | 8 bits |
| stb | [ag*X*+/-ag*Y*] | g*Z* | 8 bits |
| stb.u | [ag*X*+/-ag*Y*] | g*Z* | 8 bits |
| stb | [ag*X*]+/-ag*Y* | g*Z* | 8 bits |
| stm | [ag*X*]+(Iu2) | VPmask(Iu4) | 128 bits per VP register |
| st | [ag*X*]+Is15 | Is16,Is16 | 32 bits |

*Table continues on the next page...*

**Table 28. Data write width for st instructions (continued)**

| Store Instruction | Address Source | Data Source | Width of store to VRA |
|---|---|---|---|
| st | *I* | g*Z* | 32 bits |
| sth | *I* | g*Z* | 16 bits |
| st | Iu22 | I32 | 32 bits |
| sth | Iu22 | I16 | 16 bits |
| stC | [ag*X*]+Is18 | VRA | line |
| stC.w | [ag*X*]+Is18 | VRA | 32 bits |
| stC | [ag*X*]+Is18 | g*Y* | 32 bits |
| stC | [ag*X*+Is18] | g*Y* | 32 bits |
| stC.u | [ag*X*+Is18] | g*Y* | 32 bits |
| stC | [sp+Is18] | g*Y* | 32 bits |
| stC.u | [sp+Is18] | g*Y* | 32 bits |
| sthC | [ag*X*]+Is18 | g*Y* | 16 bits |
| sthC | [ag*X*+Is18] | g*Y* | 16 bits |
| sthC.u | [ag*X*+Is18] | g*Y* | 16 bits |
| sthC | [sp]+Is18 | g*Y* | 16 bits |
| sthC | [sp+Is18] | g*Y* | 16 bits |
| sthC.u | [sp+Is18] | g*Y* | 16 bits |
| stbC | [ag*X*]+Is18 | g*Y* | 8 bits |
| stbC | [ag*X*+Is18] | g*Y* | 8 bits |
| stbC.u | [ag*X*+Is18] | g*Y* | 8 bits |
| stbC | [sp]+Is18 | g*Y* | 8 bits |
| stbC | [sp+Is18] | g*Y* | 8 bits |
| stbC.u | [sp+Is18] | g*Y* | 8 bits |
| sth | [ag*X*]+Is9 | I16 | 16 bits |
| stb | [ag*X*]+Is9 | I8 | 8 bits |

*Table continues on the next page...*

**Table 28. Data write width for st instructions (continued)**

| Store Instruction | Address Source | Data Source | Width of store to VRA |
|---|---|---|---|
| st.low | [ag*X*]+ag*Y* | Iu8 | partial line |
| st.low | [ag*X*]-ag*Y* | Iu8 | partial line |
| st.low | [ag*X*]+Is16 | 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*]+Is16 | 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*]+Is16 | 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*]+Is16 | g*Z* | partial line |
| st.uline | ag*X* | VRA | variable |
| st.uline.llr8_mode | ag*X* | VRA | variable |
| stb | Iu22 | I8 | 8 bits |
| sth | Iu22 | I16 | 16 bits |
| stw | Iu22 | I32 | 32 bits |

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

For a description of the *st.uline.llr8_mode* instructions, see st.uline.llr8_mode.

#### 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 29.

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]}.

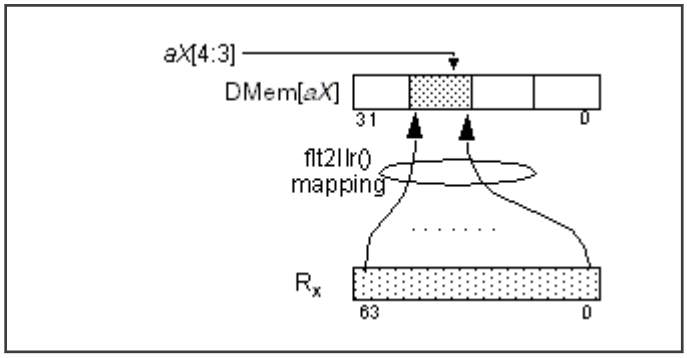
The operations of "st.llr4 [a*X*]+a*Y*" instruction are described in Table 29.

**Table 29. 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]];<br>temp[255:0] = 0;<br>i = 0;<br>foreach real_elem in Rx {<br>temp[255:0] = temp[255:0] \| (flt2llr(real_elem) << (4*i));<br>i=i+1;<br>}<br>memLine[1023:0] = DMem[*aX*];<br>j = *aX*[4:3]; // quarter-line select<br>memLine[256*(j+1)-1:256*j] = temp[255:0];<br>DMem[*aX*] <- memLine[1023:0];<br>*aX* <- *aX* + *aY*;<br>rSt.*ptr* <- rSt.*ptr* + rSt.*incr*; | Read a VRA register, selected using rSt.ptr[2:0], and assign it to Rx.<br>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].<br>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.<br>Post-increment or decrement *aX* by adding or subtracting *aY*.<br>Post-increment or decrement rSt.*ptr* by adding rSt.*incr*.<br>Note: only one-quarter of DMEM line is written. The rest of DMEM line remains unchanged. |

*Table continues on the next page...*

**Table 29. The operations of "st [aXx]+aY, 4" instruction**

| st llr Instruction | Operations | Descriptions |
|---|---|---|
| |  | |

### 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 30.

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]};
```

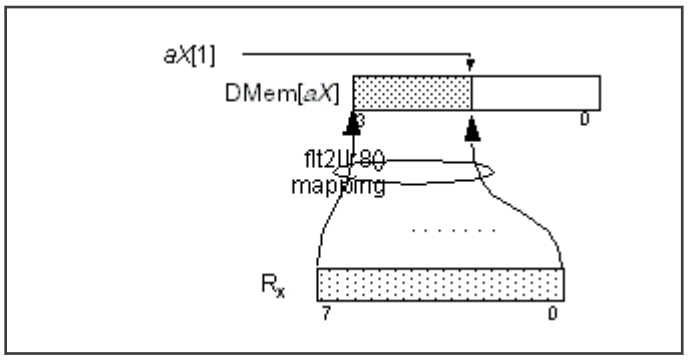
The flt2llr8() function takes a half-word number (x[15:0]) as input and outputs a two's complement 8-bit llr value (y[7:0]). The sign bit of the input will be used as the sign bit of the output.

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 30.

**Table 30. 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]];<br>temp[511:0] = 0;<br>i = 0;<br>foreach real_elem in Rx {<br>temp[511:0] = temp[511:0] \| (flt2llr8(real_elem) << (8*i));<br>i=i+1;<br>}<br>memLine[1023:0] = DMem[*aX*];<br>j = *aX*[1]; // half-line select<br>memLine[512*(j+1)-1:512*j] = temp[511:0];<br>DMem[*aX*] <- memLine[1023:0];<br>*aX* <- *aX* + *aY*;<br>rSt.*ptr* <- rSt.*ptr* + rSt.*incr*;<br>  | Read a VRA register, selected using rSt.ptr[2:0], and assign it to Rx.<br>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].<br>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.<br>Post-increment or decrement *aX* by adding or subtracting *aY*.<br>Post-increment or decrement rSt.*ptr* by adding rSt.*incr*.<br>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.uline.llr8_mode

This instruction takes data from VRA, converts the data into LLR's, and writes up to 1 full line to DMEM. There are two flavors of st.uline.llr8 instructions: "st.uline.llr8" instruction converts VRA data in 16-bit half-fixed format to 8-bit LLR, and the "st.uline.llr8half" instruction converts VRA data in 16-bit half-float format to 8-bit LLR. The conversion is done the exact same way as st.llr8 instruction (See st.llr section).

The source data for st.uline.llr8 is taken from VRA is in special patterns specific to how VSPA would be used to demodulate QAM data. The combination of rSt and QAM mode (See VSPA register descriptions for bit field descriptions of LD_RF_CONTROL register) controls how the source data is taken from VRA for the store operation, see Summary of st.uline.llr8 instructions below.

**Table 31. Summary of st.uline.llr8 instructions**

| QAM Mode | rSt | Source Size | Store Size | Source Vector[1] |
|---|---|---|---|---|
| 16QAM | - | 2 lines | 1 line | { R1[63:62], R0[63:62], … , R1[3:2], R0[3:2], R1[1:0], R0[1:0] } |
| 64QAM | 0 | 2 lines | 1 lines | { R0[43:42], R2[41:40], R1[41:40], R0[41:40], … , R2[3:2], R1[3:2], R0[3:2], R2[1:0], R1[1:0], R0[1:0] } |
| | 1 | 1 line | 1/2 line | { R2[63:62], R1[63:62], R0[63:62], … , R2[45:44], R1[45:44], R0[45:44], R2[43:42], R1[43:42] } |
| 256QAM | 0 | 2 lines | 1 line | { R3[31:30], R2[30:31], R1[30:31], R0[31:30], … , R3[3:2], R2[3:2], R1[3:2], R0[3:2], R3[1:0], R2[1:0], R1[1:0], R0[1:0] } |
| | 1 | 2 lines | 1 line | { R3[63:62], R2[63:62], R1[63:62], R0[63:62], … , R3[35:34], R2[35:34], R1[35:34], R0[35:34], R3[33:32], R2[33:32], R1[33:32], R0[33:32] } |
| 1024QAM | 0 | 2 lines | 1 line | { R3[25:24], R2[25:24], R1[25:24], R0[25:24], R4[23:22], R3[23:22], R2[23:22], R1[23:22], R0[23:22], … , R4[3:2], R3[3:2], R2[3:2], R1[3:2], R0[3:2], R4[1:0], R3[1:0], R2[1:0], R1[1:0], R0[1:0] } |
| | 1 | 2 lines | 1 line | { R2[51:50], R1[51:50], R0[51:50], R4[49:48], R3[49:48], R2[49:48], R1[49:48], R0[49:48], … , R4[27:26], R3[27:26], R2[27:26], R1[27:26], R0[27:26], R4[25:24] } |
| | 2 | 1 line | 1/2 line | { R4[63:62], R3[63:62], R2[63:62], R1[63:62], R0[63:62], … , R4[53:52], R3[53:52], R2[53:52], R1[53:52], R0[53:52], R4[51:50], R3[51:50] } |

1. Rx[a:b] signifies a pair of 16-bit VRA elements = { Rx[a], Rx[b] }

As seen in the table, the source vector differs in sizes and locations according to the QAM mode. It comes from a fixed group of registers in VRA as follows:

- Source of QAM16 is from R0-R1. Takes 1 instruction to write a data batch.
- Source of QAM64 is from R0-R2. Takes 2 instructions to write a data batch.
- Source of QAM256 is from R0-R3. Takes 2 instructions to write a data batch.
- Source of QAM1024 is from R0-R4. Takes 3 instructions to write a data batch.

All data should be ready in the source registers prior to the execution of st.uline.llr8. Notice that 64QAM and 1024QAM will be storing an output batch of 1.5 and 2.5 lines respectively. In order to pack several data batches into DMEM, st.uline.llr8_mode utilizes the native capability of st.uline where the data re-alignment and address advancement are automatically done by

hardware. The DMEM address pointer controls how the data will be stored to DMEM. If the address pointer is aligned with DMEM lines, a whole line can be stored to DMEM. If not, only a partial line will be stored, and the rest will be buffered in the hardware's aligning buffer (inaccessible to software)

All that software needs to do is setting the start address in the address register, and setting a proper transfer size with auto-restart option in ST_UL_VEC_LEN register. QAM64 requires the transfer size set to 1.5 lines and QAM1024 requires it to be 2.5 lines. The other modes can have the transfer size set to one line. See VSPA register descriptions for bit field descriptions of ST_UL_VEC_LEN register.

**Table 32. Recommended settings for st.uline.llr8 instructions**

| QAM Mode | rSt.range | ST_UL_VEC_LEN (for 16AU) |
|---|---|---|
| 16QAM | 0,0 (not actually used) | 64 (1 line) |
| 64QAM | 0,1 | 96 (1.5 lines) |
| 256QAM | 0,1 | 64 (1 line) |
| 1024QAM | 0,2 | 160 (2.5 lines) |

**Use st.uline.llr8 with QAM mode = 1024QAM**

```
.set AUTO_RESTART, 0x10000000
.set Q1024, 10
.set NUM_AU 16
mvip LD_RF_CONTROL, Q1024;
set.VRArange1 rSt, 0, 2;                // Need to be 0-2 for Q1024
set.VRAincr rSt, 1; set.VRAptr rSt, 0;
mv a0, start_address;              // Assume start_address is aligned
mvip ST_UL_VEC_LEN, NUM_AU*10 | AUTO_RESTART;  // 2.5 lines for Q1024

    // software prepares data to be stored in R0 - R4
    st.uline llr8 [a0];    // 1) 1 line is stored
    st.uline llr8 [a0];     // 2) 1 line is stored
    st.uline llr8 [a0];     // 3)  1/2 line is stored
    // software prepares new data to be stored in R0 - R4
    // a0 is now at a half-line boundary
    st.uline llr8 [a0];    // 4) 1/2 line is stored,  1/2 line is buffered.
    st.uline llr8 [a0];     // 5) 1 line is stored, 1/2 line is buffered.
    st.uline llr8 [a0];     // 6) 1 line is stored
    // …
```

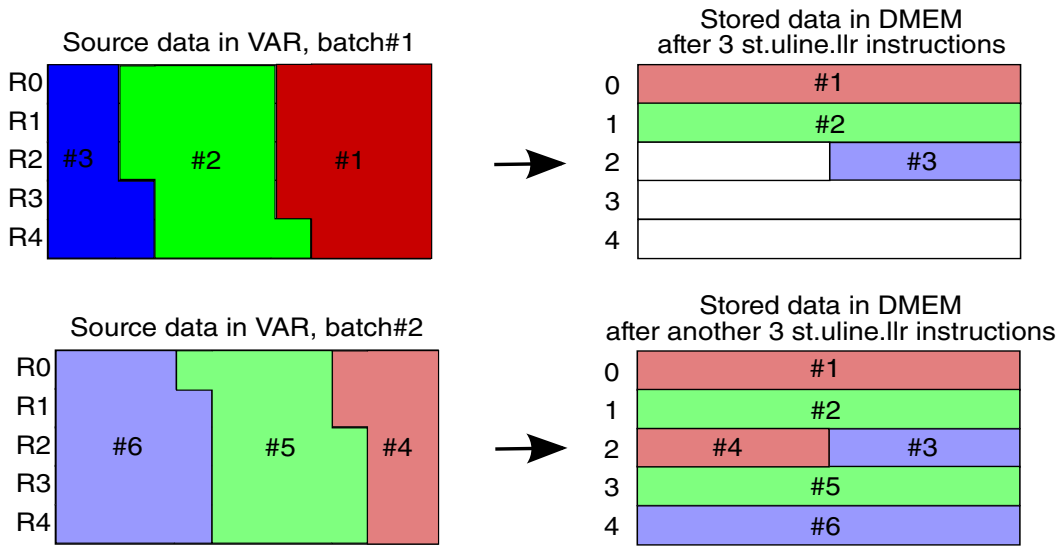



**Figure 15. Use st.uline.llr8 with QAM mode = 1024QAM**

#### 4.8.3.2.4 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.

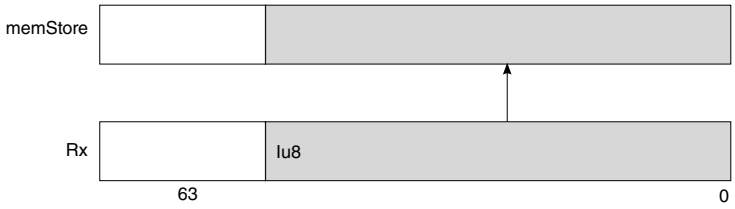



**Figure 16. st.low**

**Table 33. Summary of st.low instructions**

| st.low Instructions | Dest | Source | Description |
|---|---|---|---|
| st.low | [ag*X*]+/-ag*Y* | lu8 | 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 lu8. |
| st.low | [ag*X*]+ls16 | lu8 | 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 lu8. |
| 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. |

*Table continues on the next page...*

**Table 33. Summary of st.low instructions (continued)**

| st.low Instructions | Dest | Source | Description |
|---|---|---|---|
| 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. |

#### 4.8.3.2.5 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 17. st.high**

**Table 34. 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. |
| st.high | [ag*X*]+Is16 | 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 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 | 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 36. 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]<br>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.<br>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*. |

*Table continues on the next page...*

**Table 36. Load VRA Modes (continued)**

| Instructions | Brief Descriptions |
|---|---|
| ld.qam R*x*<br>x ∈ {0,2,4,6} | For modulation order M, take a M/16 line of memReadbus (64M bits), transform it into 2 full lines, and write it into R*x* and R*x+1*, where x ∈ {0,2,4,6}. The transformation implements QAM modulation. |
| ld.el_rev R*x* | Load memRead bus into R*x* with element reversal. |

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.
- When vspa_mode(CREG[22]) is set to 16-bit addressing, the value should be: aX[log2(NAU)+1:0]
- When vspa mode(CREG[22]) is set to 8-bit mode, the value should be: aX[log2(NAU)+2:1]
- 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 37. VRA ld instructions**

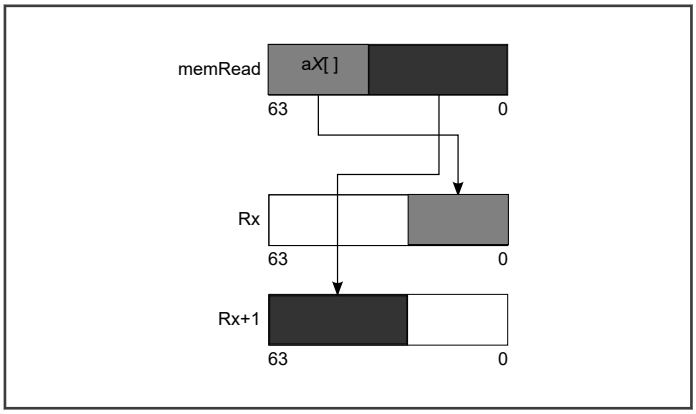
| ld Instruction | Operation | Description |
|---|---|---|
| ld.normal R*x* | R*x*[63:0] <- memRead[63:0]; where R*x* = R0,...,R7 | Load memRead bus into R*x*. |
| ld.h2l_l2h R*x* | R*x*[63- a*X*[] :0] = memRead[63:a*X*[]], where R*x*=R0,R2,R4 or R6.<br>If (a*X*[]!= 0) {<br>R*x*+1[63 : 64-a*X*[]] = memRead[a*X*[]-1 : 0]<br>}<br>where R*x*+1 denotes the next sequential VRA register after R*x*.<br>  | Load high part of memRead bus into low part of R*x*;<br>If a*X*[] is not zero, then load low part of memRead bus into high part of R*x*+1; otherwise, R*x*+1 will not be modified. |

*Table continues on the next page...*

**Table 37. VRA ld instructions (continued)**

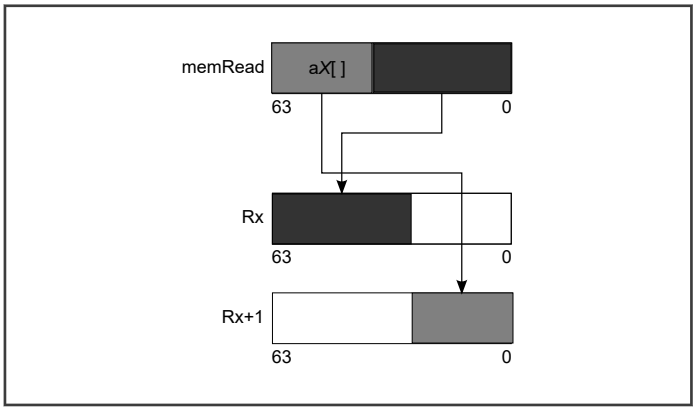
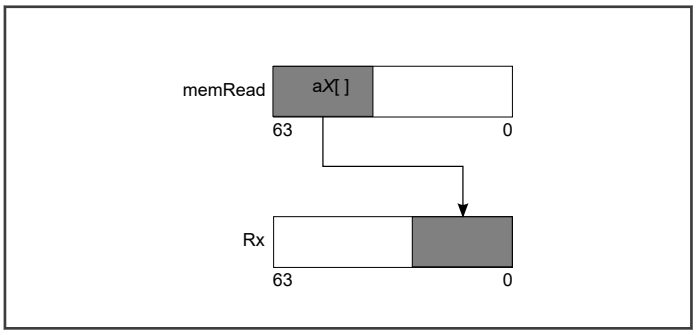
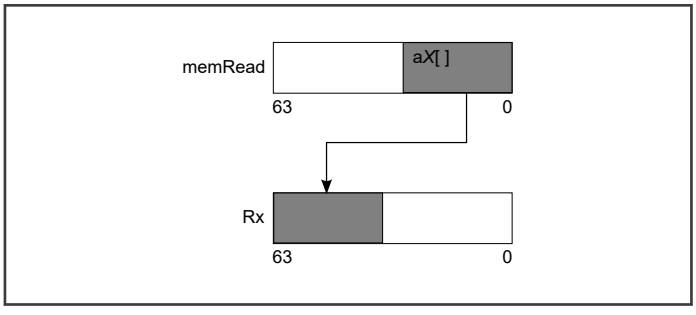
| ld Instruction | Operation | Description |
|---|---|---|
| ld.l2h_h2l R*x* | If (a*X*[] != 0) {<br>R*x*[63:64-a*X*[]] = memRead[a*X*[]-1:0];<br>}<br>where R*x*=R0,R2,R4 or R6.<br>R*x*+1[63- a*X*[] : 0] = memRead[63 : a*X*[]]; where R*x*+1 denotes the next sequential VRA register after R*x*.<br>  | If a*X*[] is not zero, then load low part of memRead bus into high part of R*x*; otherwise, R*x* will not be modified.<br>Load high part of memRead bus into low part of R*x*+1. |
| ld.h2l R*x* | R*x*[63- a*X*[] : 0] = memRead[63 : a*X*[]]; where R*x*=R0,...,R7.<br>  | Load high part of memRead bus into low part of R*x*. |
| ld.l2h R*x* | If (a*X*[] != 0) {<br>R*x*[63 : 64-a*X*[]] = memRead[a*X*[]-1 : 0];<br>}<br>where R*x*=R0,...,R7.<br>  | If a*X*[] is not zero, load low part of memRead bus into high part of R*x*; otherwise R*x* will not be modified. |

*Table continues on the next page...*

**Table 37. VRA ld instructions (continued)**

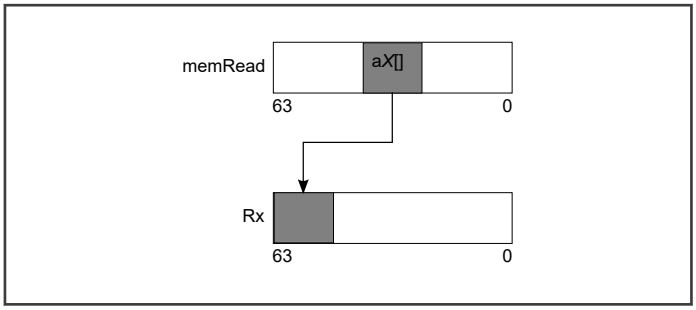
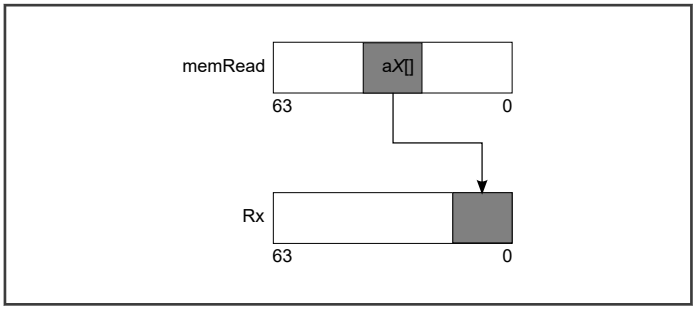
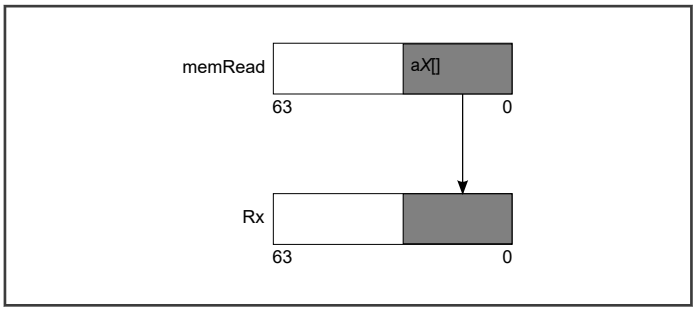
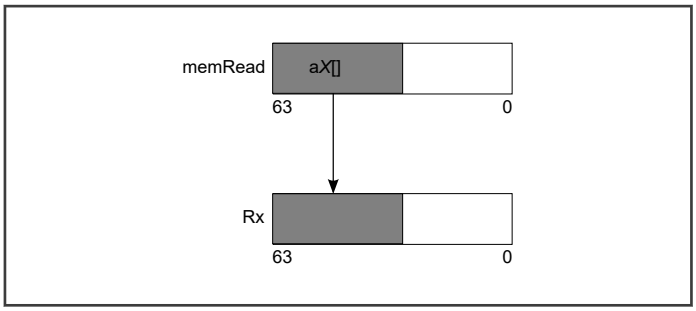
| ld Instruction | Operation | Description |
|---|---|---|
| ld.replace_h R*x* | R*x* = {memRead[a*X*[]], R*x*[62:0]}; where R*x*=R0,...,R7.   | Replace word R*x*[63] with a word on the memRead bus pointed to by a*X*[] |
| ld.replace_l R*x* | R*x* = {R*x*[63:1], memRead[a*X*[]]}; where R*x*=R0,...,R7.   | Replace word R*x*[0] with a word on the memRead bus pointed to by a*X*[] |
| ld.l2l R*x* | R*x*[a*X*[]-1 : 0] = memRead[a*X*[]-1 : 0]; where R*x*=R0,...,R7.   | Replace low part of R*x* by the low part of memRead bus. |
| ld.h2h R*x* | R*x*[63 : a*X*[]] = memRead[63 : a*X*[]]; where R*x*=R0,...,R7.   | Replace high part of R*x* by the high part of memRead bus. |
| ld.qam R*x*<br>x ∈ {0,2,4,6} | Modulation order: M (can be 1, 2, 4, 6, 8 or 10)<br>temp = memRead[a*X* + (M-1): a*X*]; | Extract 2M half-words (or M full-words) from memRead bus, |

*Table continues on the next page...*

**Table 37. VRA ld instructions (continued)**

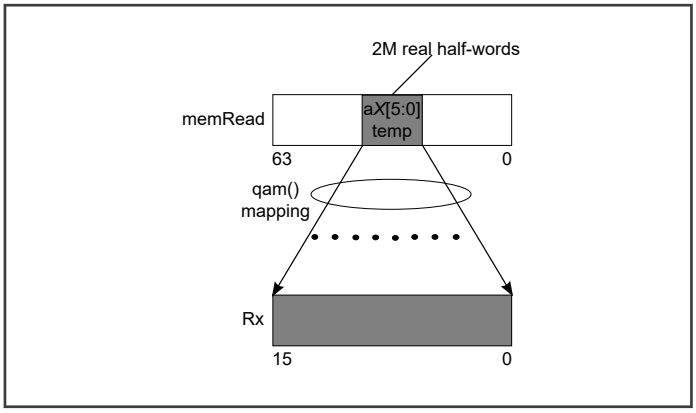
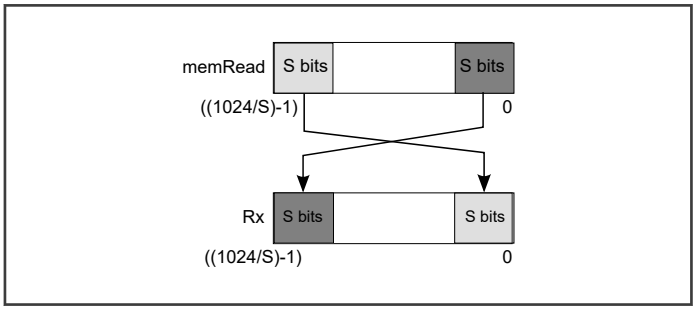
| ld Instruction | Operation | Description |
|---|---|---|
| | for (i=0; i<128; i++) {<br>j = temp & (2M-1);<br>temp = temp >> M;<br>if (i<64) {<br>R*x*[2*i + 0] = real ( qam(j) );<br>R*x*[2*i + 1] = imag ( qam(j) );<br>} else {<br>R*x+1*[2*(i-64) + 0] = real ( qam(j) );<br>R*x+1*[2*(i-64) + 1] = imag ( qam(j) );<br>}<br>}<br>  <br> | starting from the word pointed to by a*X*. Assign this 32M-bit data to temp.<br>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* or R*x+1* (when i>64) starting from element 0.<br>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 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.<br>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.el_rev R*x* | Size S∈{16,32,64}, defined by elRev in CREG index 27 or IP register VCPU_CREG0<br>temp = memRead[a*X*];<br>for (i=0; i<(1024/S); i++) {<br>R*x*[i] = temp[((1024/S)-(i+1))];<br>} | S = 64<br>• Bits 63: 0 of temp load into Rx[1023:960] (64-bit element 0 goes to element 15) |

*Table continues on the next page...*

**Table 37. VRA ld instructions (continued)**

| ld Instruction | Operation | Description |
|---|---|---|
| |  | • Bits 127:64 of temp load into R*x*[ 959:896] (64-bit element 1 goes to element 14)<br>• ...<br>• Bits 1023:960 of temp load into R*x*[63: 0] (64-bit element 15) goes to element 0)<br>S = 32<br>• Bits 31: 0 of temp load into R*x*[1023:992] (32-bit element 0 goes to element 31)<br>• Bits 63:32 of temp load into R*x*[ 991:960] (32-bit element 1 goes to element 30)<br>• ...<br>• Bits 1023:992 of temp load into R*x*[31: 0] (32-bit element 31) goes to element 0)<br>S = 16<br>• Bits 15: 0 of temp load into R*x*[1023:1008] (16-bit element 0 goes to element 63)<br>• Bits 31:16 of temp load into R*x*[ 1007:992] (16-bit element 1 goes to element 62)<br>• ...<br>• Bits 1023:1008 of temp load into R*x*[15: 0] (16-bit element 63) goes to element 0) |

**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 38. RAG Instructions**

| Instruction | Description |
|---|---|
| setA.VRAptr r*X*, Iu9 | Set the 9 bits in ptr register of a RAG register set. |
| setA.VRAptr rS2rS1rS0, Iu3S2, Iu4S1, Iu4S0 | Sets the 4 lsbs of the rS0 & rS1 pointers and sets the 3 lsbs of the rS2 pointer.<br>All remaining bits are cleared. |
| setA.VRAptr rStrVrS1, Iu3St, Iu4V, Iu3S1 | Sets the 3 lsbs of the rSt & rS1 pointers and sets the 4 lsbs of the rV pointer.<br>All remaining bits are cleared. |
| setA.VRAptr rStrV, Iu3St, Iu4V | Sets the 3 lsbs of the rSt pointer and sets the 4 lsbs of the rV pointer.<br>All remaining bits are cleared. |
| setA.VRAptrIP Iu5, Iu4 | Loads the RAG pointers designated by a 5 bit (Iu5) mask from an IP register which is offset from index 0x480 by Iu4. The offset must be even. See VSPA register descriptions for bit field descriptions of RAG_SET registers. |
| 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. |
| setB.VRAptr rS2rS1rS0, Iu3S2, Iu4S1, Iu4S0 | Sets the 4 lsbs of the rS0 & rS1 pointers and sets the 3 lsbs of the rS2 pointer.<br>All remaining bits are cleared. |
| setB.VRAptr rStrVrS1, Iu3St, Iu4V, Iu3S1 | Sets the 3 lsbs of the rSt & rS1 pointers and sets the 4 lsbs of the rV pointer.<br>All remaining bits are cleared. |
| setB.VRAptr rStrV, Iu3St, Iu4V | Sets the 3 lsbs of the rSt pointer and sets the 4 lsbs of the rV pointer.<br>All remaining bits are cleared. |
| setB.VRAptrIP Iu5, Iu4 | Loads the RAG pointers designated by a 5 bit (Iu5) mask from an IP register which is offset from index 0x480 by Iu4. The offset must be even. See VSPA register descriptions for bit field descriptions of RAG_SET registers. |
| 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 |

*Table continues on the next page...*

**Table 38. RAG Instructions (continued)**

| Instruction | Description |
|---|---|
| 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 |
| 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.

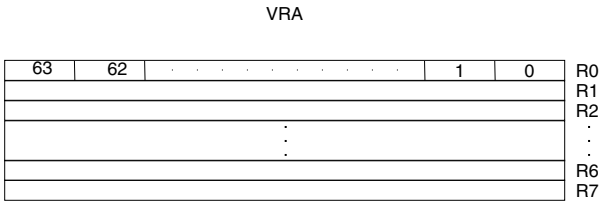



**Figure 18. Data Packing in the VRA (half-word)**

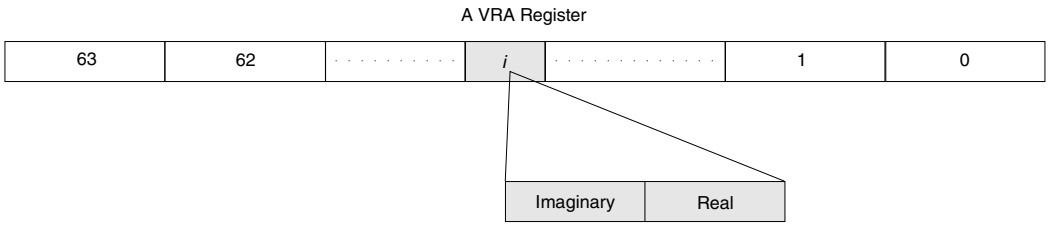



**Figure 19. Data packing in a VRA register line (half-word)**

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

**Table 39. Instruction Precedent for Parallel Instructions**

| First Instruction | Second Instruction | Shared Registers | Higher Precedent |
|---|---|---|---|
| any of:<br>• setA.VRAptr r*X*, Iu9<br>• setB.VRAptr r*X*, Iu9<br>• mv.[h,w] gX, [rS0]<br>• mv.[h,w,q] [rV], gX<br>• fill.[h,w,q] [rV], gX | rd S0 | rS0 | setA/setB |
| | rd S1 | rS1 | |
| | 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 40 shows all rot instructions. In this table, rot_mode denotes a supported rot mode. See Rotate-register modes for all supported rot_mode.

**Table 40. 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 41 shows all possible <ror_mode> for R0 and R1, along with a description for each of these modes.

- Table 42 shows all possible <ror_mode> for R2 and R3, along with a description for each of these modes.
- Note that Table 41 and Table 42 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 41. All Supported <ror_mode> - For R0 and R1**

| rot_mode | Descriptions |
|---|---|
| R0R1r1 | Rotate right R0/R1 by 1 real elements.<br>{ R1[mse:0], R0[mse:0] } = { R0[0], R1[mse:0], R0[mse:1] } |
| R0R1r2 | Rotate right R0/R1 by 2 real elements.<br>{ R1[mse:0], R0[mse:0] } = { R0[1:0], R1[mse:0], R0[mse:2] } |
| R0R1r4 | Rotate right R0/R1 by 4 real elements.<br>{ R1[mse:0], R0[mse:0] } = { R0[3:0], R1[mse:0], R0[mse:4] } |
| R0R1r8 | Rotate right R0/R1 by 8 real elements.<br>{ 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.<br>R0[mse:0] = { R1[0], R0[mse:1] } |
| R0r2 | Shift right R0 by 2 real elements.<br>R0[mse:0] = { R1[1:0], R0[mse:2] } |
| R0r4 | Shift right R0 by 4 real elements.<br>R0[mse:0] = { R1[3:0], R0[mse:4] } |
| R0r8 | Shift right R0 by 8 real elements.<br>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]. |

*Table continues on the next page...*

**Table 41. All Supported <ror_mode> - For R0 and R1 (continued)**

| rot_mode | Descriptions |
|---|---|
| R1r1 | Shift right R1 by 1 real elements.<br>R1[mse:0] = { R0[0], R1[mse:1] } |
| R1r2 | Shift right R1 by 2 real elements.<br>R1[mse:0] = { R0[1:0], R1[mse:2] } |
| R1r4 | Shift right R1 by 4 real elements.<br>R1[mse:0] = { R0[3:0], R1[mse:4] } |
| R1r8 | Shift right R1 by 8 real elements.<br>R1[mse:0] = { R0[7:0], R1[mse:8] } |
| R1rND1 | Fractional shift right R1 by 1 real elements[1]. |
| 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 42. All Supported <ror_mode> - For R2 and R3**

| rot_mode | Descriptions |
|---|---|
| R2R3r1 | Rotate right R2/R3 by 1 real elements.<br>{ R3[mse:0], R2[mse:0] } = { R2[0], R3[mse:0], R2[mse:1] } |
| R2R3r2 | Rotate right R2/R3 by 2 real elements.<br>{ R3[mse:0], R2[mse:0] } = { R2[1:0], R3[mse:0], R2[mse:2] } |
| R2R3r4 | Rotate right R2/R3 by 4 real elements.<br>{ R3[mse:0], R2[mse:0] } = { R2[3:0], R3[mse:0], R2[mse:4] } |
| R2R3r8 | Rotate right R2/R3 by 8 real elements.<br>{ 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.<br>R2[mse:0] = { R3[0], R2[mse:1] } |

*Table continues on the next page...*

**Table 42. All Supported <ror_mode> - For R2 and R3 (continued)**

| rot_mode | Descriptions |
|---|---|
| R2r2 | Shift right R2 by 2 real elements.<br>R2[mse:0] = { R3[1:0], R2[mse:2] } |
| R2r4 | Shift right R2 by 4 real elements.<br>R2[mse:0] = { R3[3:0], R2[mse:4] } |
| R2r8 | Shift right R2 by 8 real elements.<br>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]. |
| R3r1 | Shift right R3 by 1 real elements.<br>R3[mse:0] = { R2[0], R3[mse:1] } |
| R3r2 | Shift right R3 by 2 real elements.<br>R3[mse:0] = { R2[1:0], R3[mse:2] } |
| R3r4 | Shift right R3 by 4 real elements.<br>R3[mse:0] = { R2[3:0], R3[mse:4] } |
| R3r8 | Shift right R3 by 8 real elements.<br>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 Three-register rotate right

**Table 43. Supported <ror_mode> - For three-register rotate**

| ror_mode | Descriptions |
|---|---|
| R0R1R2r1 | Rotate right R0/R1/R2 by 1 real elements<br>{ R2[mse:0], R1[mse:0], R0[mse:0] } = { R0[0], R2[mse:0],R1[mse:0], R0[mse:1] }<br>This mode replaces R0R1r1 when rotate3_en[0] (bit 0 of CREG# 26) = 1 |

*Table continues on the next page...*

**Table 43. Supported <ror_mode> - For three-register rotate (continued)**

| ror_mode | Descriptions |
|---|---|
| R0R1R2r2 | Rotate right R0/R1/R2 by 1 real elements<br>{ R2[mse:0], R1[mse:0], R0[mse:0] } = { R0[1:0], R2[mse:0],R1[mse:0], R0[mse:2] }<br>This mode replaces R0R1r2 when rotate3_en[0] (bit 0 of CREG 26) = 1 |
| R0R1R2r4 | Rotate right R0/R1/R2 by 1 real elements<br>{ R2[mse:0], R1[mse:0], R0[mse:0] } = { R0[3:0], R2[mse:0],R1[mse:0], R0[mse:4] }<br>This mode replaces R0R1r4 when rotate3_en[0] (bit 0 of CREG 26) = 1 |
| R0R1R2r8 | Rotate right R0/R1/R2 by 1 real elements<br>{ R2[mse:0], R1[mse:0], R0[mse:0] } = { R0[7:0], R2[mse:0],R1[mse:0], R0[mse:8] }<br>This mode replaces R0R1r8 when rotate3_en[0] (bit 0 of CREG 26) = 1 |
| R0R1R3r1 | Rotate right R0/R1/R3 by 1 real elements<br>{ R3[mse:0], R1[mse:0], R0[mse:0] } = { R0[0], R3[mse:0],R1[mse:0], R0[mse:1] }<br>This mode replaces R2R3r1 when rotate3_en[0] (bit 0 of CREG# 26) = 1 |
| R0R1R3r2 | Rotate right R0/R1/R3 by 1 real elements<br>{ R3[mse:0], R1[mse:0], R0[mse:0] } = { R0[1:0], R3[mse:0],R1[mse:0], R0[mse:2] }<br>This mode replaces R2R3r2 when rotate3_en[0] (bit 0 of CREG 26) = 1 |
| R0R1R3r4 | Rotate right R0/R1/R3 by 1 real elements<br>{ R3[mse:0], R1[mse:0], R0[mse:0] } = { R0[3:0], R3[mse:0],R1[mse:0], R0[mse:4] }<br>This mode replaces R2R3r4 when rotate3_en[0] (bit 0 of CREG 26) = 1 |
| R0R1R3r8 | Rotate right R0/R1/R3 by 1 real elements<br>{ R3[mse:0], R1[mse:0], R0[mse:0] } = { R0[7:0], R3[mse:0],R1[mse:0], R0[mse:8] }<br>This mode replaces R2R3r8 when rotate3_en[0] (bit 0 of CREG 26) = 1 |

#### 4.10.1.4 Left rotate/shift on R4 and R5

Table 44 shows all possible <rol_mode> for R4 and R5, along with a description for each of these modes.

**Table 44. All Supported <rol_mode> - For R4 and R5**

| rot_mode | Descriptions |
|---|---|
| R4R5l1 | Rotate left R4/R5 by 1 real elements.<br>{ R5[mse:0], R4[mse:0] } = { R5[mse-1:0], R4[mse:0], R5[mse] } |
| R4R5l2 | Rotate left R4/R5 by 2 real elements.<br>{ R5[mse:0], R4[mse:0] } = { R5[mse-2:0], R4[mse:0], R5[mse:mse-1] } |

*Table continues on the next page...*

**Table 44. All Supported <rol_mode> - For R4 and R5 (continued)**

| rot_mode | Descriptions |
|---|---|
| R4R5l4 | Rotate left R4/R5 by 4 real elements.<br>{ 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.<br>{ 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.<br>R4[mse:0] = { R4[mse-1:0], R5[mse] } |
| R4l2 | Rotate left R4 by 2 real elements.<br>R4[mse:0] = { R4[mse-2:0], R5[mse:mse-1] } |
| R4l4 | Rotate left R4 by 4 real elements.<br>R4[mse:0] = { R4[mse-4:0], R5[mse:mse-3] } |
| R4l8 | Rotate left R4 by 8 real elements.<br>R4[mse:0] = { R4[mse-8:0], R5[mse:mse-7] } |
| R5l1 | Rotate left R5 by 1 real elements.<br>R5[mse:0] = { R5[mse:1], R4[mse] } |
| R5l2 | Rotate left R5 by 2 real elements.<br>R5[mse:0] = { R5[mse:2], R4[mse:mse-1] } |
| R5l4 | Rotate left R5 by 4 real elements.<br>R5[mse:0] = { R5[mse:4], R4[mse:mse-3] } |
| R5l8 | Rotate left R5 by 8 real elements.<br>R5[mse:0] = { R5[mse:8], R4[mse:mse-7] } |

#### 4.10.1.5 Left rotate/shift on R6 and R7

Table 45 shows all possible <rol_mode> for R6 and R7, along with a description for each of these modes.

**Table 45. All Supported <rol_mode> - For R6 and R7**

| rot_mode | Descriptions |
|---|---|
| R6R7l1 | Rotate left R6/R7 by 1 real elements.<br>{ R7[mse:0], R6[mse:0] } = { R7[mse-1:0], R6[mse:0], R7[mse] } |
| R6R7l2 | Rotate left R6/R7 by 2 real elements.<br>{ 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. |

*Table continues on the next page...*

**Table 45. All Supported <rol_mode> - For R6 and R7 (continued)**

| rot_mode | Descriptions |
|---|---|
| | { 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.<br>{ 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.<br>R6[mse:0] = { R6[mse-1:0], R7[mse] } |
| R6l2 | Rotate left R6 by 2 real elements.<br>R6[mse:0] = { R6[mse-2:0], R7[mse:mse-1] } |
| R6l4 | Rotate left R6 by 4 real elements.<br>R6[mse:0] = { R6[mse-4:0], R7[mse:mse-3] } |
| R6l8 | Rotate left R6 by 8 real elements.<br>R6[mse:0] = { R6[mse-8:0], R7[mse:mse-7] } |
| R7l1 | Rotate left R7 by 1 real elements.<br>R7[mse:0] = { R7[mse:1], R6[mse] } |
| R7l2 | Rotate left R7 by 2 real elements.<br>R7[mse:0] = { R7[mse:2], R6[mse:mse-1] } |
| R7l4 | Rotate left R7 by 4 real elements.<br>R7[mse:0] = { R7[mse:4], R6[mse:mse-3] } |
| R7l8 | Rotate left R7 by 8 real elements.<br>R7[mse:0] = { R7[mse:8], R6[mse:mse-7] } |

#### 4.10.1.6 Three-register rotate left

**Table 46. Supported <rol_mode> - For three-register rotate**

| rol_mode | Descriptions |
|---|---|
| R4R5R6l1 | Rotate left R4/R5/R6 by 1 real elements<br>{ R6[mse:0], R5[mse:0], R4[mse:0] } = { R6[mse-1:0], R5[mse:0], R4[mse:0], R5[mse] }<br>This mode replaces R4R5l1 when rotate3_en[1] (bit 1 of CREG# 26) = 1 |
| R4R5R6l2 | Rotate left R4/R5/R6 by 1 real elements<br>{ R6[mse:0], R5[mse:0], R4[mse:0] } = { R6[mse-2:0], R5[mse:0], R4[mse:0], R5[mse:mse-1] }<br>This mode replaces R4R5l2 when rotate3_en[1] (bit 1 of CREG 26) = 1 |

*Table continues on the next page...*

**Table 46. Supported <rol_mode> - For three-register rotate (continued)**

| rol_mode | Descriptions |
|---|---|
| R4R5R6I4 | Rotate left R4/R5/R6 by 1 real elements<br>{ R6[mse:0], R5[mse:0], R4[mse:0] } = { R6[mse-4:0], R5[mse:0], R4[mse:0], R5[mse:mse-3] }<br>This mode replaces R4R5I4 when rotate3_en[1] (bit 1 of CREG 26) = 1 |
| R4R5R6I8 | Rotate left R4/R5/R6 by 1 real elements<br>{ R6[mse:0], R5[mse:0], R4[mse:0] } = { R6[mse-8:0], R5[mse:0], R4[mse:0], R5[mse:mse-7] }<br>This mode replaces R4R5I8 when rotate3_en[1] (bit 1 of CREG 26) = 1 |
| R4R5R7I1 | Rotate left R4/R5/R7 by 1 real elements<br>{ R7[mse:0], R5[mse:0], R4[mse:0] } = { R7[mse-1:0], R5[mse:0], R4[mse:0], R5[mse] }<br>This mode replaces R6R7I1 when rotate3_en[1] (bit 1 of CREG# 26) = 1 |
| R4R5R7I2 | Rotate left R4/R5/R7 by 1 real elements<br>{ R7[mse:0], R5[mse:0], R4[mse:0] } = { R7[mse-2:0], R5[mse:0], R4[mse:0], R5[mse:mse-1] }<br>This mode replaces R6R7I2 when rotate3_en[1] (bit 1 of CREG 26) = 1 |
| R4R5R7I4 | Rotate left R4/R5/R7 by 1 real elements<br>{ R7[mse:0], R5[mse:0], R4[mse:0] } = { R7[mse-4:0], R5[mse:0], R4[mse:0], R5[mse:mse-3] }<br>This mode replaces R6R7I4 when rotate3_en[1] (bit 1 of CREG 26) = 1 |
| R4R5R7I8 | Rotate left R4/R5/R7 by 1 real elements<br>{ R7[mse:0], R5[mse:0], R4[mse:0] } = { R7[mse-8:0], R5[mse:0], R4[mse:0], R5[mse:mse-7] }<br>This mode replaces R6R7I8 when rotate3_en[1] (bit 1 of CREG 26) = 1 |

#### 4.10.1.7 Rotate-register modes examples

Following figure shows examples of rotate right by real elements:

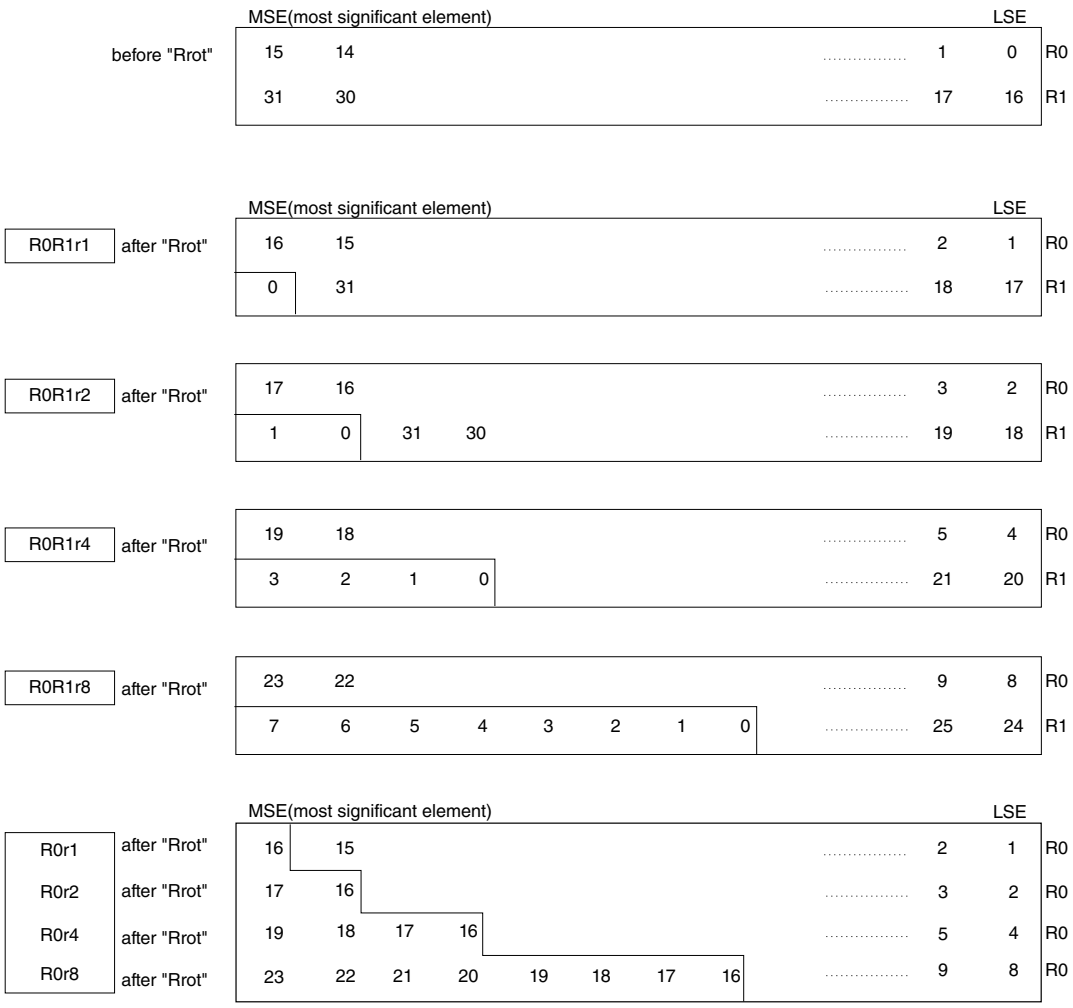



**Figure 20. Rotate right by real elements examples**

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 21.

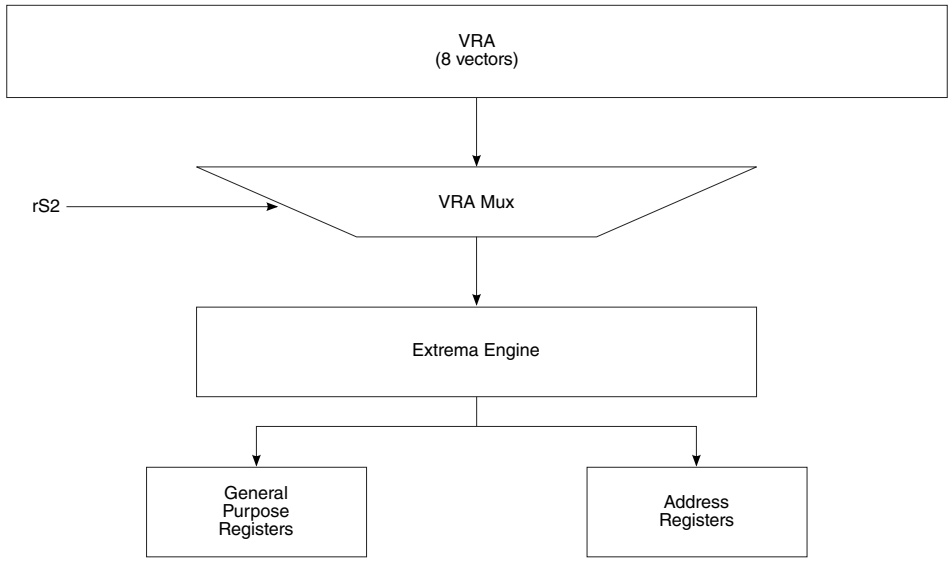



**Figure 21. Extrema Engine Data Path**

**Table 47. 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 22.

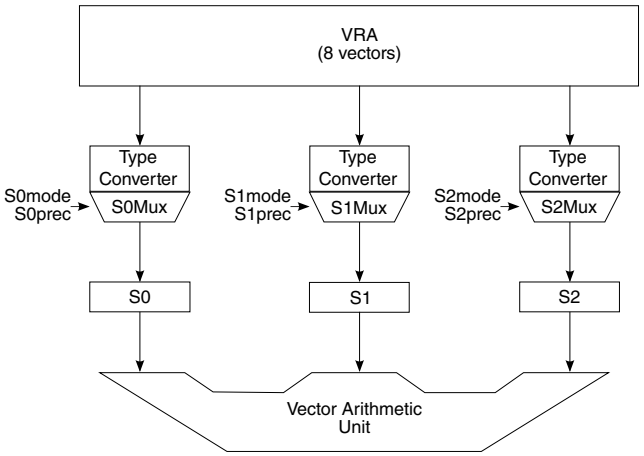



**Figure 22. VRA to Source Register Data Path**

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 48

**Table 48. 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 48. 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 48).

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 48:

- 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 48 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:

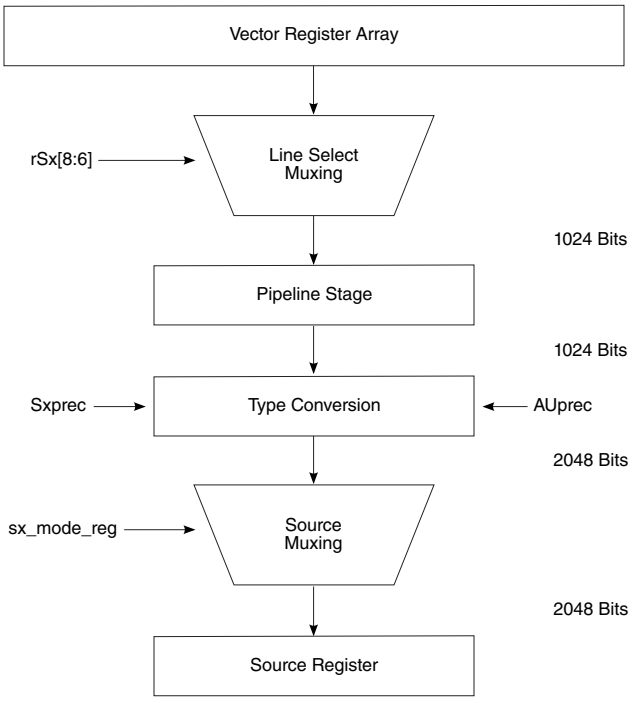



**Figure 23. Half Fixed to Single Precision Source Load**

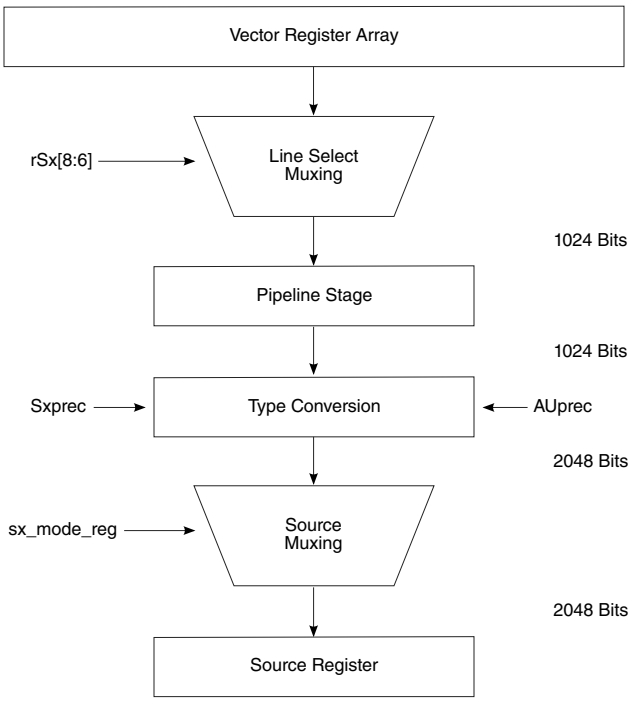



**Figure 24. Half Precision to Single Precision Source Load**

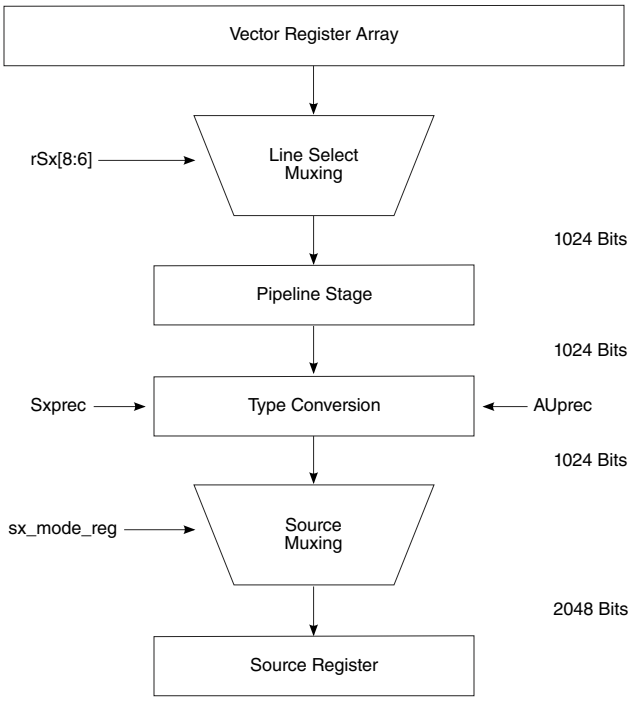



**Figure 25. Single Precision to Single Precision Source Load**

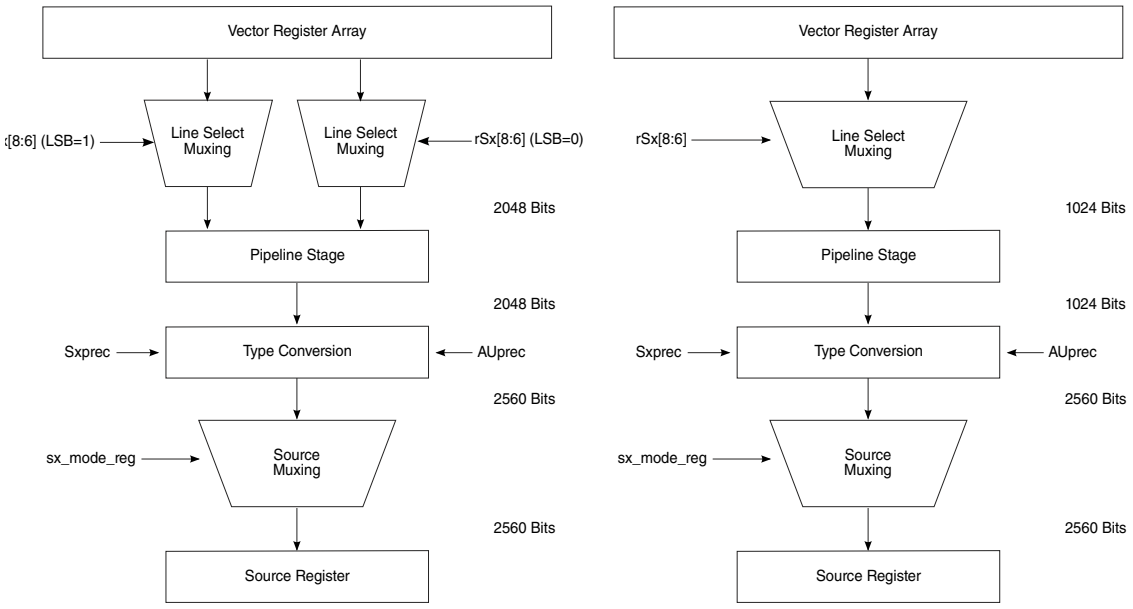



**Figure 26. Half Fixed to F24 Precision Source Load (for real and complex operations respectively)**

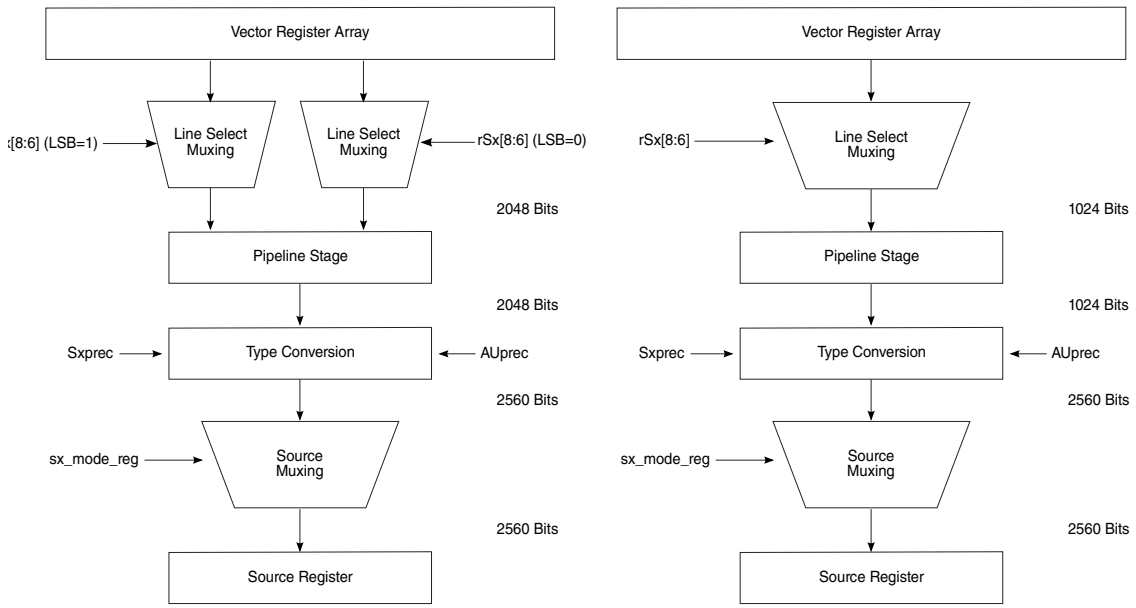



**Figure 27. Half Precision to F24 Precision Source Load (for real and complex operations respectively)**

### 4.12.4 *S0mode* options and detailed description

Table 49 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 or 128 20 bit 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 or 128 20 bit 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 49 for more details.

Table 49 illustrates S0modes in AUprec single. All S0modes are also supported in F24 mode. In F24 precision, the output of the type converter will be 2560 bits instead of 2048 bits. Thus, the half select mux will select 1280 MSB or LSB bits instead of 1024. The size of elements that get duplicated will be scaled by a factor of 20 instead of 32. For example, in S0hlinecplx the size of the element would be 40 instead of 64. The source register Sx (S0 for S0mode) will also be larger.

**Table 49. All Supported *S0mode***

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

*Table continues on the next page...*

**Table 49. All Supported *S0mode* (continued)**

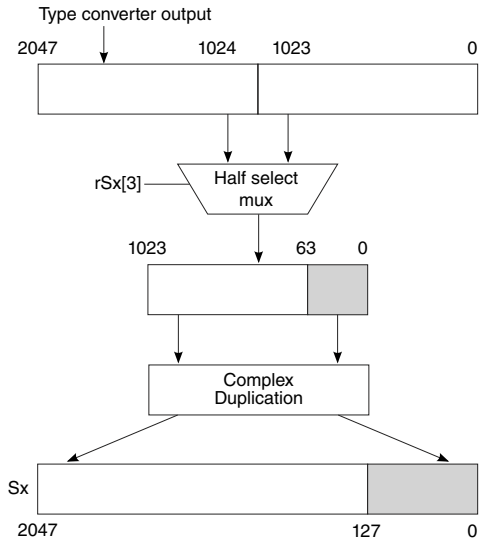
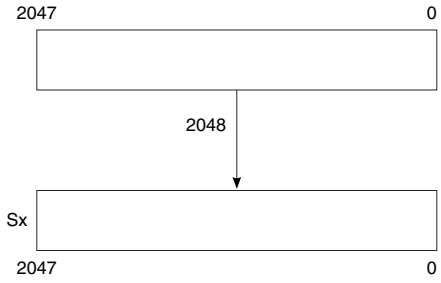
| s0mode | Restrictions | Descriptions[1] |
|---|---|---|
| | |  <br>* Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication. |
| S0straight | Real data<br>creg(15[0])=0, creg(19)=x | Load S0 with output of the type converter.<br>Update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

*Table continues on the next page...*

**Table 49. All Supported *S0mode* (continued)**

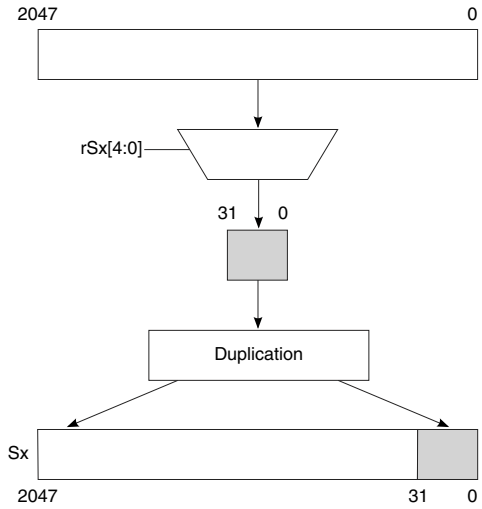
| s0mode | Restrictions | Descriptions[1] |
|---|---|---|
| | |  <br>* Note: If Sxprec is single, the rag bits used are rSx[5:2]. |
| S0group2nr | Real data<br>creg(15[0])=0, creg(23), creg(19)=x | S0group2nr<br>Load a group of real elements from output of the type converter, using rS0 as offset.<br>Duplicate this group into all elements of S0:<br>for (i = 0; i < 4*NAU/n; i++) {<br>for (j = 0; j < n; j++) {<br>S0[n*i+j] = r[j+rS0];<br>}<br>}<br>Number of elements (n) in group is determined by 2^order_g<br>Value of order_g is restricted such that:<br>1<2^order_g<NAU*4<br>Update rS0. |

*Table continues on the next page...*

**Table 49. All Supported *S0mode* (continued)**

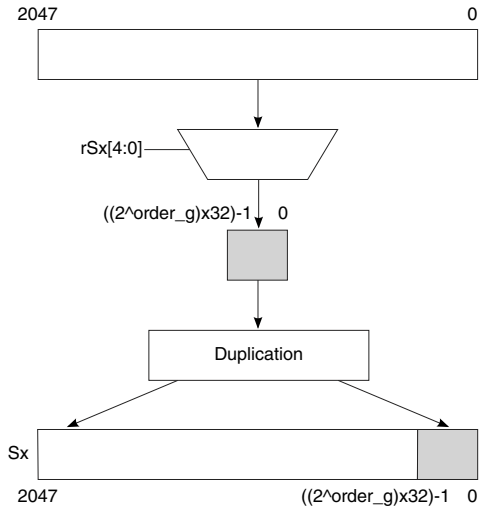
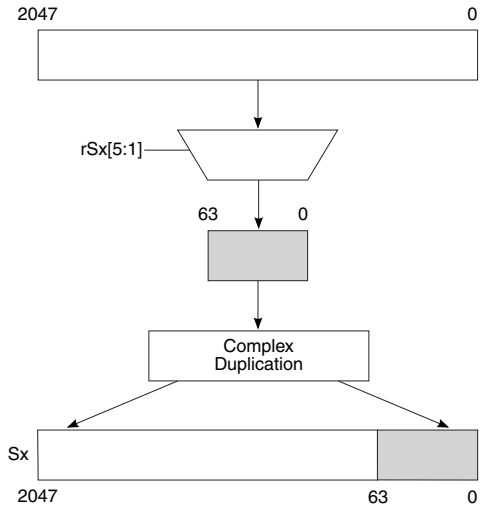
| s0mode | Restrictions | Descriptions[1] |
|---|---|---|
| | |  |
| S0word | Complex data<br>creg(15[0])=0,<br>creg(19)=x | Load a complex element from output of the type converter, using rS0 as offset, in preparation for complex multiplication (cmad or cmac).<br>Each complex element is duplicated with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

*Table continues on the next page...*

**Table 49. All Supported *S0mode* (continued)**

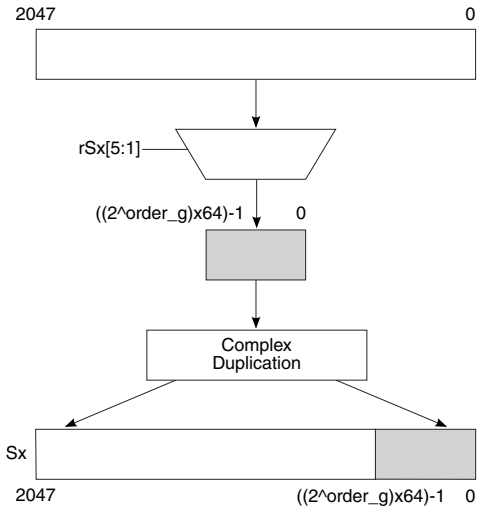
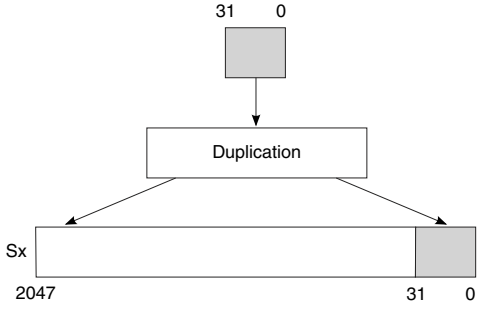
| s0mode | Restrictions | Descriptions[1] |
|---|---|---|
| | | for (j = 0; j < n; j++) {<br>S0[4*n*i+4*j] = r[2*(j+rS0)];<br>S0[4*n*i+4*j+1] = -r[2*(j+rS0)+1];<br>S0[4*n*i+4*j+2] = r[2*(j+rS0)+1];<br>S0[4*n*i+4*j+3] = r[2*(j+rS0)];<br>}<br>}<br>Number of elements (n) in group is determined by 2^order_g<br>Value of order_g is restricted such that:<br>1<2^order_g<NAU<br>Update rS0.<br>  |
| S0zeros | Real and<br>Complex data<br>creg(15[0])=0, creg(19)=x | Load all elements of S0 with floating-point constant "0".<br>Update rS0.<br>  |
| S0real1 | Real data | Load all elements of S0 with floating-point constant "1". |

*Table continues on the next page...*

**Table 49. All Supported *S0mode* (continued)**

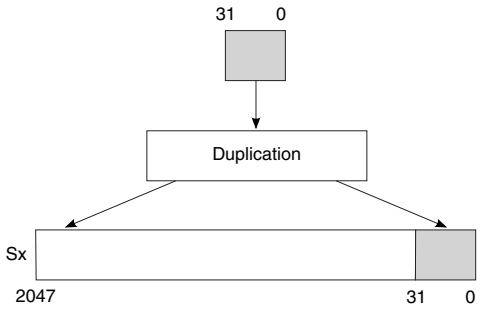
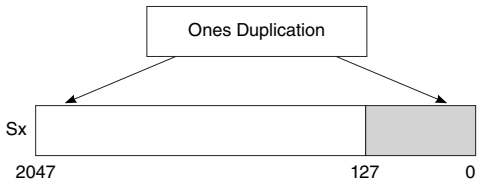
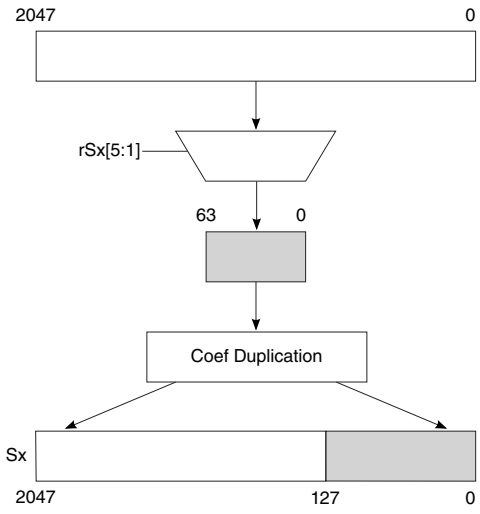
| s0mode | Restrictions | Descriptions[1] |
|---|---|---|
| | creg(15[0])=0,<br>creg(19)=x | Update rS0.<br>  |
| S0cplx1 | Complex data<br>creg(15[0])=0,<br>creg(19)=x | Load all complex elements of S0, in preparation for complex multiplication (cmad or cmac) with floating-point constant {0+1j, 1 +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

*Table continues on the next page...*

**Table 49. All Supported *S0mode* (continued)**

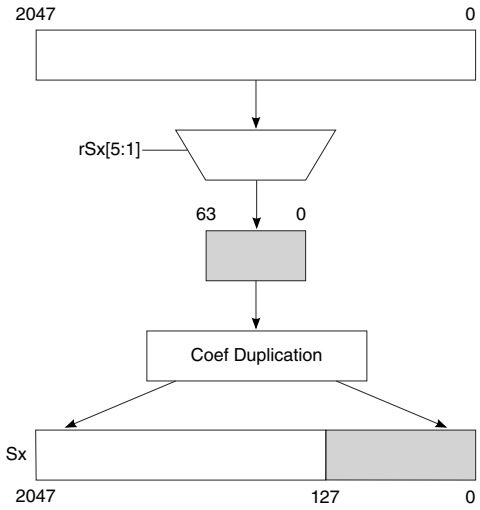
| s0mode | Restrictions | Descriptions[1] |
|---|---|---|
| S0i1i1r1r1 | Real and<br>Complex data<br>creg(15[0])=0, creg(19)=x | Load a complex element from output of the type converter, using rS0.<br>The complex element is duplicated into 4 real elements in the following format (imag, imag, real, real) and replicated into all elements of S0.<br>Update rS0.<br>  <br><br>* Note: If Sxprec is single, the rag bits used are rSx[5:2]. |
| S0fftn; | Complex data only,<br>S0prec=half_fixed /half/single,<br>AUprec=single, F24. | Using rS0, select 1 complex element from output of the type converter.<br>Replicate these elements into S0, in preparation for the DIT or DIF butterfly operation.<br>Each FFT complex element is duplicated with the following pattern (real,imag,-real,-imag,-imag,real,imag,-real).<br>FFT re-orders the complex elements as follows:<br>for(j=0; j<8; j++){<br>// first half output<br>mx_fft[127+j*128 : j*128] =<br>mx_fft_orig[127+j*256 : j*256];<br>// second half output<br>mx_fft[1024+127+j*128 : 1024+j*128]=<br>mx_fft_orig[127+j*256+128 : j*256+128];<br>}<br>Update rS0. |

*Table continues on the next page...*

**Table 49. All Supported *S0mode* (continued)**

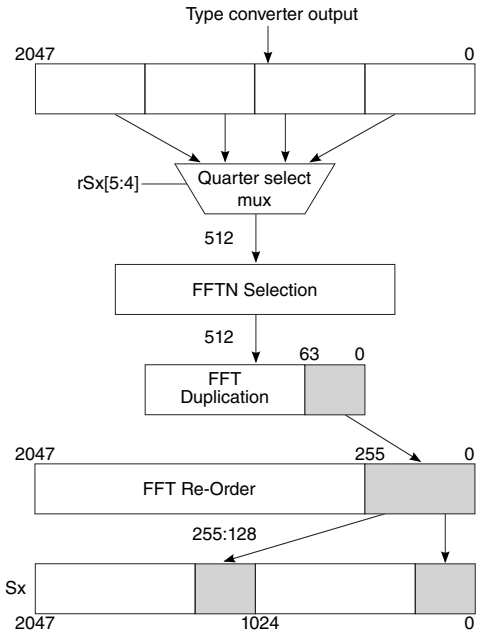
| s0mode | Restrictions | Descriptions[1] |
|---|---|---|
| | |  <br>* Note: If Sxprec is single, the quarter select mux only selects from the lower two quarters for FFT duplication. |
| S0fft4; | Complex data only,<br>S0prec=half_fixed /half/single,<br>AUprec=F24. | Using rS0, select 8 complex elements from the output of the type converter.<br>Replicate these elements into S0, in preparation for the DIT or DIF butterfly operations<br>Each FFT complex element is duplicated with the following pattern (real,imag,-real,-imag,-imag,real,imag,-real).<br>FFT re-orders the complex elements as follows:<br>for(j=0; j<8; j++){<br>// first half output<br>mx_fft[127+j*128 : j*128] =<br>mx_fft_orig[127+j*256 : j*256];<br>// second half output<br>mx_fft[1024+127+j*128 :1024+j*128]=<br>mx_fft_orig[127+j*256+128 : j*256+128];<br>}<br>Update rS0. |

*Table continues on the next page...*

**Table 49. All Supported *S0mode* (continued)**

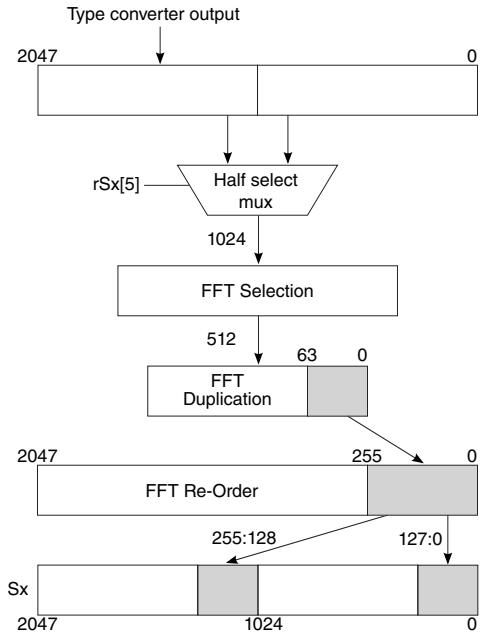
| s0mode | Restrictions | Descriptions[1] |
|---|---|---|
| | |  <br>* Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication. |
| S0fft3; | Complex data only,<br>S0prec=half_fixed /half/single,<br>AUprec=single, F24. | Using rS0, select 4 complex elements from the output of the type converter.<br>Replicate these elements into S0, in preparation for the DIT or DIF butterfly operations<br>Each FFT complex element is duplicated with the following pattern (real,imag,-real,-imag,-imag,real,imag,-real).<br>FFT re-orders the complex elements as follows:<br>for(j=0; j<8; j++){<br>// first half output<br>mx_fft[127+j*128 : j*128] =<br>mx_fft_orig[127+j*256 : j*256];<br>// second half output<br>mx_fft[1024+127+j*128 :1024+j*128]=<br>mx_fft_orig[127+j*256+128 : j*256+128];<br>}<br>Update rS0. |

*Table continues on the next page...*

**Table 49. All Supported *S0mode* (continued)**

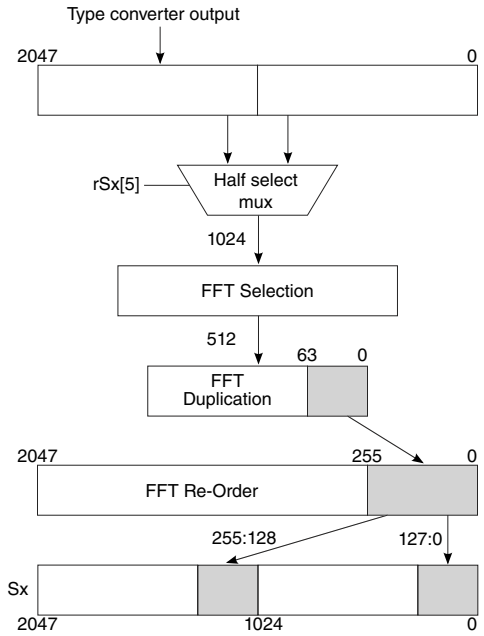
| s0mode | Restrictions | Descriptions[1] |
|---|---|---|
| | |  <br>* Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication. |
| S0fft2; | Complex data only,<br>S0prec=half_fixed /half/single,<br>AUprec=single, F24. | Using rS0, select 2 complex elements from the output of the type converter.<br>Replicate these elements into S0, in preparation for the DIT or DIF butterfly operation.<br>Each FFT complex element is duplicated with the following pattern (real,imag,-real,-imag,-imag,real,imag,-real).<br>FFT re-orders the complex elements as follows:<br>for(j=0; j<8; j++){<br>// first half output<br>mx_fft[127+j*128 : j*128] =<br>mx_fft_orig[127+j*256 : j*256];<br>// second half output<br>mx_fft[1024+127+j*128 :1024+j*128]=<br>mx_fft_orig[127+j*256+128 : j*256+128];<br>}<br>Update rS0. |

*Table continues on the next page...*

**Table 49. All Supported *S0mode* (continued)**

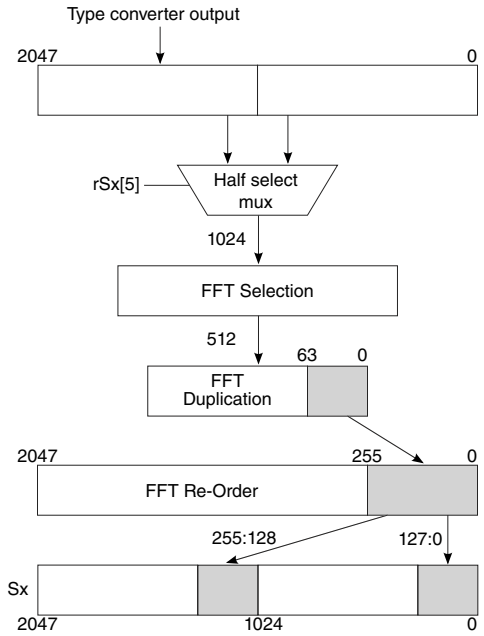
| s0mode | Restrictions | Descriptions[1] |
|---|---|---|
| | |  <br>* Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication. |
| S0fft1; | Complex data only,<br>S0prec=half_fixed /half/single,<br>AUprec=single, F24. | Using rS0, select 1 complex elements from the output of the type converter.<br>Replicate these elements into S0, in preparation for the DIT or DIF butterfly operation.<br>Each FFT complex element is duplicated with the following pattern (real,imag,-real,-imag,-imag,real,imag,-real).<br>FFT re-orders the complex elements as follows:<br>for(j=0; j<8; j++){<br>// first half output<br>mx_fft[127+j*128 : j*128] =<br>mx_fft_orig[127+j*256 : j*256];<br>// second half output<br>mx_fft[1024+127+j*128 :1024+j*128]=<br>mx_fft_orig[127+j*256+128 : j*256+128];<br>}<br>Update rS0. |

*Table continues on the next page...*

**Table 49. All Supported *S0mode* (continued)**

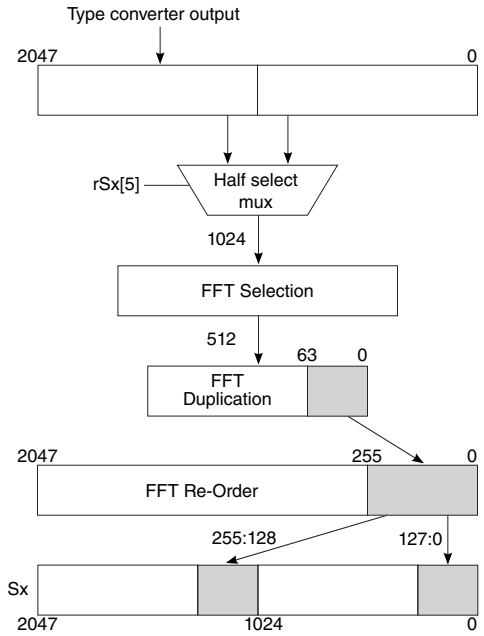
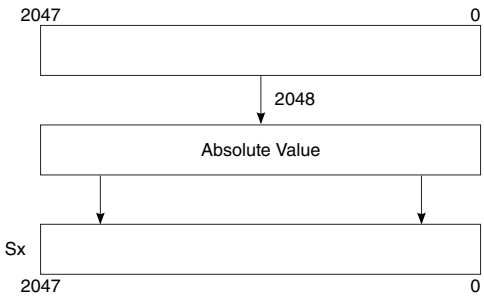
| s0mode | Restrictions | Descriptions[1] |
|---|---|---|
| | |  <br>* Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication. |
| S0abs | Real and<br>Complex data<br>creg(15[0])=0, creg(19)=x | Load S0 with absolute value of each element.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

*Table continues on the next page...*

**Table 49. All Supported *S0mode* (continued)**

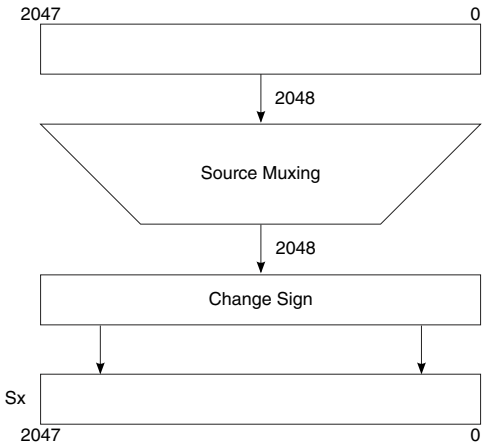
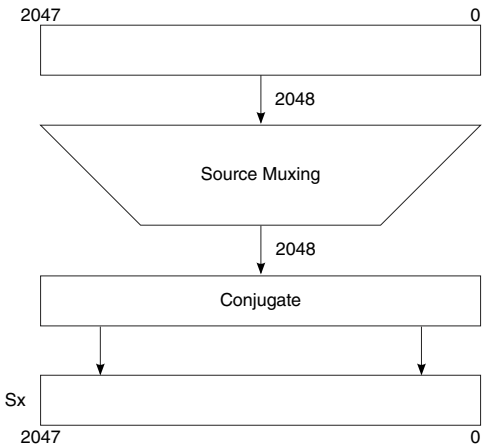
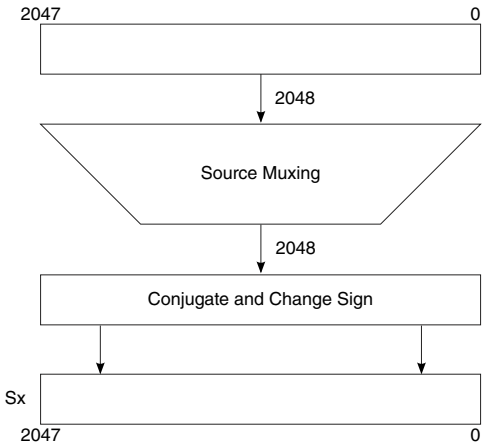
| s0mode | Restrictions | Descriptions[1] |
|---|---|---|
| | |  |
| <s0mode>, S0conj; | Real and Complex data | Perform conjugate on all the complex data elements; that is, invert the sign bit of all imaginary elements.<br>  |
| <s0mode>, S0conj, sign; | Real and Complex data | Perform both conjugate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

*Table continues on the next page...*

**Table 49. 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 50 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 or 128 20 bit 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 or 128 20 bit elements.

Table 50 illustrates S1modes in AUprec single. All S1modes except S1qline, the CMM modes (S1interp2nr, S1interp2nc), and CGU modes (S1cgu_hlinecplx, S1cgu_i2i1r2r1, S1cgu_r2c_im0, S1cgu_r2c_re0) are also supported in F24 mode. In F24 precision, the output of the type converter will be 2560 bits instead of 2048 bits. Thus, the half select mux will select 1280 MSB or LSB bits instead of 1024. The size of elements that get duplicated will be scaled by a factor of 20 instead of 32. For example, in S1hlinecplx the size of the element would be 40 instead of 64. The source register Sx (S1 for S1mode) will also be larger.

**Table 50. 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).<br>Each complex element is duplicated with the following pattern (imag,real,imag,real).<br>Update rS1. |

*Table continues on the next page...*

**Table 50. All Supported *S1mode* (continued)**

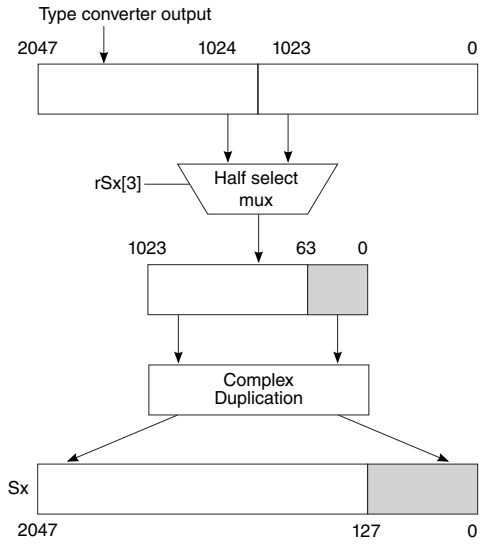
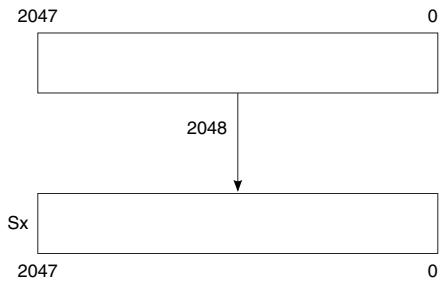
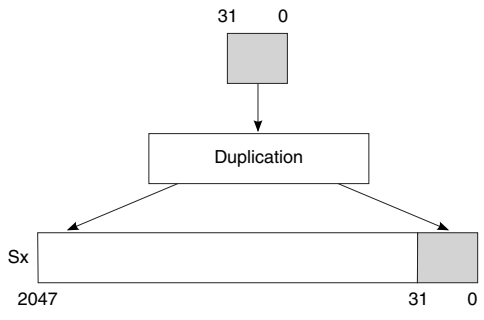
| S1mode | Restrictions | Descriptions |
|---|---|---|
| | |  <br>* Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication. |
| S1straight | Real data | Load S1 with output of the type converter.<br>Update rS1.<br>  |
| S1zeros | Real and<br>Complex data | Load all elements of S1 with floating-point constant "0".<br>Update rS1.<br>  |

*Table continues on the next page...*

**Table 50. All Supported *S1mode* (continued)**

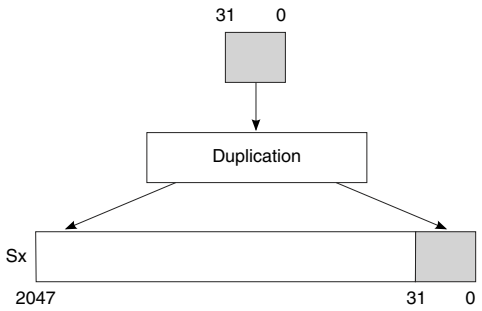
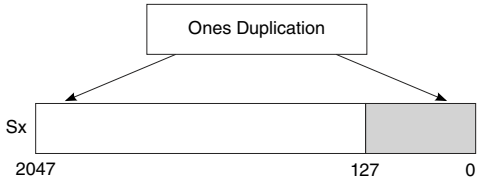
| S1mode | Restrictions | Descriptions |
|---|---|---|
| S1real1 | Real data | Load all elements of S1 with floating-point constant "1".<br>Update rS1.<br>**NOTE**<br>This S1 mode will only produce floating point values and so may NOT be used in conjuction with AUprec padd or paddF24, which require values in half fixed format.<br>  |
| 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}.<br>Update rS1.<br>**NOTE**<br>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.<br>  |
| S1real_conj | Real data | Same as "straight" with post-conjugate operation; that is, invert the sign bit of the odd elements. |

*Table continues on the next page...*

**Table 50. All Supported *S1mode* (continued)**

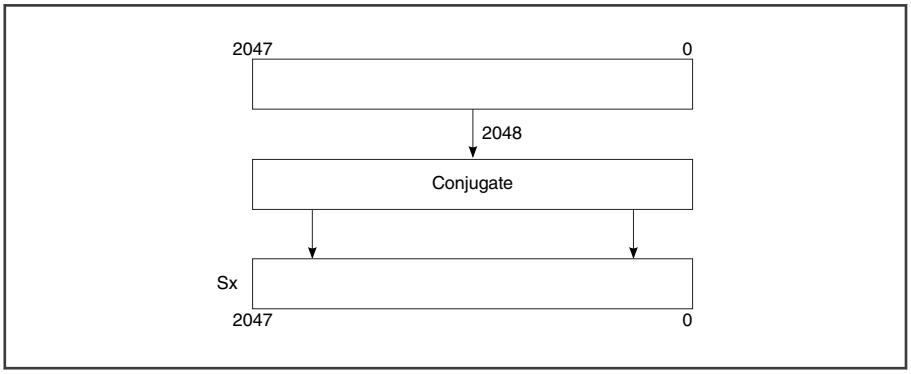
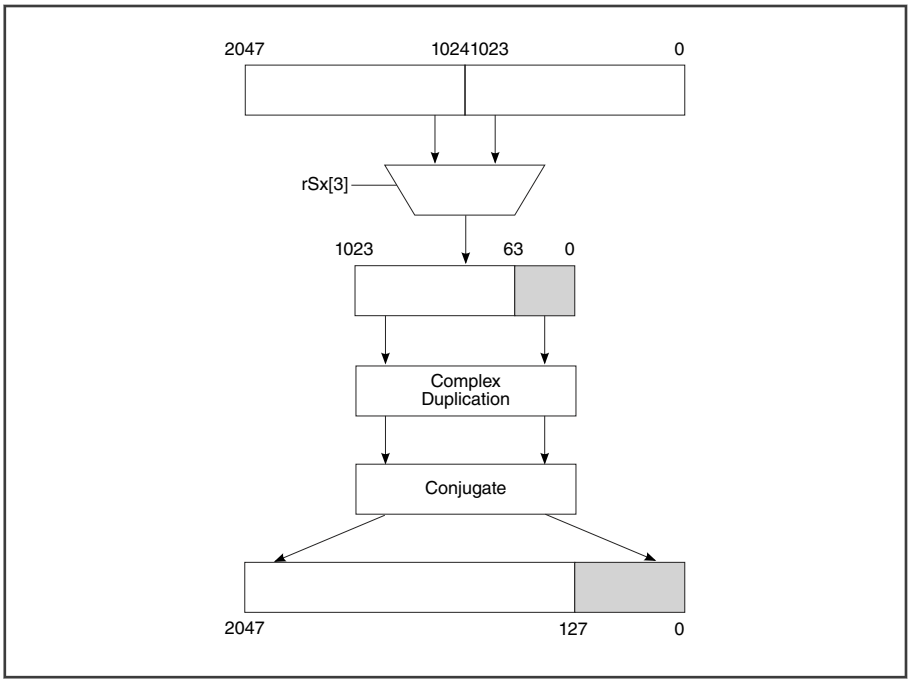
| S1mode | Restrictions | Descriptions |
|---|---|---|
| | |  |
| S1cplx_conj | Complex data | Same as "hlinecplx" with post-conjugate operation; that is, invert the sign bit of the imaginary elements.<br>  |
| S1r2c | Real and Complex data | Load S1 real-to-complex with output of the type converter. Each complex element is replicated as follows (imag,imag,real,real) into S1.<br>Update rS1. |

*Table continues on the next page...*

**Table 50. All Supported *S1mode* (continued)**

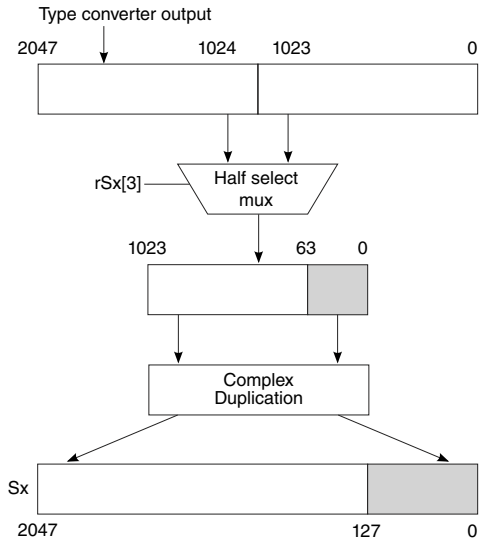
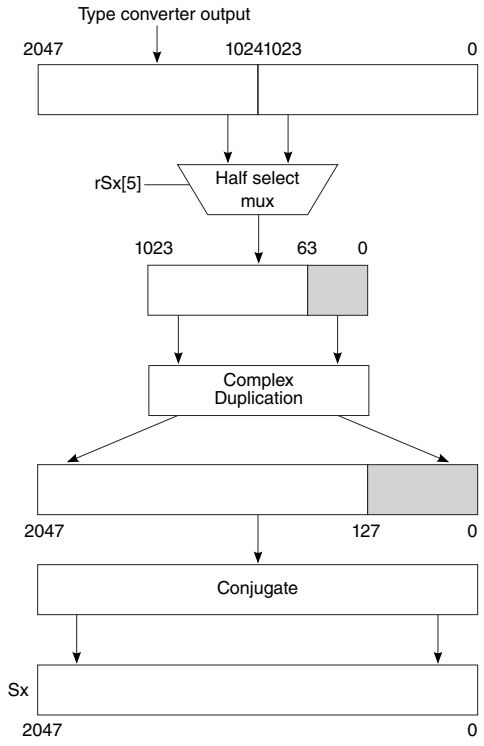
| S1mode | Restrictions | Descriptions |
|---|---|---|
| | |  <br>* Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication. |
| S1r2c_conj | Real and<br>Complex data | Same as "r2c" with post-conjugate operation; that is, invert the sign bit of the imaginary elements.<br>  |

*Table continues on the next page...*

**Table 50. All Supported *S1mode* (continued)**

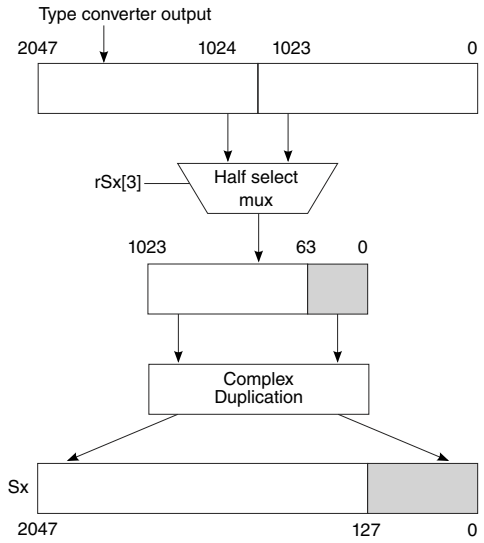
| S1mode | Restrictions | Descriptions |
|---|---|---|
| | | * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication. |
| S1r2c_im0 | Real and<br>Complex data | Load S1 real-to-complex with output of the type converter. Each complex element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

*Table continues on the next page...*

**Table 50. All Supported *S1mode* (continued)**

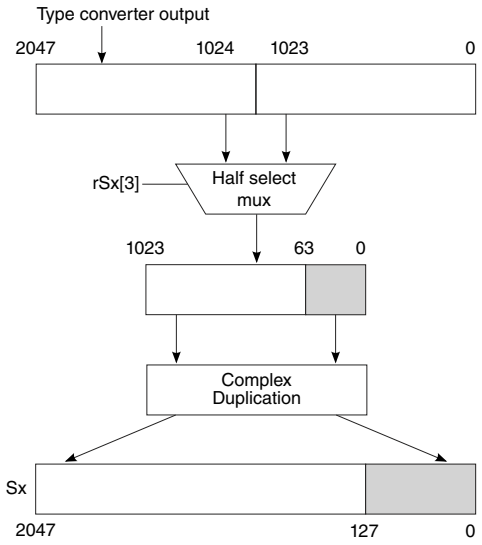
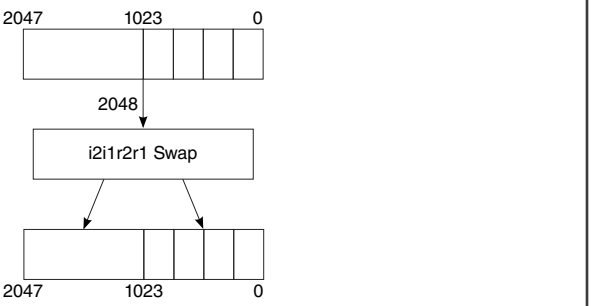
| S1mode | Restrictions | Descriptions |
|---|---|---|
| | |  <br>* Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication. |
| S1i2i1r2r1 | Real data | Same as S1hlinecplx |
| | Complex data | The i2i1r2r1 mode is for up-down filter with complex inputs<br>Load S1 with output of the type converter, swapping the 2nd and 3rd elements of each group of 4 elements.<br>Update rS1.<br>  |
| | Complex data | The udfR mode is for up-down filter with real inputs<br>Same as "straight" mode. |
| S1interp2nr | Real data<br>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.<br>Duplicate elements into S1:<br>for (i = 0; i < 4*NAU/n; i++) { |

*Table continues on the next page...*

**Table 50. All Supported *S1mode* (continued)**

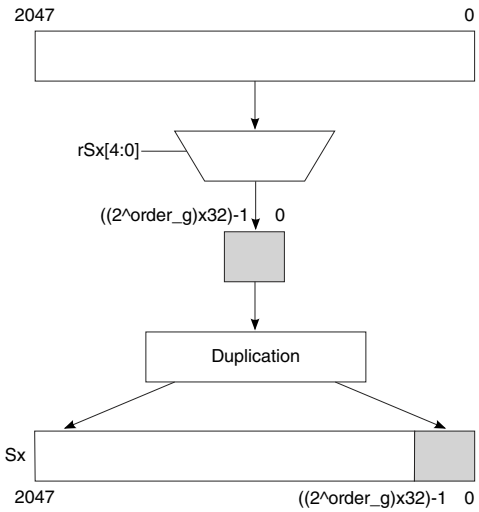
| S1mode | Restrictions | Descriptions |
|---|---|---|
| | | for (j = 0; j < n; j++) {<br>S1[n*i+j] = r[i+rS1];<br>}<br>}<br>Number of elements (n) in group is determined by 2^order_i<br>Value of order_i is restricted such that:<br>1 < 2^order_i <= NAU*4<br>Update rS1.<br>  <br> |
| S1interp2nc | Complex data<br>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.<br>Duplicate elements into S1:<br>for (i = 0; i < NAU/n; i++) {<br>for (j = 0; j < n; j++) {<br>S1[4*n*i+4*j] = r[2*(i+rS1)];<br>S1[4*n*i+4*j+1] = r[2*(i+rS1)+1];<br>S1[4*n*i+4*j+2] = r[2*(i+rS1)];<br>S1[4*n*i+4*j+3] = r[2*(i+rS1)+1];<br>}<br>}<br>Number of elements (n) in group is determined by 2^order_i<br>Value of order_i is restricted such that:<br>1 < 2^order_i <= NAU<br>Update rS1. |

*Table continues on the next page...*

**Table 50. All Supported *S1mode* (continued)**

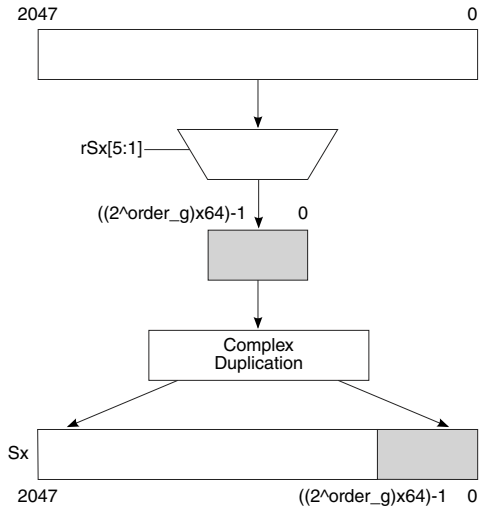
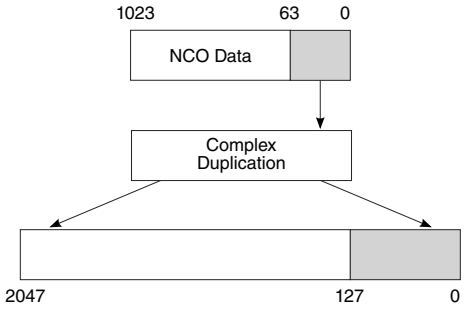
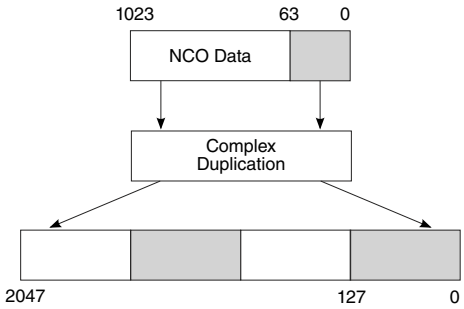
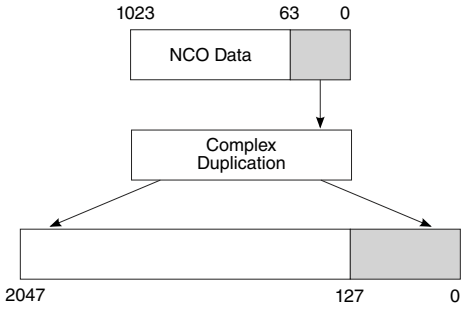
| S1mode | Restrictions | Descriptions |
|---|---|---|
| | |  |
| S1nco | Complex data Only<br>creg(25)=0, creg(19)=0 | Load S1 with complex tone samples being generated by nco module.<br>Update rS1.<br>**NOTE**<br>The Vector NCO is capable of producing single-precision or 24 bit precision twiddle factors. If *S1nco*, then the programmer must constrain AUprec to be single or f24 and AUprec must be set to the desired mode (single or f24) before using the NCO.<br>When nco_mode=singles, take first complex element and duplicate with the following pattern (imag,real,imag,real).<br>  <br>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 50. All Supported *S1mode* (continued)**

| S1mode | Restrictions | Descriptions |
|---|---|---|
| | |  <br>When nco_mode=normal, duplicate each complex element with the following pattern (imag,real,imag,real).<br>  |
| S1cgu_hlinecplx | Complex data | Same as "hlinecplx" with the CGU modification on the data being loaded into S1.<br>Update the CGU state. |
| S1cgu_straight | Real and Complex data | Same as "straight" with the CGU modification on the data being loaded into S1.<br>Update the CGU state. |
| S1cgu_i2i1r2r1 | Real and Complex data | Same as "i2i1r2r1" with the CGU modification on the data being loaded into S1.<br>Update the CGU state. |
| S1cgu_r2c_im0 | Real and Complex data | Same as "r2c_im0" with the CGU modification on the data being loaded into S1.<br>Update the CGU state. |
| S1cgu_r2c_re0 | Real and Complex data | Same as "r2c_re0" with the CGU modification on the data being loaded into S1.<br>Update the CGU state. |

### 4.12.6 *S2mode* options and detailed description

Table 51 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 or 128 20 bit 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 or 128 20 bit elements.

Table 51 illustrates S2modes in AUprec single. In F24 precision, the output of the type converter will be 2560 bits instead of 2048 bits. Thus, the half select mux will select 1280 MSB or LSB bits instead of 1024. The size of elements that get duplicated will be scaled by a factor of 20 instead of 32. For example, in S2hlinecplx the size of the element would be 40 instead of 64. The source register Sx (S2 for S2mode) will also be larger.

**Table 51. All Supported *S2mode***

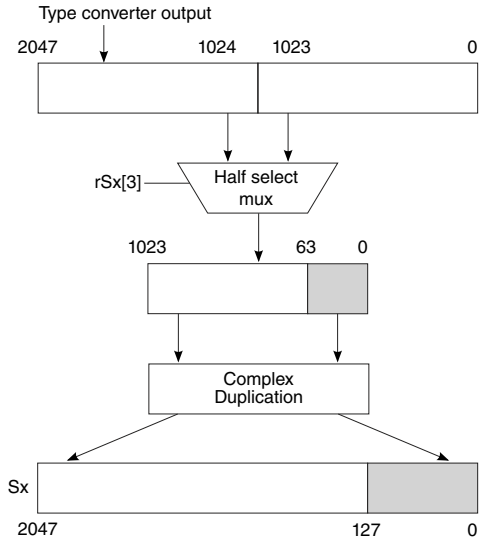
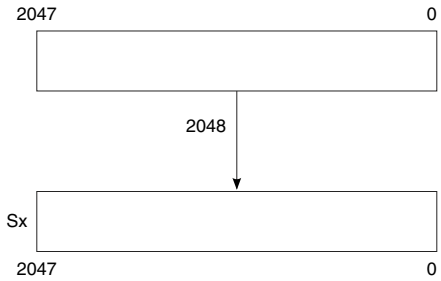
| S2mode | Restrictions | Descriptions |
|---|---|---|
| S2hlinecplx | Complex data<br>creg(15[0])=x,<br>creg(19)=x | Load S2 with output of the type converter, in preparation for complex multiplication (cmad or cmac).<br>Each complex element is duplicated with the following pattern (0,imag,0,real).<br>Update rS2.<br>  <br><br>* 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.<br>Update rS2.<br>  <br> |
| S2zeros | Real and<br>Complex data | Load all elements of S2 with floating-point constant "0".<br>Update rS2. |

*Table continues on the next page...*

**Table 51. All Supported *S2mode* (continued)**

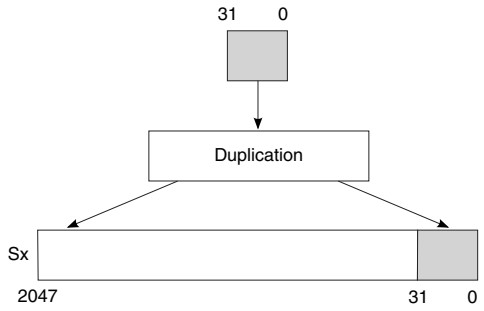
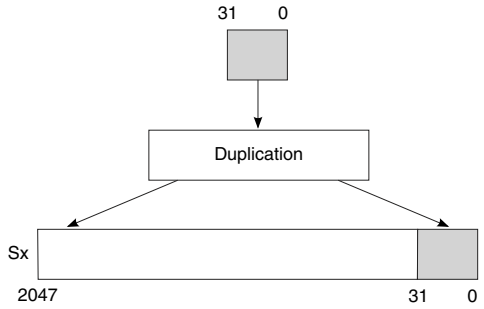
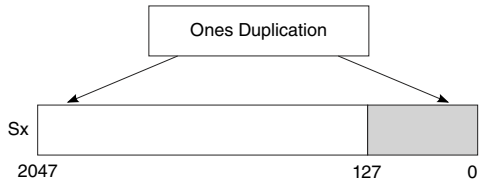
| S2mode | Restrictions | Descriptions |
|---|---|---|
| | |  |
| S2real1 | Real data | Load all elements of S2 with floating-point constant "1".<br>Update rS2.<br>**NOTE**<br>This S2 mode will only produce floating point values and so may NOT be used in conjuction with AUprec padd or paddF24, which require values in half fixed format.<br>  |
| 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}.<br>Update rS2.<br>**NOTE**<br>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.<br>  |
| S2i1r1i1r1 | Real and | Load a complex element from the output of the type converter, using rS2. |

*Table continues on the next page...*

**Table 51. All Supported *S2mode* (continued)**

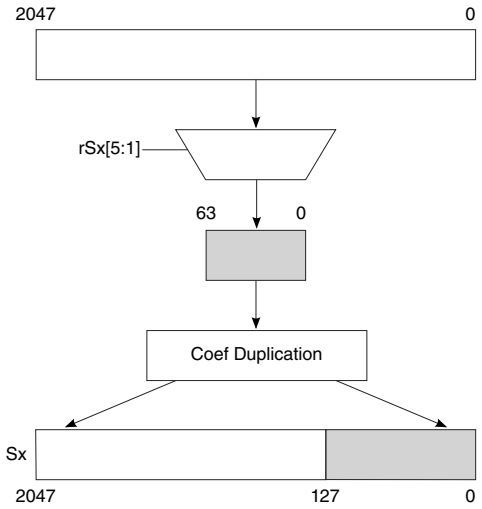
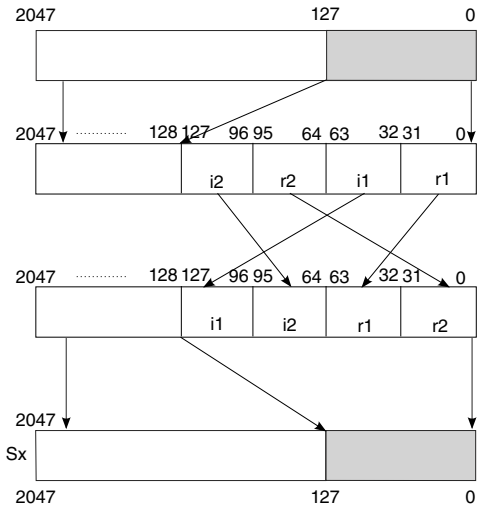
| S2mode | Restrictions | Descriptions |
|---|---|---|
| | Complex data | The complex element is expanded into 4 real elements in the following format (imag, real, imag, real) and replicated into all elements of S2.<br>Update rS2.<br>  <br>* Note: If Sxprec is single, the rag bits used are rSx[5:2]. |
| S2i1i2r1r2 | Real and<br>Complex data | Load complex elements from the output of the type converter, using rS2.<br>The complex elements are re-ordered in the following format (imag1, imag2, real1, real2) and loaded into S2.<br>  |
| S2fftn; | Complex data only,<br>S2prec=half_fixed /half/single, | Using rS2, select 2 complex elements from the output of the type converter.<br>Replicate these elements into S2, in preparation for the DIT or DIF butterfly operation. |

*Table continues on the next page...*

**Table 51. All Supported *S2mode* (continued)**

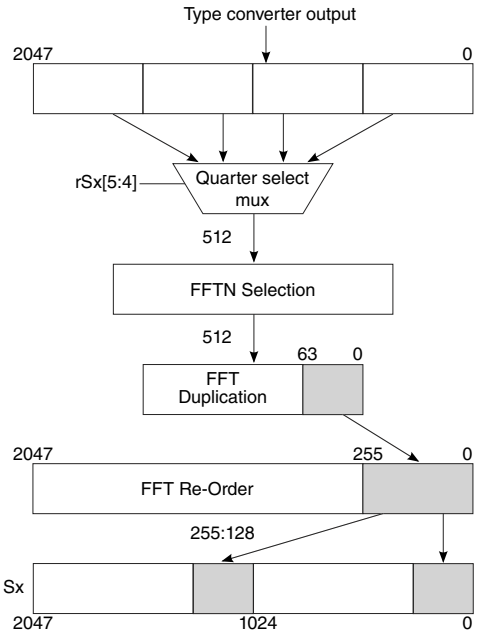
| S2mode | Restrictions | Descriptions |
|---|---|---|
| | AUprec=single, F24.<br>creg(15[0])=1, creg(19)=0 | Each FFT complex element is duplicated with the following pattern (real,imag,0,imag,-imag,real,0,real).<br>FFT re-orders the complex elements as follows:<br>for(j=0; j<8; j++){<br>// first half output<br>mx_fft[127+j*128 : j*128] =<br>mx_fft_orig[127+j*256 : j*256];<br>// second half output<br>mx_fft[1024+127+j*128 : 1024+j*128]=<br>mx_fft_orig[127+j*256+128 : j*256+128];<br>}<br>Update rS2.<br>  <br><br>* Note: If Sxprec is single, the quarter select mux only selects from the lower two quarters for FFT duplication. |
| S2fft4; | Complex data only,<br>S2prec=half_fixed /half/single,<br>AUprec=F24. | Using rS2, select 8 complex elements from the output of the type converter.<br>Replicate these elements into S2, in preparation for the DIT or DIF butterfly operations<br>Each FFT complex element is duplicated with the following pattern (real,imag,0,imag,-imag,real,0,real).<br>FFT re-orders the complex elements as follows:<br>for(j=0; j<8; j++){ |

*Table continues on the next page...*

**Table 51. All Supported *S2mode* (continued)**

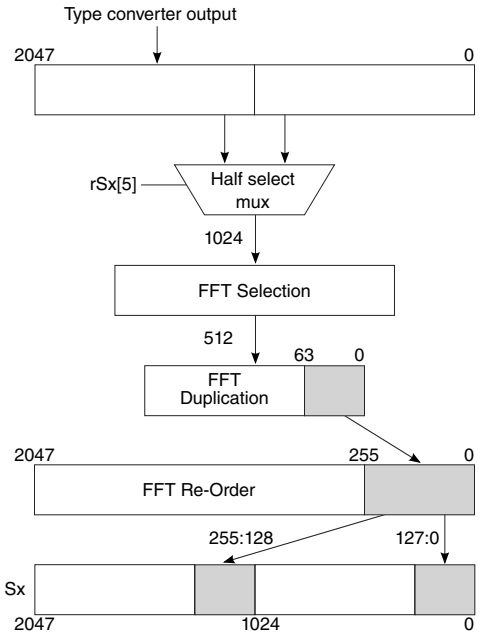
| S2mode | Restrictions | Descriptions |
|---|---|---|
| | creg(15[0])=1, creg(19)=0 | // first half output<br>mx_fft[127+j*128 : j*128] =<br>mx_fft_orig[127+j*256 : j*256];<br>// second half output<br>mx_fft[1024+127+j*128 : 1024+j*128]=<br>mx_fft_orig[127+j*256+128 : j*256+128];<br>}<br>Update rS2.<br>  <br><br>* 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,<br>S2prec=half_fixed /half/single,<br>AUprec=single, F24.<br>creg(15[0])=1, creg(19)=0 | Using rS2, select 4 complex elements from the output of the type converter.<br>Replicate these elements into S2, in preparation for the DIT or DIF butterfly operations<br>Each FFT complex element is duplicated with the following pattern (real,imag,0,imag,-imag,real,0,real).<br>FFT re-orders the complex elements as follows:<br>for(j=0; j<8; j++){<br>// first half output<br>mx_fft[127+j*128 : j*128] =<br>mx_fft_orig[127+j*256 : j*256]; |

*Table continues on the next page...*

**Table 51. All Supported *S2mode* (continued)**

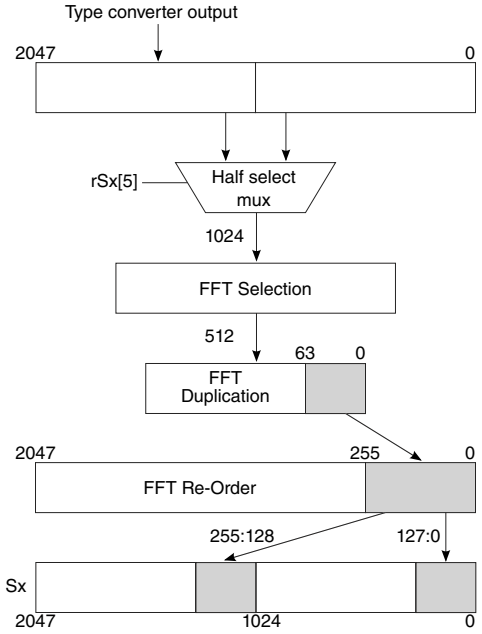
| S2mode | Restrictions | Descriptions |
|---|---|---|
| | | // second half output<br>mx_fft[1024+127+j*128 : 1024+j*128]=<br>mx_fft_orig[127+j*256+128 : j*256+128];<br>}<br>Update rS2.<br>  <br><br>* Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication. |
| S2fft2; | Complex data only,<br>S2prec=half_fixed /half/single,<br>AUprec=single, F24.<br>creg(15[0])=1, creg(19)=0 | Using rS2, select 2complex elements from the output of the type converter.<br>Replicate these elements into S2, in preparation for the DIT or DIF butterfly operations<br>Each FFT complex element is duplicated with the following pattern (real,imag,0,imag,-imag,real,0,real).<br>FFT re-orders the complex elements as follows:<br>for(j=0; j<8; j++){<br>// first half output<br>mx_fft[127+j*128 : j*128] =<br>mx_fft_orig[127+j*256 : j*256];<br>// second half output<br>mx_fft[1024+127+j*128 : 1024+j*128]=<br>mx_fft_orig[127+j*256+128 : j*256+128]; |

*Table continues on the next page...*

**Table 51. All Supported *S2mode* (continued)**

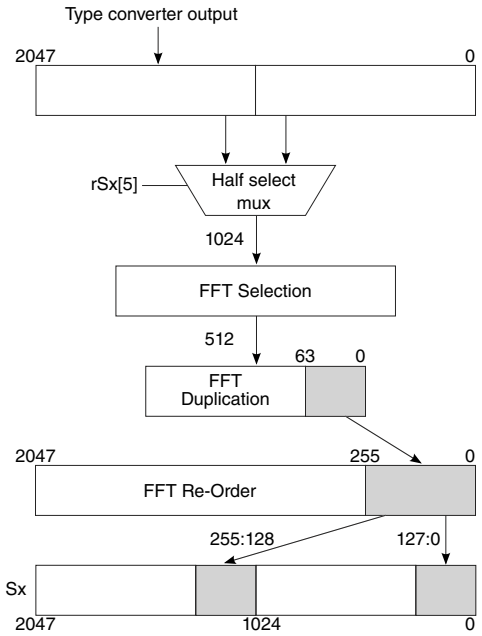
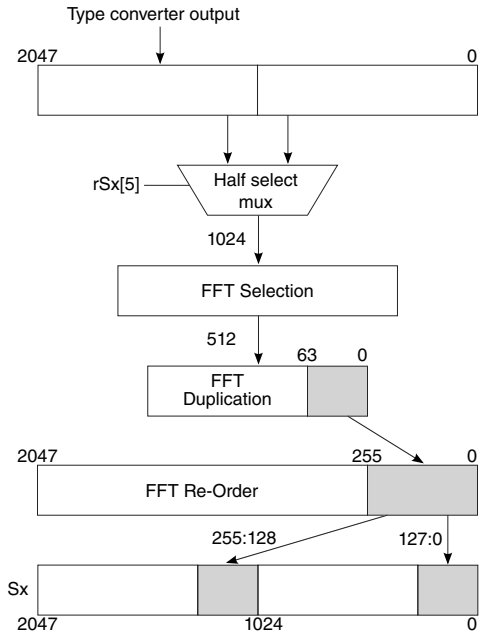
| S2mode | Restrictions | Descriptions |
|---|---|---|
| | | }<br>Update rS2.<br>  <br><br>* Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication. |
| S2fft1; | Complex data only,<br>S2prec=half_fixed /half/single,<br>AUprec=single, F24.<br>creg(15[0])=1, creg(19)=0 | Using rS2, select 1 complex elements from the output of the type converter.<br>Replicate these elements into S2, in preparation for the DIT or DIF butterfly operations<br>Each FFT complex element is duplicated with the following pattern (real,imag,0,imag,-imag,real,0,real).<br>FFT re-orders the complex elements as follows:<br>for(j=0; j<8; j++){<br>// first half output<br>mx_fft[127+j*128 : j*128] =<br>mx_fft_orig[127+j*256 : j*256];<br>// second half output<br>mx_fft[1024+127+j*128 : 1024+j*128]=<br>mx_fft_orig[127+j*256+128 : j*256+128];<br>}<br>Update rS2. |

*Table continues on the next page...*

**Table 51. All Supported *S2mode* (continued)**

| S2mode | Restrictions | Descriptions |
|---|---|---|
| | |  <br>* 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 52. 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]<br>for i=0,1,..,63 | Undefined |
| rmad.sau | V[i][n] = (S0[i][n-4]*SAUout[i][n-4]) + S2[i][n-4]<br>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]<br>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]<br>for i=0,2,4,6,...,62 |
| rmac | V[i][n] = (S0[i][n-4]*S1[i][n-4]) + V[i][n-1]<br>for i=0,1,..,63 | Undefined |
| rmac.sau | V[i][n] = (S0[i][n-4]*SAUout[i][n-4]) + V[i][n-1]<br>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]<br>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]<br>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]<br>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]<br>for i=0,1,..,63 | Undefined |
| maf | V[i][n] = (V[i][n-4]*S1[i][n-4]) + S2[i][n-4]<br>for i=0,1,..,63 | Undefined |
| mafac | V[i][n] = (V[i][n-4]*S1[i][n-4]) + V[i][n-1]<br>for i=0,1,..,63 | Undefined |

*Table continues on the next page...*

**Table 52. 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]);<br>V[i+2][n]=S0[i+2][n-4] + S2[i+2][n-4];<br>for i=0,4,..,60 |
| clr.au | Clear Vector Accumulator<br>V[i] = 0<br>for i=0,1,..,63 | Clear Vector Accumulator<br>V[i] = 0<br>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, 128 real elements in F24 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 and F24 mode.
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 or F24 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;
```

#### 4.13.2.1 Complex number power calculation

The example below computes the abs()^2, |z|^2 = abs2(z) = a^2+b^2.

Here , z=a+ib;

```
 __ld_Rx_mem(rX0, pInput_c);
/* get a + ib */
/* computes power a^2 + b^2 of complex number */
__set_VRAptr_rS0(_VSPA_VRA_REG_OFFSET(rX0));
__set_VRAptr_rS1(_VSPA_VRA_REG_OFFSET(rX0));
__set_VRAptr_rV(_VSPA_VRA_REG_OFFSET(rX1));
__set_Smode(S0straight, S1straight, S2zeros);
__rd_S0();
__rd_S1();
__rd_S2();
```

```
__cmad(); /* V = R0 x R0 + 0 */
__wr(straight); /* R1 = V */
```

### 4.13.3 Multiply add functionality

**Table 53. MAD Descriptions**

| Instructions | Real/ Complex Modes | AUprec | Operations | Descriptions |
|---|---|---|---|---|
| rmad | Real | Single | for (i=0; i<64; i++) {<br>V[i][n] = (S0[i][n-4]*S1[i][n-4]) + S2[i][n-4]<br>} | Perform 64 parallel multiply-add operations on S0/S1/S2 elements. |
| rmad | Real | F24 | for (i=0; i<128; i++) {<br>V[i][n] = (S0[i][n-4]*S1[i][n-4]) + S2[i][n-4]<br>} | Perform 128 parallel multiply-add operations on S0/S1/S2 elements. |
| cmad | Complex | Single | for (i=0; i<64; i +=2) {<br>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]<br>} | This can perform 16 complex multiply-add operations, where V[0] represent the real outputs and V[2] represent the imaginary outputs. |
| cmad | Complex | F24 | for (i=0; i<128; i +=2) {<br>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]<br>} | This can perform 32 complex multiply-add operations, where V[0] represent the real outputs and V[2] represent the imaginary outputs. |

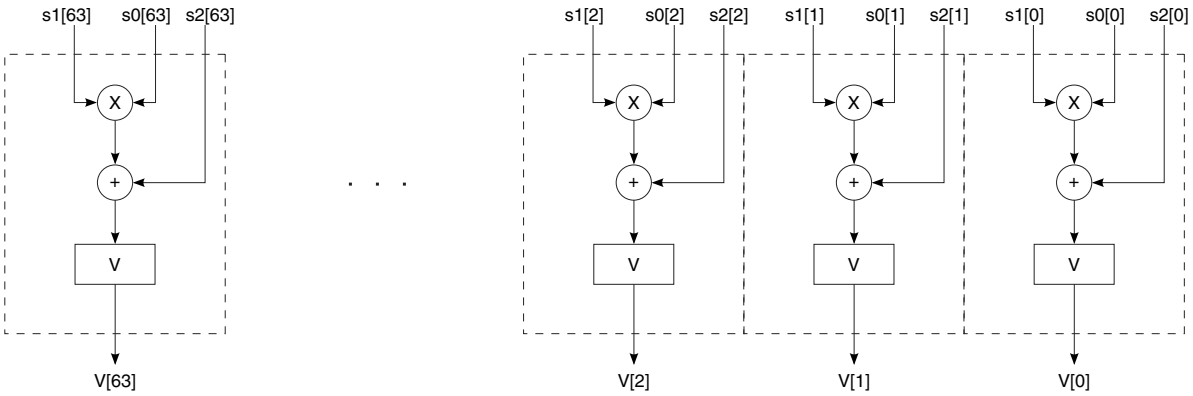



**Figure 28. Real mad Operation (Single Precision)**

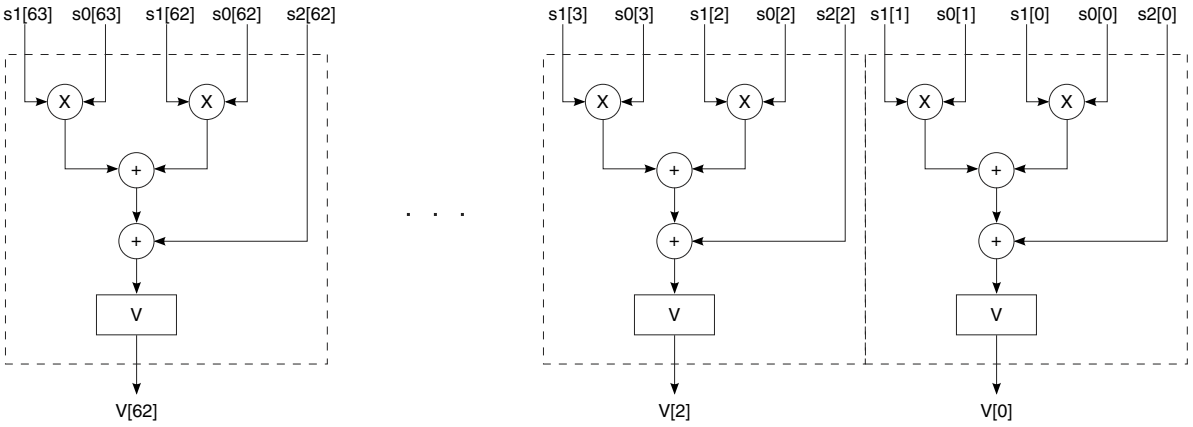



**Figure 29. Complex mad Operation (Single Precision)**

### 4.13.4 Multiply and add with sign conversion

**Table 54. MADS Description**

| Instructions | Real/ Complex Modes | AUprec | Operations | Descriptions |
|---|---|---|---|---|
| mads | Real Only | Single | for (i=0; i<64; i++) {<br>V[i][n] = (S0[i][n-4]*S1[i][n-4]) + sign( (S0[i][n-4]*S1[i][n-4]))*S2[i][n-4]<br>}<br>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.<br>This instruction is useful in using the AU to perform float-to-fixed or fixed-to-float conversion |
| mads | Real Only | F24 | for (i=0; i<128; i++) {<br>V[i][n] = (S0[i][n-4]*S1[i][n-4]) + sign( (S0[i][n-4]*S1[i][n-4]))*S2[i][n-4]<br>}<br>where sign() function returns 1 or -1 based on the sign bit of the input. | Perform 128 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.<br>This instruction is useful in using the AU to perform float-to-fixed or fixed-to-float conversion |

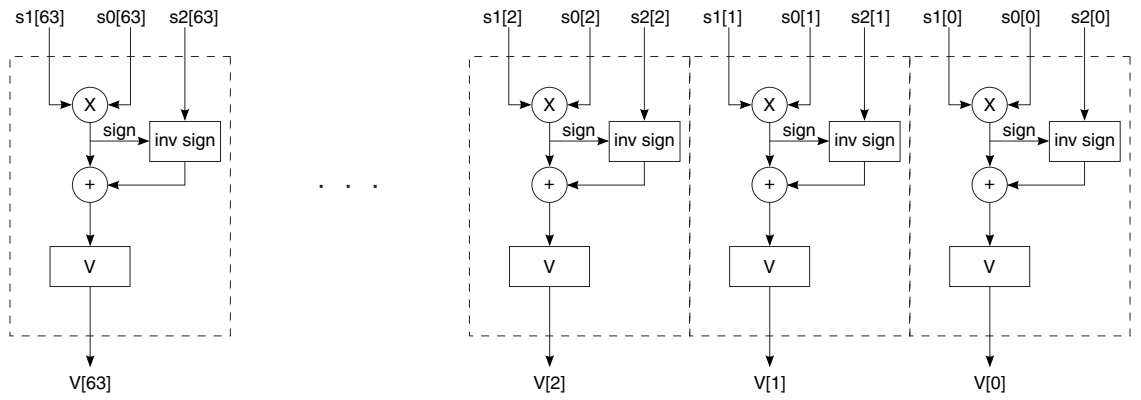



**Figure 30. mads Operation in Real Mode (Single Precision)**

### 4.13.5 Multiply accumulate functionality

**Table 55. MAC Descriptions**

| Instructions | Real/ Complex Modes | AUprec | Operations | Descriptions |
|---|---|---|---|---|
| rmac | Real | Single | for (i=0; i<64; i++) {<br>V[i][n] = (S0[i][n-4]*S1[i][n-4]) + V[i][n-1]<br>} | Perform 64 parallel multiply-accumulate operations on S0/S1 elements. |
| rmac | Real | F24 | for (i=0; i<128; i++) {<br>V[i][n] = (S0[i][n-4]*S1[i][n-4]) + V[i][n-1]<br>} | Perform 128 parallel multiply-accumulate operations on S0/S1 elements. |
| cmac | Complex | Single | for (i=0; i<64; i +=2) {<br>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]<br>} | This can perform 16 complex multiply-accumulate operations, where V[0] represent the real accumulators and V[2] represent the imaginary accumulators. |
| cmac | Complex | F24 | for (i=0; i<128; i +=2) {<br>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]<br>} | This can perform 32 complex multiply-accumulate operations, where V[0] represent the real accumulators and V[2] represent the imaginary accumulators. |

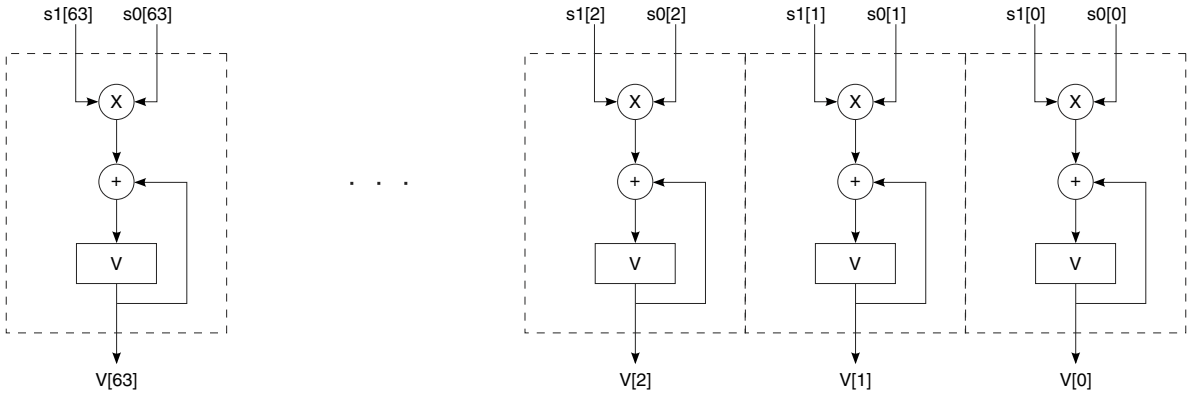



**Figure 31. Real mac Operation (Single Precision)**

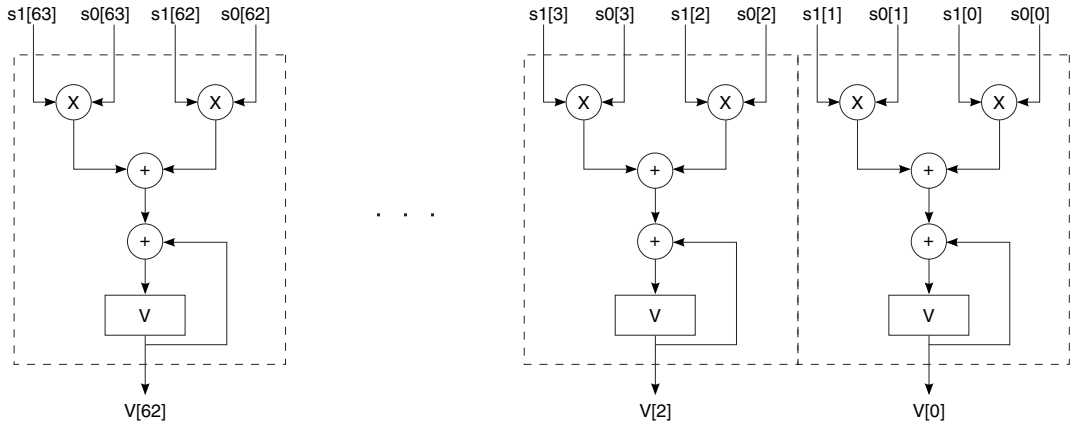



**Figure 32. Complex mac Operation (Single Precision)**

**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 56. MAF Descriptions**

| Instructions | Real/ Complex Modes | AUprec | Operations | Descriptions |
|---|---|---|---|---|
| maf | Real | Single | for (i=0; i<64; i++) {<br>V[i][n] = (V[i][n-4]*S1[i][n-4]) + S2[i][n-4]<br>} | Perform 64 parallel multiply-accumlate with feedback operations with S1 and S2 elements. |
| maf | Real | F24 | for (i=0; i<128; i++) {<br>V[i][n] = (V[i][n-4]*S1[i][n-4]) + S2[i][n-4]<br>} | Perform 128 parallel multiply-accumlate with feedback operations with S1 and S2 elements. |

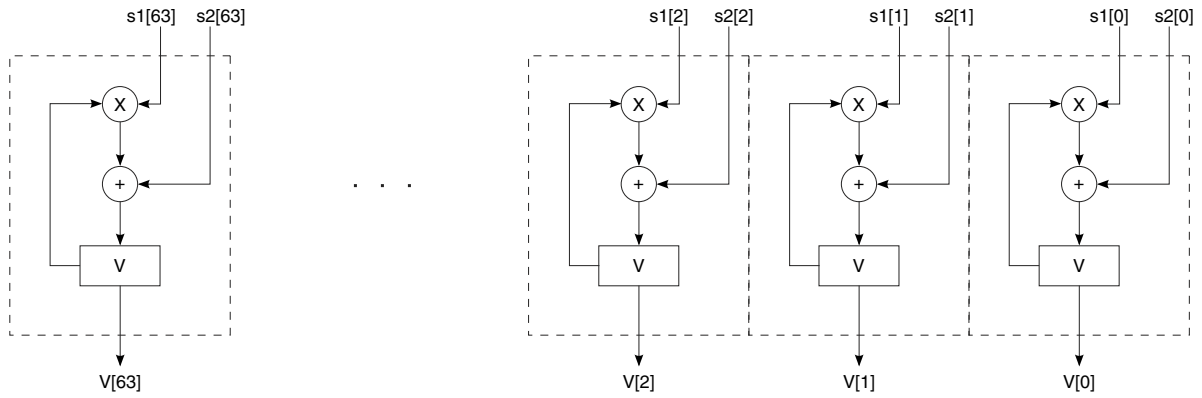



**Figure 33. maf Operation (Single Precision)**

### 4.13.7 Decimation in time and frequency butterfly functionality

**Table 57. Decimation Descriptions**

| Instructions | Real/ Complex Modes | AUprec | Operations | Descriptions |
|---|---|---|---|---|
| cmad | Complex Only | Single Precision Only | for (i=0; i<64; i +=2) {<br>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]<br>} | Perform 32 parallel multiply-add-add operations on S0/S1/S2 elements.<br>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) {<br>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]);<br>V[i+2][n]=S0[i+2][n-4] + S2[i+2][n-4];<br>} | Perform 32 parallel FFT-specific operations on S0/S1/S2 elements.<br>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. |

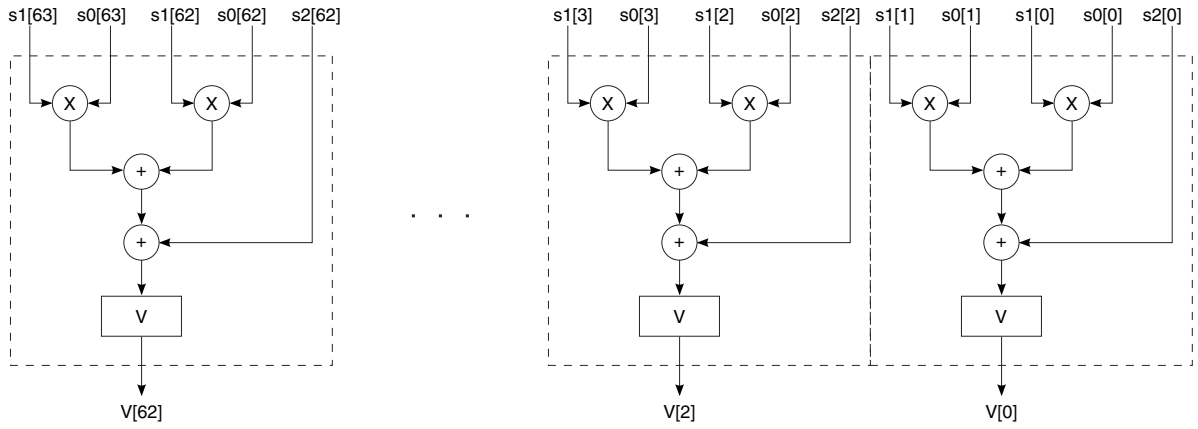



**Figure 34. cmad Operation (Single Precision)**

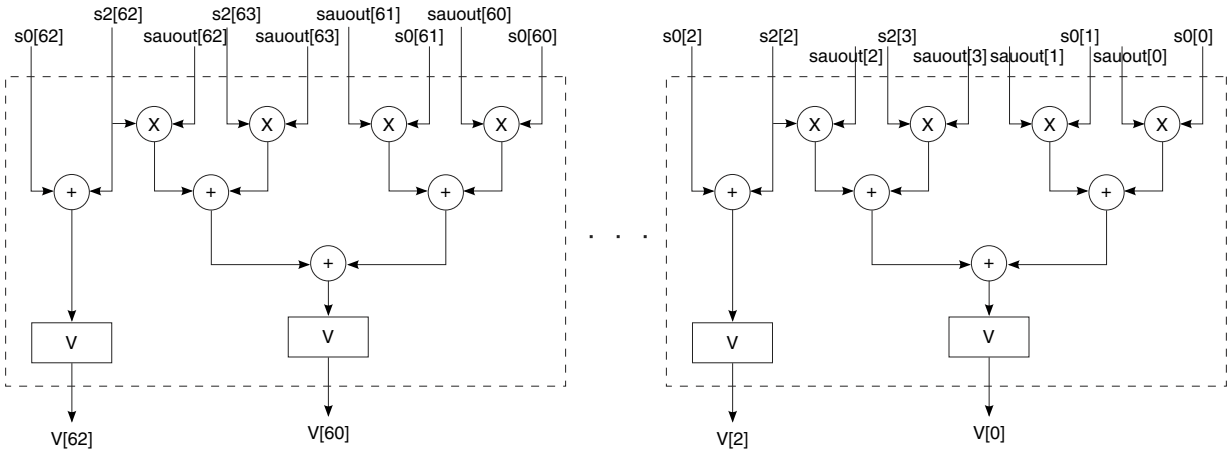



**Figure 35. dif Operation (Single Precision)**

### 4.13.7.1 FFT DIF code examples

The following examples are given as sample code for three differenct precisions SP, HF, and HP, respectively. The element size in these examples is 128.

**FFT DIF for SP**

```
           .global  _fft_DIF_128_single
   .type   _fft_DIF_128_single, @function
_fft_DIF_128_single:

set.prec single, single, single, single, single;
// set precision before set.nco

jmp core_fft_DIF_128_single;

set.nco radix2, 1, 33554432;                        // 128

setB.creg 22, 2;
//set 16 bits addressing mode (VSPA3)

   .size   _fft_DIF_128_single, .-_fft_DIF_128_single

//
================================================================================
===
   .global  _ifft_DIF_128_single
   .type   _ifft_DIF_128_single, @function
_ifft_DIF_128_single:
set.prec single, single, single, single, single;
// set precision before set.nco

jmp core_fft_DIF_128_single;

set.nco radix2, 1, -33554432;                        // 128

setB.creg 22, 2;
//set 16 bits addressing mode (VSPA3)


.size   _ifft_DIF_128_single, .-_ifft_DIF_128_single


// ***************************** CORE FUNCTION ******************************
```

```
.type   core_fft_DIF_128_single, @function
core_fft_DIF_128_single:

set.creg 25, 0;                          // use sau NCO

set.range a1, a2, g0;                   // circular input

ld.laddr
[a1]+1;
clr.VRA;

ld.laddr
[a1]+3;
set.VRAincr rS0, 32;

ld.laddr
[a1]+1;

ld.laddr [a1]-3;
ld.h2l     R0;
set.Smode S0fftn, S1nco, S2fftn;

ld.laddr [a1]+4;
ld.l2h_h2l R0;
setB.creg 15, 0xB; // for FFT mode

ld.laddr [a1]-4;
ld.h2l     R4;
setB.VRAptr rS0, 256;

ld.laddr [a1]+1;
ld.l2h_h2l R4;
set.VRAincr rV, 32;

ld.laddr [a1]+3;
ld.l2h     R1;
set.VRAincr rS2, 32;

ld.laddr [a1]+1;
ld.l2h     R5;

ld.laddr [a1]-3;
ld.h2l     R2;
rd s0;
nco;
rd s2;

ld.laddr [a1]+4;
ld.l2h_h2l R2;
rd s0;
nco; rd s2;

ld.laddr [a1]+1;
ld.h2l     R6;
rd s0;
nco;
rd s2;
dif.sau;

ld.l2h_h2l R6;
rd s0;
nco;
rd s2;
dif.sau;

ld.l2h     R3;
rd s0;
nco;
```

```
rd s2;
dif.sau;
set.VRAincr rSt, 1;

ld.l2h     R7;
rd s0;
nco;
rd s2;
dif.sau;

rd s0;
nco;
rd s2;
dif.sau;
wr.fftn;
mv nco_k, 2;

setA.VRAptr rS2, 0;
rd s0; nco;
rd s2;
dif.sau;
wr.fftn;
setB.VRAptr rV, 128;

setA.VRAptr rS0, 256;
dif.sau;
wr.fftn;

setA.VRAptr rV, 256;
dif.sau;
wr.fftn;


wr.fftn;
rd S0;
nco;
rd S2;
wr.fftn;
setB.VRAptr rV, 384;

setA.VRAptr rS2, 128;
rd S0;
nco;
rd S2;
wr.fftn;
setB.VRAptr rS0, 384;
rd S0;
nco;
rd S2;
dif.sau;
wr.fftn;
setB.VRAptr rV, 0;
setA.VRAptr rS2, 64;
rd S0;
nco;
rd S2;
dif.sau;
setB.VRAptr rS0, 320;
rd S0;
nco;
rd S2;
dif.sau;
setA.VRAptr rS2, 192;
rd S0;
nco;
rd S2;
dif.sau;
setB.VRAptr rS0, 448;
rd S0;
```

```
nco;
rd S2;
dif.sau;
wr.fftn;
mv nco_k, 4;
setA.VRAptr rS2, 0;
rd S0;
nco;
rd S2;
dif.sau;
wr.fftn;
setB.VRAptr rV, 128;
dif.sau;
wr.fftn;
setB.VRAptr rS0, 128;
rd S0;
nco;
rd S2;
dif.sau;
wr.fftn;
setB.VRAptr rV, 256;
rd S0;
nco;
rd S2;
wr.fftn;
 rd S0;
nco;
rd S2;
dif.sau;
wr.fftn;
setB.VRAptr rV, 384;

setA.VRAptr rS2, 256;
rd S0;
nco;
rd S2;
dif.sau;
wr.fftn;
setB.VRAptr rS0, 384;
rd S0;
nco;
rd S2;
dif.sau;
wr.fftn;
setB.VRAptr rV, 0;
rd S0;
nco;
rd S2;
dif.sau;
mv nco_k, 8;
rd S0;
nco;
rd S2;
dif.sau;
wr.fftn;
setA.VRAptr rS2, 0;
rd S0;
nco;
rd S2;
dif.sau;
wr.fftn;
setB.VRAptr rV, 128;

setA.VRAptr rS0, 32;
dif.sau;
wr.fftn;
set.VRAincr rS0, 64;

setA.VRAptr rV, 256;
```

```
dif.sau;
wr.fftn;
set.VRAincr rS2, 64;
rd S0;
nco;
rd S2;
wr.fftn;
rd S0;
nco;
rd S2;
wr.fftn;
setB.VRAptr rV, 384;
rd S0;
nco;
rd S2;
dif.sau;
wr.fftn;
set.creg 15, 0xC;

setA.VRAptr rV, 0;
rd S0;
nco;
rd S2;
dif.sau;
wr.fftn;
set.VRAincr rV, 64;
rd S0;
nco;
rd S2;
dif.sau;
rd S0;
nco;
rd S2;
dif.sau;
mv nco_k, 16;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft4;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft4;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft4;
set.Smode S0fft3, S2fft3;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft4;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft4;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft4;
rd S0;
nco;
rd S2;
```

```
dif.sau;
wr.fft4;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft4;
mv nco_k, 32;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft3;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft3;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft3;
set.Smode S0fft2, S2fft2;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft3;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft3;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft3;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft3;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft3;
mv nco_k, 64;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft2;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft2;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft2;
set.Smode S0fft1, S2fft1;
rd S0;
nco;
rd S2;
```

```
dif.sau;
wr.fft2;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft2;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft2;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft2;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft2;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft1;
st.laddr [a0]+1;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft1;
st.laddr [a0]+1;
dif.sau;
wr.fft1;
st.laddr [a0]+1;
dif.sau;
wr.fft1;

st.laddr [a0]+1;
wr.fft1;
st.laddr
[a0]+1;
wr.fft1;
st.laddr
[a0]+1;
wr.fft1;
st.laddr
[a0]+1;
wr.fft1;

st.laddr
[a0]+1;


mv a4, 0;

set.range a1, a4, 0;
rts;
set.creg  15, 0xA; // (reset to default value)

nop;

.size   core_fft_DIF_128_single, .-core_fft_DIF_128_single
```

**FFT DIF for HF**

```
 .global  _fft_DIF_128_hfx
.type   _fft_DIF_128_hfx, @function
_fft_DIF_128_hfx:

set.prec half_fixed, single, half_fixed, f24, half_fixed;
// set precision before set.nco

jmp core_fft_DIF_128_hfx;

set.nco radix2, 1, 33554432;                                    // 128

set.creg 22, 2;                     // set 16 bits addressing mode (VSPA3)


.size   _fft_DIF_128_hfx, .-_fft_DIF_128_hfx

//
================================================================================
===
   .global  _ifft_DIF_128_hfx
   .type   _ifft_DIF_128_hfx, @function
_ifft_DIF_128_hfx:
set.prec half_fixed, single, half_fixed, f24, half_fixed;
// set precision before set.nco

jmp core_fft_DIF_128_hfx;

set.nco radix2, 1, -33554432;
// 128

set.creg 22, 2;
// set 16 bits addressing mode (VSPA3)


.size   _ifft_DIF_128_hfx, .-_ifft_DIF_128_hfx


// ***************************** CORE FUNCTION ******************************

.type   core_fft_DIF_128_hfx, @function
core_fft_DIF_128_hfx:

set.creg 25, 0;                         // use sau NCO

set.range a1, a2, g0;         // circular input

set.creg 13, 1;                       // half fixed scale by 1/2

setB.creg 15, 0xB;                      // for FFT mode

set.VRArange1 rSt, 0, 7;              // for st

ld.laddr
[a1]+1;
set.Smode S0fftn, s1nco, S2fftn;

ld.laddr
[a1]+1;
set.VRAincr rS0, 32;                  // 128 FFT index

ld.laddr
[a1]+1;
setB.VRAptr rS2, 0;
```

```
ld.laddr [a1]-1;
ld.h2l     R0;
setB.VRAptr rS0, 128;

ld.laddr [a1]+2;
ld.l2h_h2l R0;
setB.VRAptr rV, 0;

ld.laddr [a1]+0;
ld.h2l     R2;
set.VRAincr rV, 32;

ld.l2h_h2l R2;
set.VRAincr rS2, 32;

ld.l2h     R1;
rd s0;
nco;
rd s2;
set.VRAincr rSt, 1;

ld.l2h     R3;
rd s0;
nco;
rd s2;
setB.VRAptr rSt, 0;

rd s0;
nco;
rd s2;
dif.sau;
mv nco_k, 2;

setA.VRAptr rS2, 0;
rd s0;
nco;
rd s2;
dif.sau;
setB.VRAptr rS0, 128;


dif.sau;
dif.sau;


wr.fftn;
wr.fftn;
setB.VRAptr rV, 128;
wr.fftn;
rd S0;
nco;
rd S2;
wr.fftn;
rd S0;
nco;
rd S2;
rd S0;
nco;
rd S2;
dif.sau;
rd S0;
nco;
rd S2;
dif.sau;
dif.sau;
setB.VRAptr rV, 0;
dif.sau;
mv nco_k, 4;
wr.fftn;
```

```
set.VRAincr rS2, 64;                // 2*64 FFT done

setA.VRAptr rS2,
0;
wr.fftn;
setB.VRAptr rV, 128;

setA.VRAptr rS0,
32;
wr.fftn;
set.VRAincr rS0, 64;
rd S0;
nco;
rd S2;
wr.fftn;
rd S0;
nco;
rd S2;
rd S0;
nco;
rd S2;
dif.sau;
set.creg 15, 0xC;

setA.VRAptr rV, 0;
rd S0;
nco;
rd S2;
dif.sau;
set.VRAincr rV, 64;
dif.sau;
mv nco_k, 8;

setA.VRAptr rS2, 0;
dif.sau;
setB.VRAptr rS0, 0;
wr.fft5;
rd S0;
nco;
rd S2;
wr.fft5;
set.Smode S0fft4, S2fft4;
rd S0;
nco;
rd S2;
wr.fft5;
mv nco_k, 16;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft5;

setA.VRAptr rS2, 0;
rd S0;
nco;
rd S2;
dif.sau;
setB.VRAptr rS0, 0;
dif.sau;

setA.VRAptr rV, 0;
dif.sau;
wr.fft4;
rd S0;
nco;
rd S2;
wr.fft4;
set.Smode S0fft3, S2fft3;
```

```
rd S0;
nco;
rd S2;
wr.fft4;
mv nco_k, 32;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft4;

setA.VRAptr rS2, 0;
rd S0;
nco;
rd S2;
dif.sau;
setB.VRAptr rS0, 0;
dif.sau;

setA.VRAptr rV, 0;
dif.sau;
wr.fft3;
rd S0;
nco;
rd S2;
wr.fft3;
set.Smode S0fft2, S2fft2;
rd S0;
nco;
rd S2;
wr.fft3;
mv nco_k, 64;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft3;

setA.VRAptr rS2, 0;
rd S0;
nco;
rd S2;
dif.sau;
setB.VRAptr rS0, 0;
dif.sau;

setA.VRAptr rV, 0;
dif.sau;
wr.fft2;
rd S0;
nco;
rd S2;
wr.fft2;
set.Smode S0fft1, S2fft1;
rd S0;
nco;
rd S2;
wr.fft2;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft2;
rd S0;
nco;
rd S2;
dif.sau;
dif.sau;
```

```
setA.VRAptr rV, 0;
dif.sau;
wr.fft1;
wr.fft1;

st.laddr
[a0]+1;

wr.fft1;

st.laddr
[a0]+1;

wr.fft1;
st.laddr [a0]+1;
st.laddr
[a0]+1;

mv a4, 0;
set.range a1, a4, 0;
// half fixed scale by 1/2 -> off
rts;
set.creg  15, 0xA;             // (reset to default value)
nop;
.size   core_fft_DIF_128_hfx, .-core_fft_DIF_128_hfx
```

**FFT DIF for HP**

```
           .global  _fft_DIF_128_hflt
   .type   _fft_DIF_128_hflt, @function
_fft_DIF_128_hflt:

set.prec half, single, half, f24, half;
// set precision before set.nco

jmp core_fft_DIF_128_hflt;

set.nco radix2, 1, 33554432;                                    // 128

set.creg 22, 2;
// set 16 bits addressing mode (VSPA3)

   .size   _fft_DIF_128_hflt, .-_fft_DIF_128_hflt

//
================================================================================
===
   .global  _ifft_DIF_128_hflt
   .type   _ifft_DIF_128_hflt, @function
_ifft_DIF_128_hflt:

set.prec half, single, half, f24, half;
// set precision before set.nco

jmp core_fft_DIF_128_hflt;

set.nco radix2, 1, -33554432;
// 128

set.creg 22, 2;
// set 16 bits addressing mode (VSPA3)


.size   _ifft_DIF_128_hflt, .-_ifft_DIF_128_hflt


// ***************************** CORE FUNCTION *****************************
```

```
   .type   core_fft_DIF_128_hflt, @function
core_fft_DIF_128_hflt:

set.creg 25, 0;
// use sau NCO

set.range a1, a2, g0;
// circular input

setB.creg 15, 0xB;
// for FFT mode

set.VRArange1 rSt, 0, 7;
// for st

ld.laddr
[a1]+1;
set.Smode S0fftn, S1nco, S2fftn;

ld.laddr
[a1]+1;
set.VRAincr rS0, 32; // 128 FFT index

ld.laddr
[a1]+1;
setB.VRAptr rS2, 0;

ld.laddr [a1]-1;
ld.h2l     R0;
setB.VRAptr rS0, 128;

ld.laddr [a1]+2;
ld.l2h_h2l R0;
setB.VRAptr rV, 0;

ld.laddr [a1]+0;
ld.h2l     R2;
set.VRAincr rV, 32;

ld.l2h_h2l R2;
set.VRAincr rS2, 32;

ld.l2h     R1;
rd s0;
nco;
rd s2;
set.VRAincr rSt, 1;

ld.l2h     R3;
rd s0;
nco;
rd s2;
setB.VRAptr rSt, 0;
rd s0;
nco;
rd s2;
dif.sau;
mv nco_k, 2;

setA.VRAptr rS2, 0;
rd s0;
nco;
rd s2;
dif.sau;
setB.VRAptr rS0, 128;


dif.sau;
dif.sau;
```

```
wr.fftn;
wr.fftn;
setB.VRAptr rV, 128;
wr.fftn;
rd S0;
nco;
rd S2;
wr.fftn;
rd S0;
nco;
rd S2;
rd S0;
nco;
rd S2;
dif.sau;
rd S0;
nco;
rd S2;
dif.sau;
dif.sau;
setB.VRAptr rV, 0;
dif.sau;
mv nco_k, 4;
wr.fftn;
set.VRAincr rS2, 64;// 2*64 FFT done
setA.VRAptr rS2,
0;
wr.fftn;
setB.VRAptr rV, 128;
setA.VRAptr rS0,
32;
wr.fftn;
set.VRAincr rS0, 64;
rd S0;
nco;
rd S2;
wr.fftn;
rd S0;
nco;
rd S2;
rd S0;
nco;
rd S2;
dif.sau;
set.creg 15, 0xC;
setA.VRAptr rV, 0;
rd S0;
nco;
rd S2;
dif.sau;
set.VRAincr rV, 64;
dif.sau;
mv nco_k, 8;
setA.VRAptr rS2, 0;
dif.sau;
setB.VRAptr rS0, 0;
wr.fft5;
rd S0;
nco;
rd S2;
wr.fft5;
set.Smode S0fft4, S2fft4;
rd S0;
nco;
rd S2;
wr.fft5;
mv nco_k, 16;
```

```
rd S0;
nco;
rd S2;
dif.sau;
wr.fft5;

setA.VRAptr rS2, 0;
rd S0;
nco;
rd S2;
dif.sau;
setB.VRAptr rS0, 0;
dif.sau;

setA.VRAptr rV, 0;
dif.sau;
wr.fft4;
rd S0;
nco;
rd S2;
wr.fft4;
set.Smode S0fft3, S2fft3;
rd S0;
nco;
rd S2;
wr.fft4;
mv nco_k, 32;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft4;
setA.VRAptr rS2, 0;
rd S0;
nco;
rd S2;
dif.sau;
setB.VRAptr rS0, 0;
dif.sau;

setA.VRAptr rV, 0;
dif.sau;


wr.fft3;
rd S0;
nco;
rd S2;
wr.fft3;
set.Smode S0fft2, S2fft2;
rd S0;
nco;
rd S2;
wr.fft3;
mv nco_k, 64;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft3;
setA.VRAptr rS2, 0;
rd S0;
nco;
rd S2;
dif.sau;
setB.VRAptr rS0, 0;
dif.sau;

setA.VRAptr rV, 0;
```

```
dif.sau;
wr.fft2;
rd S0;
nco;
rd S2;
wr.fft2;
set.Smode S0fft1, S2fft1;
rd S0;
nco;
rd S2;
wr.fft2;
rd S0;
nco;
rd S2;
dif.sau;
wr.fft2;
rd S0;
nco;
rd S2;
dif.sau;
dif.sau;

setA.VRAptr rV, 0;
dif.sau;
wr.fft1;
wr.fft1;

st.laddr
[a0]+1;

wr.fft1;

st.laddr
[a0]+1;

wr.fft1;

st.laddr
[a0]+1;


st.laddr
[a0]+1;


mv a4, 0;

rts;

set.creg  15, 0xA; // (reset to default value)

nop;

.size   core_fft_DIF_128_hflt, .-core_fft_DIF_128_hflt
```

## 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 58. SAU Instructions**

| Mnemonic | Function Performed |
|---|---|
| rcp | V[i*2] = 1/s1[i*2]<br>SAUout[i*2] = 1/s1[i*2]<br>SAUout[i*2+1] = 1/s1[i*2]<br>for i=0,1,..,31<br>input: single precision<br>output: single precision |
| rrt | V[i*2] = 1/√\|s1[i*2]\|<br>SAUout[i*2] = 1/√\|s1[i*2]\|<br>SAUout[i*2+1] = 1/√\|s1[i*2]\|<br>for i=0,1,..,31<br>input: single precision<br>output: single precision |
| srt | V[i*2] = √\|s1[i*2]\|<br>SAUout[i*2] = √\|s1[i*2]\|<br>SAUout[i*2+1] = √\|s1[i*2]\|<br>for i=0,1,..,31<br>input: single precision<br>output: single precision |
| nco | See Numerically controlled oscillator (NCO) instructions |
| padd | V[i]=s2[i]+s1[i]<br>SAUout[i] = s2[i]+s1[i]<br>for i=0,1,...,63<br>input: half-fixed precision<br>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 59. Reciprocal Descriptions**

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

### 4.14.5 Reciprocal square root functionality

**Table 60. Reciprocal Square Root Descriptions**

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

### 4.14.6 Square root functionality

**Table 61. Square Root Descriptions**

| Instruction | AUprec | Operation | Description |
|---|---|---|---|
| srt | Single | for (i=0; i<64; i+=2)<br>V[i] = √\|s1[i]\|;<br>SAUout[i] = √\|s1[i]\|;<br>SAUout[i+1] = √\|s1[i]\|; | Calculate the square root of all real elements of s1.<br>**NOTE**<br>The result of srt is an approximation. |

### 4.14.7 Pre adder functionality

**Table 62. Pre adder Descriptions**

| Instruction | AUprec | Operation | Description |
|---|---|---|---|
| padd | padd | for (i=0; i<64; i+=1)<br>V[i] = s2[i] + s1[i]<br>SAUout[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 in single precision format and 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 | F24/Single | Not applicable | - |
| padd | paddF24 | for (i=0; i<128; i+=1)<br>V[i] = s2[i] + s1[i]<br>SAUout[i] = s2[i] + s1[i] | The same operation is performed as when AUprec is set to padd except the output vector from the pre-adder is in F24 format, the AU operates in F24 mode, and 128 pre-add operations are performed. |

## 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 63 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 63. 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. |
| 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:

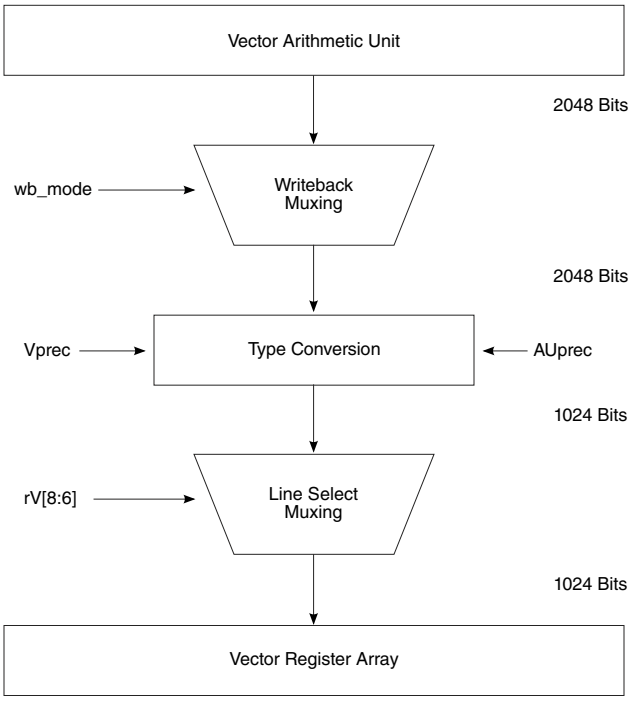



**Figure 36. Single Precision to Half Fixed VRA store**

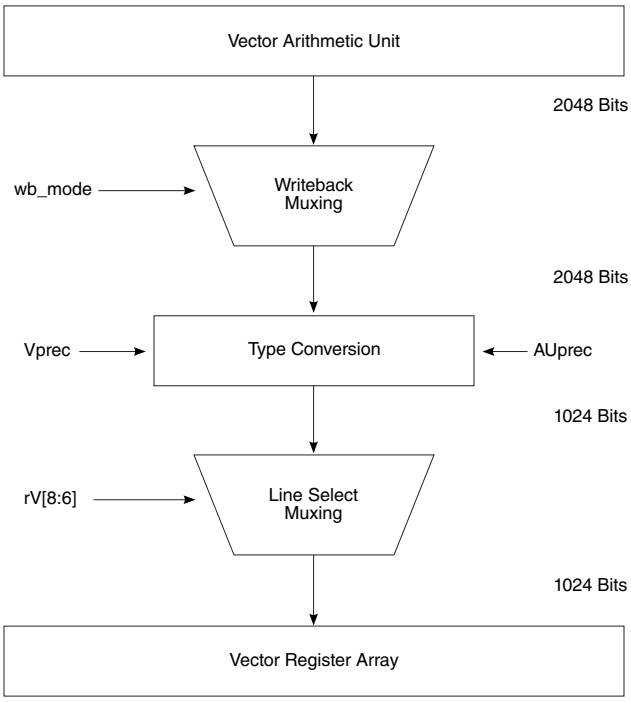



**Figure 37. Single Precision to Half Precision VRA store**

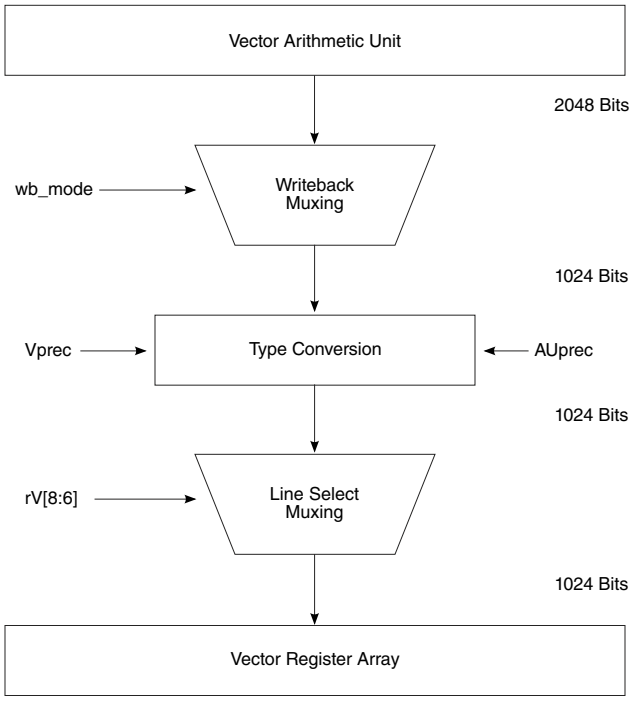



**Figure 38. Single Precision to Single Precision VRA store**

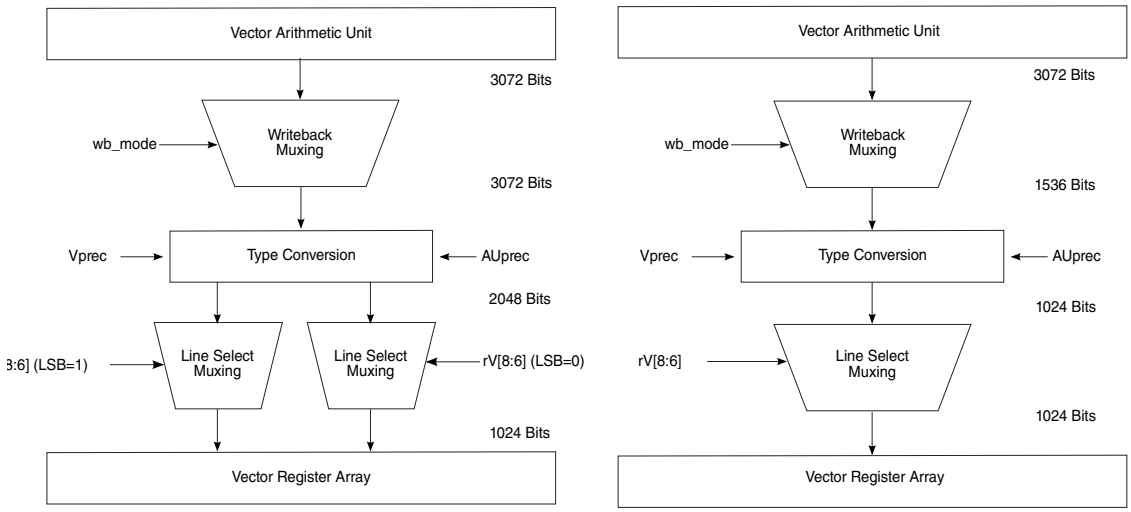



**Figure 39. F24 Precision to Half Fixed VRA store (for real and complex operations respectively)**

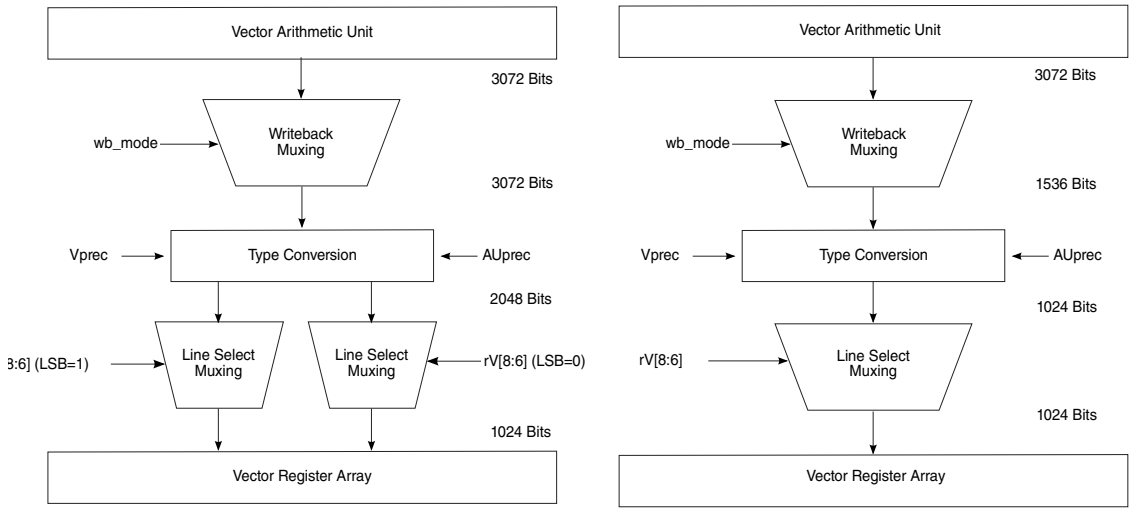



**Figure 40. F24 Precision to Half Precision VRA store (for real and complex operations respectively)**

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 or 128 elements if AUprec is F24.
- 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 23

**Table 64. All Supported Vmode, Write-back Modes**

| Instruction | Restrictions | Operation | Description |
|---|---|---|---|
| wr.fn; | Real &<br>Complex data<br>creg(15[2])=0, creg(25), creg(21), creg(19) | R*x* = VRA [rV[8:6]];<br>for (i=0,j=0;j < 64;j =+2,i++ ) {<br>R*x* [i] = V [j];<br>}<br>update_rag(rV); | Store the vector output of the SAU into the VRA. Only the even elements of V are used.<br>Update rV. |
| wr.fn1; | Real &<br>Complex data<br>creg(15[2])=0, creg(25), creg(21), creg(19) | Rx [rV[5:0]] = V [0];<br>update_rag(rV); | Store the first SAU output into a line of register a VRA register pointed to by rV.<br>Update rV. |
| wr.even; | Real data<br>creg(15[2])=0, creg(19) | R*x* = VRA [rV[8:6]];<br>for (i=0,j=0;j < 128;j +=2,i++ ) {<br>R*x* [i] = V [j]; | Store the output of AU complex operation into the vector register array. Only the even elements of V are used. |

*Table continues on the next page...*

**Table 64. All Supported Vmode, Write-back Modes (continued)**

| Instruction | Restrictions | Operation | Description |
|---|---|---|---|
| | | }<br>update_rag(rV); | Update rV.<br>Note that, wr.even only works for half_fixed or half_float precisions. |
| wr.fftn; | Complex data only,<br>Vprec=half-fixed/half/ single<br>creg(15[2])=0, creg(19) | R*x* = VRA [rV[8:6] & 0x6];<br>R*x*+1 = VRA [rV[8:6] & 0x6 + 1];<br>switch (rV[5:4]) {<br>// store to Q0 of R*x*, R*x*+1<br>0: i_list = {0,...,15};<br>// store to Q1 of R*x*, R*x*+1<br>1: i_list = {16,...,31};<br>// store to Q2 of R*x*, R*x*+1<br>2: i_list = {32,...,47};<br>// store to Q3 of R*x*, R*x*+1<br>3: i_list = {48,...,63};<br>j_list = {2,6,...,58,62};<br>foreach (i, j) in ( i_list , j_list ) {<br>R*x* [ i ] = V [ j ];<br>R*x*+1[ i ] = V [ j - 2 ];<br>}<br>update_rag(rV); | Store the output of AU fft operation into vector register array.<br>This mode is intended for DIF FFT stage 8 and for DIT FFT stage *N*-7.<br>Only supports an even register as the starting register. That is, R*x* = R0, R2, R4, or R6<br>R*x* is the output for the top leg of butterfly, while R*x*+1 is the output of the bottom one.<br>Update rV. |
| wr.fft5; | Complex data only,<br>Vprec=half-fixed/half<br>creg(15[2])=1, creg(19) | R*x* = VRA [rV[8:6]];<br>if (rV[5]== 0)<br>i_list = {0-31}<br>else<br>i_list = {32-63}<br>j_list = {2,6,...,58,62};<br>foreach (i, j) in ( i_list , j_list ) {<br>R*x* [ i ] = V [ j ];<br>R*x* [ i + 32 ] = V [ j - 2 ];<br>}<br>update_rag(rV); | Store the output of AU fft operation into the vector register array.<br>This mode is intended for DIF FFT stage 5 and for DIT FFT stage *N*-4.<br>Update rV. |
| wr.fft4; | Complex data only,<br>Vprec=half-fixed/half/ single | R*x* = VRA [rV[8:6]];<br>if (rV[5]== 0) | Store the output of AU fft operation into the vector register array. |

*Table continues on the next page...*

**Table 64. All Supported Vmode, Write-back Modes (continued)**

| Instruction | Restrictions | Operation | Description |
|---|---|---|---|
| | creg(15[2])=1, creg(19) | i_list = {0-15,..., 16-31}<br>else<br>i_list = {32-47,..., 48-63}<br>j_list = {2,6,...,58,62};<br>foreach (i, j) in ( i_list , j_list ) {<br>R*x* [ i ] = V [ j ];<br>R*x* [ i + 16 ] = V [ j - 2 ];<br>}<br>update_rag(rV); | This mode is intended for DIF FFT stage 4 and for DIT FFT stage *N*-3.<br>Update rV. |
| wr.fft3; | Complex data only,<br>Vprec=half-fixed/half/ single<br>creg(15[2])=1, creg(19) | R*x* = VRA [rV[8:6]];<br>if (rV[5]== 0)<br>i_list = {0-7,..., 24-31}<br>else<br>i_list = {32-39,..., 56-63}<br>j_list = {2,6,...,58,62};<br>foreach (i, j) in ( i_list , j_list ) {<br>R*x* [ i ] = V [ j ];<br>R*x* [ i + 8 ] = V [ j - 2 ];<br>}<br>update_rag(rV); | Store the output of AU fft operation into the vector register array.<br>This mode is intended for DIF FFT stage 3 and for DIT FFT stage *N*-2.<br>Update rV. |
| wr.fft2; | Complex data only,<br>Vprec=half-fixed/half/ single<br>creg(15[2])=1, creg(19) | R*x* = VRA [rV[8:6]];<br>if (rV[5]== 0)<br>i_list = {0-3,..., 28-31}<br>else<br>i_list = {32-35,..., 60-63}<br>j_list = {2,6,...,58,62};<br>foreach (i, j) in ( i_list , j_list ) {<br>R*x* [ i ] = V [ j ];<br>R*x* [ i + 4 ] = V [ j - 2 ];<br>}<br>update_rag(rV); | Store the output of AU fft operation into the vector register array.<br>This mode is intended for DIF FFT stage 2 and for DIT FFT stage *N*-1.<br>Update rV. |
| wr.fft1; | Complex data only, | R*x* = VRA [rV[8:6]];<br>if (rV[5]== 0) | Store the output of AU fft operation into the vector register array. |

*Table continues on the next page...*

**Table 64. All Supported Vmode, Write-back Modes (continued)**

| Instruction | Restrictions | Operation | Description |
|---|---|---|---|
| | Vprec=half-fixed/half/single<br>creg(15[2])=1, creg(19) | i_list = {0-1,..., 30-31}<br>else<br>i_list = {32-33,..., 62-63}<br>j_list = {2,6,...,58,62};<br>foreach (i, j) in ( i_list , j_list ) {<br>Rx [ i ] = V [ j ];<br>Rx [ i + 2 ] = V [ j - 2 ];<br>}<br>update_rag(rV); | This mode is intended for DIF FFT stage 1 and for DIT FFT stage *N*.<br>Update rV. |
| wr.straight | Real data<br>AUprec=single<br>creg(15[2])=0, creg(19) | Rx = VRA [rV[8:6]];<br>for (i=0;i<64; i++) {<br>Rx [i] = V [i];<br>}<br>update_rag(rV); | Store the output of AU real operation into the vector register array.<br>Update rV. |
| | Real data<br>AUprec=F24<br>creg(15[2])=0, creg(19) | Rx = RF [2*floor(rV[8:6]/2)];<br>for (i=0;i<64; i++) {<br>Rx [i] = V [i];<br>}<br><br>Rx = RF [2*floor(rV[8:6]/2)+1];<br>for (i=64;i<128; i++) {<br>Rx [i] = V [i];<br>} | |
| wr.hlinecplx | Complex data<br>AUprec=single<br>creg(15[2])=x, creg(19) | Rx = VRA [rV[8:6]];<br>for (i=0,j=0;j <64;j =+2,i++ ) {<br>Rx [i] = V [j];<br>}<br>update_rag(rV); | Store the output of AU complex operation into vector register array. Only the even elements of V are used.<br>Update rV. |
| | Complex data<br>AUprec=F24<br>creg(15[2])=x, creg(19) | Rx = VRA [rV[8:6]];<br>for (i=0,j=0;j <128;j =+2,i++ ) {<br>Rx [i] = V [j];<br>}<br>update_rag(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 Vector predication

The VSPA vector predication system provides resources to simplify and speed up conditional element by element operations on vector data. It is tightly coupled with the vector AU subsystem, and is only intended for use with real numbers.

The main purpose of the predication features is to perform alternative processing on selected elements within a vector. For example, take all elements of a vector that are negative and set them to zero:

```
for (int ii=0; ii < vecsize; ii++) if (vec[ii] < 0) vec[ii] = 0;
```

The following resources are available to accelerate these sorts of operations:

- 4 Vector Predicate (VP) registers, each wide enough to represent every possible element of a vector, when the elements are stored in a 16-bit precision format (1/8 the width of an R register)
- self-incrementing read and write pointers for the VP registers
- per-element condition detection logic at the outputs of the AUs - can be captured into VP registers
- VP register controlled, per-element S1 and S2 data modification at the AU inputs
- AU instruction extensions that optionally capture AU output conditions zero or negative, or both
- AU instruction extensions that optionally allow VP register controlled S1 and S2 data modification at the AU inputs
- load and store multiple VP register instructions for fast data memory transactions
- Basic instructions to manipulate VP registers, including AND, OR, XOR, COM (multiple), and CLR (multiple)

As previously mentioned, the vector predication can provide alterative processing for selected elements of a vector. This happens at the same time as a non-alternative instruction is performed on the non-selected elements of the vector. Here's how it works.

First, program the VP_CONTROL register to set up predicate alternate action and attributes. One mvip instruction can configure the following:

- Set the Action Pointer (ACT_PTR) field to choose which VP register will control the predicate action
- Set the Action Mode (ACT_MODE) field to choose the alternate predicate action
- Set the Action Increment (ACT_INCR) field to choose the post-increment of the action pointer

Subsequently, execute one of the following vector AU instructions:

- rmad.[vpz|vpnz]
- rmac.[vpz|vpnz]
- rmad.sau.[vpz|vpnz]
- rmac.sau.[vpz|vpnz]
- maf.[vpz|vpnz]

These are multiply-add, multiply-add with SAU, multiply-add and accumulate, and multiply-add with feedback instructions, modified with a conditional suffix of either .vpz or .vpnz.

The suffix .vpz means "execute primary action if vector predicate is zero, else execute alternate action".

The suffix .vpzn means "execute primary action if vector predicate is NOT zero, else execute alternate action".

When one of these vector AU instructions is executed, the selected VP register's bits will control which vector elements will experience the primary action, and which will experience the chosen alternate action.

The alternate actions are accomplished by modifying the vector AU operand inputs. The possible modifications are:

- replace S1 or S2 with 0
- replace S1 or S2 with +1
- replace S1 or S2 with -1
- change the sign of S2

Not every combination of these modifications are available, just the following variations.

Mathematic manipulations

- replace S2 with 0, that is, deactivate the addition feature, that is, make the AU output into S0 * S1.
- replace S1 with +1, that is, deactivate the multiply feature, that is, make the AU output into S0 + S2.
- replace S1 with +1, and change the sign of S2 (only for rmad), that is, make the AU output into S0 - S2.

Constant outputs

- replace S1 with 0, and S2 with 0, that is, make the AU output 0.
- replace S1 with 0, and S2 with +1, that is, make the AU output +1.
- replace S1 with 0, and S2 with -1, that is, make the AU output -1.

Operand forwarding

- replace S1 with +1, and S2 with 0, that is, forward the S0 operand to the AU output.
- replace S1 with 0, that is, forward the S2 operand to the AU output.
- replace S1 with -1, and S2 with 0, that is, forward the -S0 operand to the AU output.
- replace S1 with 0 and change the sign of S2 (only for rmad and maf), that is, forward the -S2 operand to the AU output.

While the previous section described some of the ways predicate information can be used, VSPA's vector predication system also allows the creation of the predication information. This can happen in several ways:

- the VP registers can capture information that is created as a result of a vector AU operation
- the VP registers can be loaded from and stored to VSPA DMEM
- the VP registers can be manipulated and combined via basic logical operation instructions

A very important method of predicate creation is capture. Predicate capture occurs upon the completion of a vector AU operation. The vector AU outputs produce two condition bits for each result element, a Zero (VZ) bit and a Negative (VN) bit. As you might guess, the VZ bit is set when the result is zero, and cleared otherwise. The VN bit is set when the result is negative, and cleared otherwise. Depending on how the VP_CONTROL register is configured, you can capture either the VZ or the VN bit in a VP register, or both the VZ and VN bits in a pair of VP registers.

Predicate capture from the AU result outputs will occur upon execution of one of the following vector AU instructions:

- rmad.uvp
- rmac.uvp
- rmad.sau.uvp
- rmac.sau.uvp
- maf.uvp

The directive to update vector predicate is denoted by the .uvp suffix. Whenever one of these instructions is executed, the vector predicate registers will be updated in the same cycle as the AU outputs are updated.

The VP_CONTROL register is used to set up predicate capture mode, register pointer, and increment mode. One mvip instruction can configure the following:

- Set the Capture Pointer (CAP_PTR) field to choose which VP register(s) will capture the predicate(s)

- Set the Capture Mode (CAP_MODE) field to choose whether to capture the VZ bits, the VN bits, both VZ and VN bits, or nothing
- Set the Capture Increment (CAP_INCR) field to choose whether the VP register pointer will post-increment following a predicate capture

Another method of predicate creation is to create predication flags directly in DMEM. This can be done with st instructions or with assembler or compiler directives. Once in memory, the ldm VPmask(Iu4),[agY-Iu2] instruction can be used to copy them into VP registers. This instruction can load all 4 VP registers at once, or just one or two VP registers at once.

The VP registers can also be manipulated or used by the following instructions:

- mv VPx(Iu1),h
- mv h,VPx(Iu1)
- ldm VPmask(Iu4),[agY-Iu2]
- stm [agY]+(Iu2),VPmask(Iu4)
- and VpZ,VpX,VpY
- or VpZ,VpX,VpY
- xor VpZ,VpX,VpY
- com VPmask(Iu4)
- clrm VPmask(Iu4)

The first two instructions allow pairs of VP registers to be loaded from or stored to the H register. The next two instructions allow one or more VP registers to be loaded from or stored to DMEM. The remaining instructions allow simple boolean manipulations of the VP registers.

A simple example, and typical use of vector predicate is to compare two vectors (Va and Vb), and determine element by element which is largest. This can be done quite simply.

1. mvip VP_CONTROL, (1<<CAP_INCR)+(1<<CAP_MODE); // set up the capture. Start with VP0, capture VN, and auto-increment
2. setup the S-mux read modes to allow an rmad.uvp instruction to multipy Vb by -1, and load Va normally.
3. execute a rmad.uvp

When the rmad.uvp instruction's AU output becomes valid, VP0 (captured VN) also becomes valid. If the input vectors are too large for the AU to subtract in a single rmad, then up to three additional rmad.uvp instructions can be pipelined in the following cycles, and the VN results from those instructions will be captured in VP1, VP2, and VP3. If even more rmad.uvp instructions are needed to fully compare the vectors, then the four VP registers can be simultaneously written to DMEM after they are valid. This sequence can be continued until the vector subtraction is complete, after which, DMEM will contain a bit map of the sign bits resulting from the vector subtraction.

The VP register capture/storage format associates one VP register bit with each element of an AU result. Bit 0 of the all VP registers is associated with AU[0] result 0, and VP registers' bit 1 are associated with AU[0] result 1. If the AU precision (VPREC) is configured as single (32-bit results), each AU will produce 2 results per AU operation, and there are 2 real VP bits associated with each AU. In that case, VP registers' bit 2 will be associated with AU[1] result 0, and VP registers' bit 3 will be associated with AU[1] result 1. This mapping pattern repeats across all the AUs.

If the AU precision (VPREC) is configured as f24 (16-bit results), each AU will produce 4 real results per AU operation, and there are 4 real VP bits associated with each AU. In that case, VP registers' bit 2 will be associated with AU[0] result 2, and VP registers' bit 3 will be associated with AU[0] result 3. VP registers' bit 4 will map to AU[1] result 0. This mapping pattern repeats across all the AUs.

## 4.17 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.

### 4.17.1 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,

#### 4.17.1.1 Move vector register array instructions

Table 65 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 65. 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]<br>update_rag(rS0); | Move a 16-bit real element from VRA, pointed to by rS0, to g*X*.<br>rS0 is then updated. |
| | mv.w g*X*, [rS0]; | g*X*[15:0] <- [rS0]<br>g*X*[31:16] <- [rS0+1]<br>update_rag(rS0); | Move a 32-bit complex element from VRA, pointed to by rS0, to g*X*.<br>rS0 is then updated. |
| | mv.h [rV], g*X*; | [rV] <- g*X*[15:0];<br>update_rag(rV); | Move the lower 16 bits of g*X* to an element in Rx, pointed to by rV.<br>rV is then updated. |
| | mv.w [rV], g*X*; | [rV] <- g*X*[15:0];<br>[rV+1] <- g*X*[31:16];<br>update_rag(rV); | Move the 32 bit contents of g*X* to an element in Rx, pointed to by rV.<br>rV is then updated. |

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

#### 4.17.1.2 Move hardware register instructions

Table 66 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 66. 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*<br>mv g*X*, nco_k |

*Table continues on the next page...*

**Table 66. Hardware Registers Accessible by *mv* Instruction (continued)**

| Hardware Register <hw_reg> | Description | Related Blocks | No of Bits | Readable | Writeable | Related Instructions |
|---|---|---|---|---|---|---|
| nco_phase | NCO phase register | | 16 | Yes | Yes | mv nco_phase, g*X*<br>mv g*X*, nco_phase |
| nco_freq | NCO freq register | | 32 | No | Yes | mv nco_freq, g*X* |
| gold_x1 | LTE gold sequence first LFSR register | CGU | 31 | Yes | Yes | mv cgu_reg, g*X*<br>mv g*X*, cgu_reg |
| gold_x2 | LTE gold sequence second LFSR register | | 31 | Yes | Yes | |
| ul_sc_x | Long-sequence uplink scrambling first LFSF register | | 25 | Yes | Yes | |
| ul_sc_y | Long-sequence uplink scrambling first LFSF register | | 25 | Yes | Yes | |
| ul_sc_short | Short-sequence uplink scrambling LFSF registers (concatenated) | | 32 | Yes | Yes | |
| dl_sc_x | Downlink scrambling first LFSR register | | 18 | Yes | Yes | |
| dl_sc_y | Downlink scrambling second LFSR register | | 18 | Yes | Yes | |
| ovsf_num | OVSF code number | | 8 | No | Yes | |
| ovsf_dl_sc_y[1] | Concatenation of ovsf_num and dl_sc_y register,<br>dl_sc_y is located at bits 17:0<br>ovsf_num is located at bits 27:20 | | 28 | No | Yes | |
| ovsf_dl_sc_y_bak[2] | A backed up copy of ovsf_num and dl_sc_y register (concatenated the same way as ovsf_dl_sc_y) | | 28 | Yes | No | |
| dl_sc_x_bak[3] | A backed up copy of dl_sc_x register | | 18 | Yes | No | |
| vd_ts_d, vd_ts | Shift register | CGU | 23 | Yes | Yes | |
| vd_qs_d, vd_qs | Shift register | | 11 | Yes | Yes | |

1. ovsf_dl_sc_y is a duplication of ovsf_num and dl_sc_y to allow programming the two register in a single *mv cgu_reg gX*.
2. ovsf_dl_sc_y_bak is updated only when one or more of its primary registers that is, ovsf_num, dl_sc_y, or ovsf_dl_sc_y is written by *mv cgu_reg gX*.

3. dl_sc_x_bak is updated only when its primary register, dl_sc_x, is written by *mv dl_sc_x gX*

#### 4.17.1.3 Move scalar registers

Table 67 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 67. 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 |
| 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<br>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.

#### 4.17.1.4 Move IP instruction

Table 68 shows a list of instructions which move data between the GP registers and the memory mapped IP registers.

**Table 68. 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 |
| | | 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 |

### 4.17.2 Linear feedback shift register instructions

**Table 69. 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:<br>`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:<br>`Iu32[0] ^ Iu32[1] ^ Iu32[2] ^ ...^ Iu32[31]` |

### 4.17.3 Floating point generation instructions

**Table 70. lsb2rf instructions**

| Instruction Syntax | Operation |
|---|---|
| lsb2rf [rV], g*X* | if (g*X*[0] == 0)<br>[rV] = 0; |

*Table continues on the next page...*

**Table 70. lsb2rf instructions (continued)**

| Instruction Syntax | Operation |
|---|---|
| | else<br>[rV] = 0.5;<br>Update rV. |
| lsb2rf.sr [rV], g*X* | if (g*X*[0] == 0)<br>[rV] = 0;<br>else<br>[rV] = 0.5;<br>Update rV.<br>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*.

### 4.17.4 Arithmetic instructions

**Table 71. 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. |

*Table continues on the next page...*

**Table 71. Arithmetic Instructions (continued)**

| Operation | Family | CC[1] | Syntax | Operation | Description |
|---|---|---|---|---|---|
| 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*,Iu22 | aX-Iu22 | Subtract 22 bit integer from a*X* memory pointer and update condition codes accordingly. |
| 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* |

*Table continues on the next page...*

**Table 71. Arithmetic Instructions (continued)**

| Operation | Family | CC[1] | Syntax | Operation | Description |
|---|---|---|---|---|---|
| | | | | | and store the signed result into g*Z*, if optional condition is 'true'. |
| | | | mpyS.z g*X*,lu16 | g*Z*<-g*Z**{16'h0000,lu16} | Multiply contents of g*Z* with Zero extended 16 bit integer and store result back into g*Z*. |
| | | | mpyS.s g*X*,ls16 | g*Z*<-g*Z**{{16{ls[15]}},ls16} | Multiply contents of g*Z* with Sign extended 16 bit integer and store result back into g*Z*. |
| mac[2,3] | opS | creg | mac(.cc)(.s) g*Z* g*X*, g*Y* | g*Z*<-g*X**g*Y* | Integer multiply-accumulate g*Z* = g*Z* + (g*X* * g*Y*)<br>If the optional condition test is 'true', then multiply contents of g*X* with contents of g*Y*, then add the multiply result to g*Z* and store the signed result into g*Z*, if optional condition is 'true'. |
| 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[4,2] | opS | creg | div(.cc)(.s) g*Z* g*X*, g*Y* | g*Z*<-g*X*/g*Y*<br>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*, lu16 | g*Z*<-g*Z*/{16'h0000,lu16}<br>internal<-g*Z*%16'h0000,lu16 | 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*, ls16 | g*Z*<-g*Z*/{{16{ls[15]}},ls16}<br>internal<-g*Z*%{{16{ls[15]}},ls16} | 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[4,2] | opS | creg | rdiv.z g*Z*, lu16 | g*Z*<-{16'h0000,lu16}/g*Z*<br>internal<-{16'h0000,lu16}%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*, ls16 | g*Z*<-{{16{ls[15]}},ls16}/g*Z* | Divide Sign extended 16 bit integer with contents of g*Z* and store signed truncated |

*Table continues on the next page...*

**Table 71. Arithmetic Instructions (continued)**

| Operation | Family | CC[1] | Syntax | Operation | Description |
|---|---|---|---|---|---|
| | | | | internal<-{{16{ls[15]}},ls16}%g*Z* | integer quotient back into g*Z*. In addition, the signed remainder (modulus operation) is available in an internal register. |
| mod[4,2] | opS | creg | mod(.cc)(.s) g*Z* g*X*, g*Y* | g*Z*<-g*X*%g*Y*<br>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*, lu16 | g*Z*<-g*Z*%{16'h0000,lu16}<br>internal<-g*Z*/{16'h0000,lu16} | 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. |
| | | | mod.s g*Z*, ls16 | g*Z*<-g*Z*%{{16{ls[15]}},ls16}<br>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[4,2] | opS | creg | rmod.z g*Z*, lu16 | g*Z*<-{16'h0000,lu16}%g*Z*<br>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*<br>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]) ?<br>~g*X*[31:0] +'h1 :<br>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<br>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*. |

*Table continues on the next page...*

**Table 71. Arithmetic Instructions (continued)**

| Operation | Family | CC[1] | Syntax | Operation | Description |
|---|---|---|---|---|---|
| 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<br>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. Due to hardware resource conflicts with the mpy instruction, mac should not be used in the cycle immediately following an mpy.
4. 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

#### 4.17.4.1 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
```

#### 4.17.4.2 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.fffffffffffffff
1  15 fractional bits
```

#### 4.17.4.3 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.

#### 4.17.4.4 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.

#### 4.17.4.5 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.

#### 4.17.4.6 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.

#### 4.17.4.7 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.

#### 4.17.4.8 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.

## 4.18 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 72. 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.

### 4.18.1 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
```

#### 4.18.1.1 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
```

#### 4.18.1.2 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
```

#### 4.18.1.3 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
```

### 4.18.2 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
```

### 4.18.3 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
```

### 4.18.4 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;
```

### 4.18.5 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;
```

### 4.18.6 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;
```

### 4.18.7 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
```

## 4.19 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 73. 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 76 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).

### 4.19.1 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.

#### 4.19.1.1 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.

### 4.19.2 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)
```

### 4.19.3 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;
```

## 4.20 Conditional instructions

The instructions shown in Table 74 and Table 75 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 76 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 74. Conditional Scalar Instructions**

| Instruction | Type | Description |
|---|---|---|
| mpy*(.cc)*(.s) gZ gX,gY | opS | Multiply |
| div*(.cc)*(.s) gZ gX,gY | opS | Divide |
| 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 75. 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.

### 4.20.1 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 77 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 76. Logical test instruction modifiers**

| Logical Test Modifiers | Descriptions | Operation Type | Condition Code Flags[1] | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | N | Z | V | C | au_ap | au_an | au_az |
| al | Always | Simple | x | x | x | x | x | x | x |
| ap_az | AU all zero | AU OP | x | x | x | x | x | x | 1 |
| ap_naz | AU not all zero | AU OP | x | x | x | x | x | x | 0 |
| au_an[2] | AU all negative | AU OP | x | x | x | x | x | 1 | x |
| au_ap[2] | AU all positive | AU OP | x | x | x | x | 1 | x | x |
| au_nan[2] | AU not all negative | AU OP | x | x | x | x | x | 0 | x |
| au_nap[2] | AU not all positive | AU OP | x | x | x | x | 0 | x | x |
| cc | Carry Clear | Simple | x | x | x | 0 | x | x | x |
| clr | bit clear | Simple | x | 1 | x | x | x | x | x |
| cs | Carry Set | Simple | x | x | x | 1 | x | x | x |
| eq | Equal Zero | Simple | x | 1 | x | x | x | x | x |
| ge | Greater or Equal<br>not (N xor V) | Signed | - | x | - | x | x | x | x |
| gt | Greater<br>not ((N xor V) or Z) | Signed | - | - | - | x | x | x | x |
| hi | Higher not (Z or C) | Unsigned | x | - | x | - | | | x |
| hs | Higher or Same | Unsigned | x | x | x | 0 | x | x | x |

*Table continues on the next page...*

**Table 76. Logical test instruction modifiers (continued)**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| le | Less Than or Equal ((N xor V) or Z) | Signed | - | - | - | x | x | x | x |
| lo | Lower | Unsigned | x | x | x | 1 | x | x | x |
| ls | Lower or Same (Z or C) | Unsigned | x | - | x | - | x | x | x |
| lt | Less Than (N xor V) | Signed | - | x | - | x | x | x | x |
| mi | Minus | Simple | 1 | x | x | x | x | x | x |
| ne | Not Equal | Simple | x | 0 | x | x | x | x | x |
| nv | Never | Simple | x | x | x | x | x | x | x |
| pl | Plus | Simple | 0 | x | x | x | x | x | x |
| set | bit set | Simple | x | 0 | x | x | x | x | x |
| vc | Overflow Clear | Simple | x | x | 0 | x | x | x | x |
| vs | Overflow Set | Simple | x | x | 1 | x | 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.

### 4.20.2 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 77 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 79 for a list of instructions which can update the scalar condition codes when this register is set.

**Table 77. 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 |

*Table continues on the next page...*

**Table 77. VSPA Condition Flags (continued)**

| Condition Flag | Descriptions | Compare Operation Triggered by[1] |
|---|---|---|
| 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 78 will modify the condition code bits regardless of the state of system control register 4.

**Table 78. 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 |

*Table continues on the next page...*

**Table 78. Instructions which always modify condition code flags (continued)**

| Instruction | Type | Description |
|---|---|---|
| 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 79 will modify the condition code bits only if system control register 4 is set to a 1.

**Table 79. 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) |
| 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] |

*Table continues on the next page...*

**Table 79. Instructions which optionally modify condition code flags (continued)**

| Instruction | Type | Description |
|---|---|---|
| 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).

### 4.20.3 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 80 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 80. 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 |

*Table continues on the next page...*

**Table 80. Conditional instructions setup time (continued)**

| | | |
|---|---|---|
| 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.

## 4.21 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)

$$\exp\{j2\pi kf\,(i + nv + [0, 1, 2, ..., v - 1]) / 2^{32}\} \text{ where, } n = 0, 1, 2, ...$$

and produces *v* complex single precision scalars or *v**2 complex 24 bit precision scalars via the mux path described in S1mode options and detailed description. The generator is a function of the parameters described inTable 81.

**Table 81. 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 82.

**Table 82. NCO parameter control instructions**

| Mnemonic | Family | Description |
|---|---|---|
| set.nco {radix2, singles, normal}, Iu10, Is31 | 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.<br>Order of arguments is nco_k, nco_freq.<br>**NOTE**<br>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.<br>**NOTE**<br>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. |
| 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. |

*Table continues on the next page...*

**Table 82. NCO parameter control instructions (continued)**

| Mnemonic | Family | Description |
|---|---|---|
| | | **NOTE**<br>• g*X* is modifed by this instruction.<br>• 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.<br>**NOTE**<br>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;
```

## 4.22 CGU instructions

The Code Generation Unit (CGU) generates special sequences to be used by W-CDMA and LTE applications. It comprises a Scrambling Unit (SC) and an OVSF unit. The SC unit in turn comprises six scrambling subunits:

- Downlink Scrambling subunit
- Long Sequence Uplink Scrambling subunit
- Short Sequence Uplink Scrambling subunit
- LTE Scrambling subunit
- G.fast/VDSL Training Sequence subunit
- G.fast/VDSL Quadratic Scrambling Sequence subunit

Only one of the six scrambling subunits can be active at a time, but the OVSF unit and a scrambling subunit can be active at the same time.

The CGU sends its output control signals to modify the data read from VRA before they are being loaded into S1 register (See S1Mux add-on for CGU and Figure 42). The CGU parameters are programmed through gp.move instruction and the CGU mode of operation is programmed by cgu.set instruction. When a ld.s1 instruction is used with a *S1mode* with a "cgu_" prefix, the current CGU output will modify the data being loaded into S1 and the active CGU unit(s) will update their own states and outputs for the next active cycle.

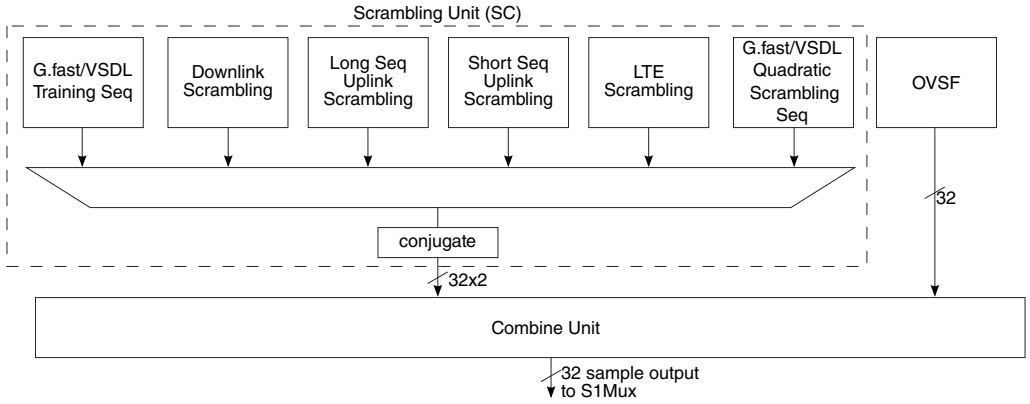



**Figure 41. CGU Block Diagram**

The CGU's mode of operation is set up by set.cgu instruction. This instruction configures three things:

- which of the six scrambling subunits will be active;
- whether the output of the scrambling unit will be conjugated or not;
- whether the OVSF unit will be active or not.

The output of the OVSF unit is real, one bit signal with a "0" representing +1, and a "1" representing -1. When disabled, it always outputs a "0".

The output of LTE scrambling subunit is also real, 2-bit signal with a "00" representing +1, and a "11" representing -1.

The output of all other scrambling subunits are complex, 2-bits signal with LSB for real part (I) and MSB for imaginary (Q). Thus,

- "00" representing 1 + j
- "11" representing -1 - j
- "01" representing -1 + j
- "10" representing 1 - j

The final CGU output is a phase-shifted product of the active scrambling subunit's output and OVSF unit's output, as follows:

$$CGU_{output}(i) = [SC(i)e^{-j\pi/4}] \text{ x } OVSF(i) \text{ where, } i = 0,...,31$$

The term $e^{-j\pi/4}$ does not apply in the case of LTE scrambling.

**Table 83. CGU Output Mapping for Non-LTE Modes**

| SC Output | SC Output in Binary Representation | OVSF Output | OVSF Output in Binary Representation | CGU Output | CGU Output in Binary Representation |
|---|---|---|---|---|---|
| 1+j | 00 | 1 | 0 | 1 | 00 |
| -1-j | 11 | 1 | 0 | -1 | 11 |
| -1+j | 01 | 1 | 0 | j | 01 |
| 1-j | 10 | 1 | 0 | -j | 10 |
| 1+j | 00 | -1 | 1 | -1 | 11 |
| -1-j | 11 | -1 | 1 | 1 | 00 |
| -1+j | 01 | -1 | 1 | -j | 10 |
| 1-j | 10 | -1 | 1 | j | 01 |

**Table 84. CGU Output Mapping for LTE Mode**

| SC Output | SC Output in Binary Representatation | OVSF Output | OVSF Output in Binary Representatation | CGU Output | CGU Output in Binary Representatation |
|---|---|---|---|---|---|
| 1 | 00 | 1 | 0 | 1 | 00 |
| -1 | 11 | 1 | 0 | -1 | 11 |

### 4.22.1 S1Mux add-on for CGU

S1Mux supports sign-inversion and real/imaginary swap of data just before the data is loaded into S1. The CGU sends its two-bit output to S1Mux to control these two operations. This is depicted in Figure 42 and Table 85.

Note that the real and imaginary are treated slightly different for the udfC/Complex mode.

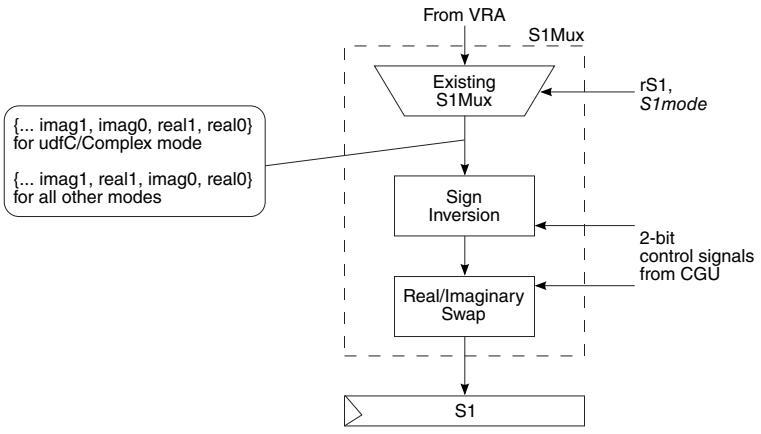



**Figure 42. CGU Interface to S1Mux**

**Table 85. CGU Manipulation of S1 Data (per 1 Complex Element)**

| CGU Output | CGU Output in Binary Represention | Invert Real Sign | Invert Imaginary sign | Swap Real/Imaginary |
|---|---|---|---|---|
| 1 | 00 | - | - | - |
| -1 | 11 | Y | Y | - |
| j | 01 | - | Y | Y |
| -j | 10 | Y | - | Y |

Only the *S1modes* listed below will activate the CGU unit. All other *S1modes* will not. Note that, these modes are not supported when the VAU is in F24 mode.

- cgu_hlinecplx
- cgu_udfc
- cgu_r2c_re0
- cgu_r2c_im0

When a ld.s1 instruction is executed with one of the above *S1mode*, the data pattern to be loaded into S1 will be the same as their non-cgu counterparts, except that the data will be post-modified by the "current" CGU output (that is, by the CGU output that was produced in the previous cycle). Then, the CGU states will be "post" updated and the CGU output will be used in subsequent ld.s1 instruction.

## 4.22.2 Downlink scrambling subunit

The Downlink Scrambling subunit contains two 18-bit feedback shift registers with fixed polynomials as specified by 3GPP TS 25.213 v8.2.0 Section 5.1.4. It is summarized below.

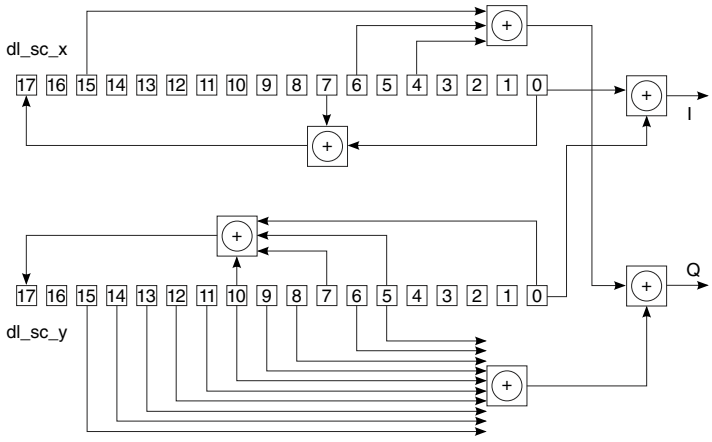



**Figure 43. Downlink Scrambling Shift Registers**

The Downlink Scrambling subunit will be become active when the *cgu_mode* is set to one of the following two modes:

- dl_sc1: shift by 1 bits and duplicate the 1-sample output by 8 times.
- dl_sc : shift by 8 bits and output 8 parallel samples

The Downlink Scrambling subunit contains 2 hardware registers, called *dl_sc_x* and *dl_sc_y*; both are accessible through the gp.move instruction.

### 4.22.3 Long-sequence uplink scrambling subunit

The Long-Sequence Uplink Scrambling subunit contains two 25-bit feedback shift registers with fixed polynomials as specified by 3GPP TS 25.213 v8.2.0 Section 5.1.4. It is summarized below.

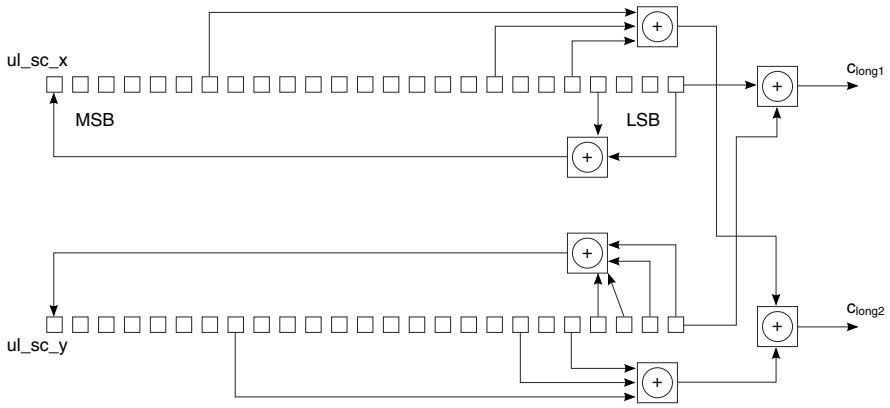



**Figure 44. Long-Sequence Uplink Scrambling Shift Registers**

$$C_{long}(i) = C_{long1}(i)\left(1 + j(-1)^{i} C_{long2}(2\lfloor i/2 \rfloor)\right) \ldots i = 0,1,\ldots,31$$

where $\lfloor\ \rfloor$ denotes round down operation.

The subunit output Clong(*i*) represents a complex sample in the form: I + jQ.

The Long-Sequence Uplink Scrambling subunit will be become active when the *cgu_mode* is set to "ul_sc_long".

The subunit supports "shift by 32-bits" only. It outputs 32 parallel samples per cycle.

The subunit contains 2 hardware registers, called *ul_sc_x* and *ul_sc_y*; both accessible through the gp.move instruction.

### 4.22.4 Short-sequence uplink scrambling subunit

The Short-Sequence Uplink Scrambling subunit contains

- two 8-bit feedback shift registers, d(i) and b(i), and
- one quatenrnary 8-digit feedback shift register, a(i).

Their polynomials are specified in 3GPP TS 25.213 v8.2.0 Section 5.1.4. It is summarized below.

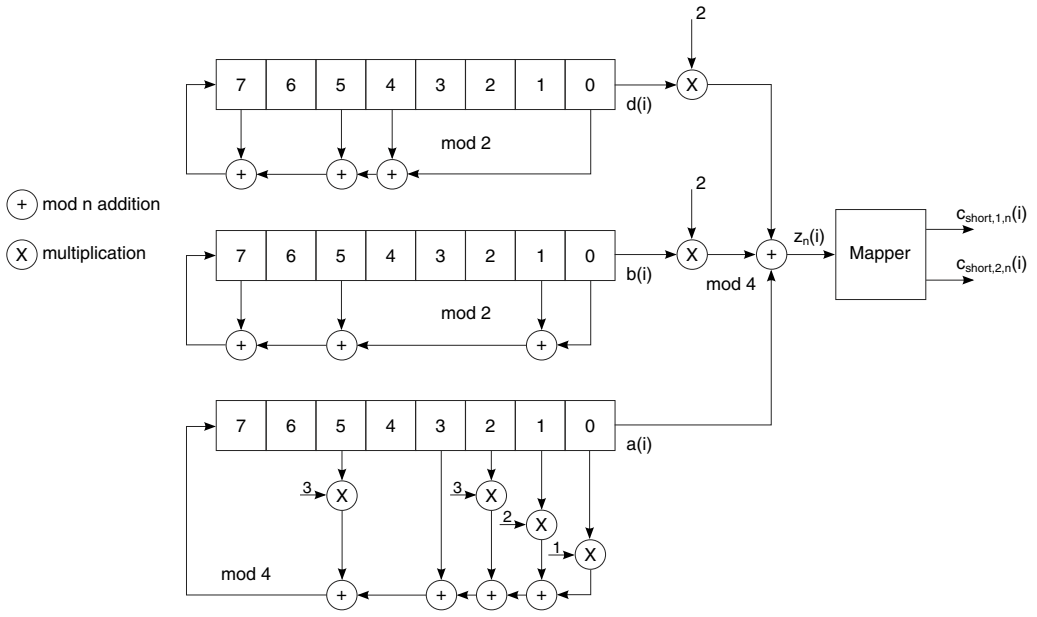



**Figure 45. Short-Sequence Uplink Scrambling Shift Registers**

**Table 86. Mapping from Zn(i) to Cshort1(i) and Cshort2(i), i=0,1,...,255**

| Zn(i) | Cshort1(i) | Cshort2(i) |
|---|---|---|
| 0 | +1 | +1 |
| 1 | -1 | +1 |
| 2 | -1 | -1 |
| 3 | +1 | -1 |

$$C_{short}(i) = C_{short1}(i)\left(1 + j(-1)^i C_{short2}\left(2\lfloor i/2 \rfloor\right)\right) \ldots i = 0,1,\ldots,31$$

where $\lfloor\ \rfloor$ denotes rounding-down operation.

Each $C_{short}(i)$ represents a complex sample in the form: I + jQ.

The 3GPP TS 25.213 standard stipulates using (*i* mod 256) in place of the *i* above, as shown below.

$$C_{short}(i) = C_{short1}(i \bmod 256)\left(1 + j(-1)^i C_{short2}\left(2\lfloor (i \bmod 256)/2 \rfloor\right)\right)$$

However, this mod function is not implemented in the CGU. Thus the CGU will only generate the correct output sequence for a length of up to 256 samples.

The Short-Sequence Uplink subunit will become active when *cgu_mode* is set to "ul_sc_short".

The subunit supports shift by 32-bits only. It outputs 32 parallel samples per cycle.

The subunit contains one hardware register, called *ul_sc_short*, accessible through the gp.move instruction. The *ul_sc_short* register is a concatenation version of the three registers specified in the 3GPP standard, as follows:

```
ul_sc_short = {a[7]lsb, a[6]lsb, ..., a[0]lsb,
               d[7], d[6], ..., d[0],
               b[7], b[6], ...' b[0],
               a[7]msb, a[6]msb, ..., a[0]msb }
```

where d[i] and b[i] are binary symbol (1 bit per symbol) and a[i] is quaternary symbol (2 bits per symbol). Thus, ul_sc_short is 4 bits wide.

Note: The bit format of *ul_sc_short* allows the translation from the scrambling code number, n[23:0], to be done simply by *ul_sc_short* = n + 0x1000000, that is, setting bit#24 of the register that contains n before programming the *ul_sc_short*.

### 4.22.5 LTE scrambling subunit (Gold sequence)

The LTE Scrambling subunit contains two 31-bit feedback shift registers with fixed polynomials as specified by 3GPP TS 36.211 v8.4.0 Section 7.2. It is summarized below.

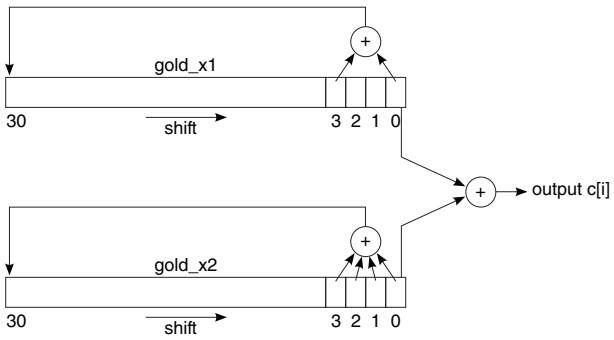



**Figure 46. LTE Gold Sequence Shift Registers**

The LTE Scrambling subunit will become active when *cgu_mode* is set to "lte_gold".

The subunit supports shift by 32-bits only. It outputs 32 parallel samples per cycle.

The subunit contains 2 hardware registers, called *gold_x1* and *gold_x2*, both accessible through the gp.move instruction.

### 4.22.6 V.fast/VDSL training sequence subunit

The V.fast/VDSL Training Sequence subunit, also known as G.fast/VDSL MEDLEY Sequence, contains a 23-bit feedback shift register with fixed polynomials as specified by ITU-T G.9701-201412 Section 10.2.1.4.1. It is summarized below.

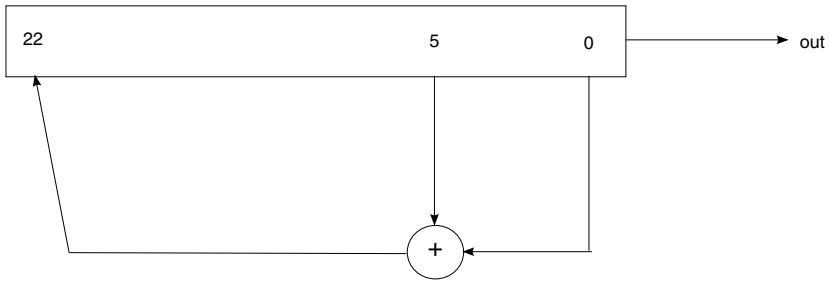



**Figure 47. V.fast/VDSL Training Sequence Subunit**

Every two bit output from G.fast/VDSL training sequence subunit is treated as one sample. The sample can be modified by the conjugate option and will be applied to the S1Mux in the same way as the other scrambling subunits.

The G.fast/VDSL training sequence subunit will become active when the cgu_mode is set to one of the following two modes:

- vd_ts1: shift by 2 bits and duplicate the 1-sample output by 8 times
- vd_ts: shift by 16 and output 8 in parallel samples

The G.fast/VDSL training sequence subunit contains one hardware register, called vd_ts_d, which is accessible through the gp.move instruction.

**NOTE**

After programming vd_ts_d register, the first output bit will be from bit 0 of vd_ts_d. To generate the sequence as specified in ITU-T G.9701-201412 where the first 23 bits are 1, the initial vd_ts should be programmed as 0x7fffff

### 4.22.7 V.fast/VDSL quadratic scrambling sequence subunit

The V.fast/VDSL Quadratic Scrambling Sequence subunit, also known as G.fast/VDSL MEDLEY Sequence, contains a 11-bit feedback shift registers with fixed polynomials as specified by ITU-T G.9701-201412 Section 10.2.2.4. It is summarized below.

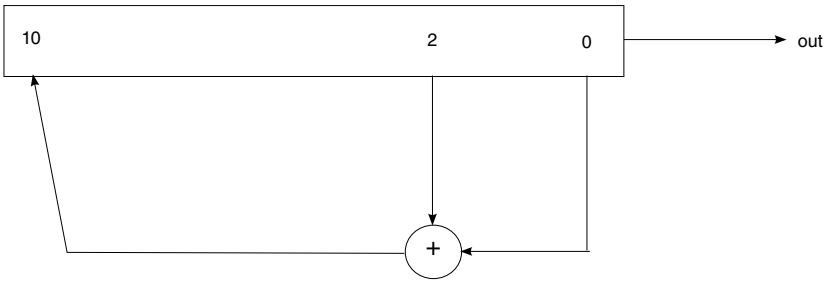



**Figure 48. V.fast/VDSL Quadratic Scrambling Sequence Subunit**

Every two bit output from G.fast/VDSL quadratic scrambling sequence subunit is treated as one sample. The sample can be modified by the conjugate option and will be applied to the S1Mux in the same way as the other scrambling subunits.

The G.fast/VDSL quadratic scrambling subunit will become active when the cgu_mode is set to one of the following two modes:

- vd_qs1: shift by 2 bits and duplicate the 1-sample output by 8 times
- vd_qs: shift by 16 and output 8 in parallel samples

The G.fast/VDSL quadratic scrambling subunit contains one hardware register, called vd_qs_d, which is accessible through the gp.move instruction.

**NOTE**

After programming vd_qs_d register, the first output bit will be from bit 0 of vd_qs_d.

### 4.22.8 OVSF unit

The OVSF unit generates OVSF Code Sequence with a Spreading Factor (SF) of 256, as specified by 3GPP TS 36.211 v8.4.0 Section 7.2. This is summarized in Figure 49. There are 256 unique codes for SF = 256.

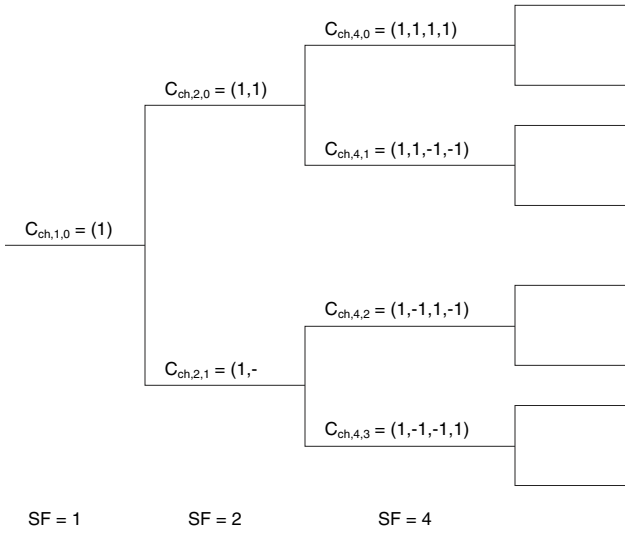



**Figure 49. OVSF Code Tree (from 3GPP TS 25.213)**

The OVSF unit generates 32 parallel samples in one cycle.

It contains one hardware register, *ovsf_num*, writeable through gp.move instruction.

It maintains an 8-bit internal sample index, called *ovsf_idx*. This register is reset whenever the *ovsf_num* register is written to by the gp.move instruction.

Note: The microcode can use the following expression to generate OVSF code with SF smaller than 256, by left-justifying the *ovsf_num* to an 8-bit value, that is,

$$\text{ovsf_num}_{SF=256} = \text{ovsf_num} << (8 - \log2\ (SF))$$

# 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.

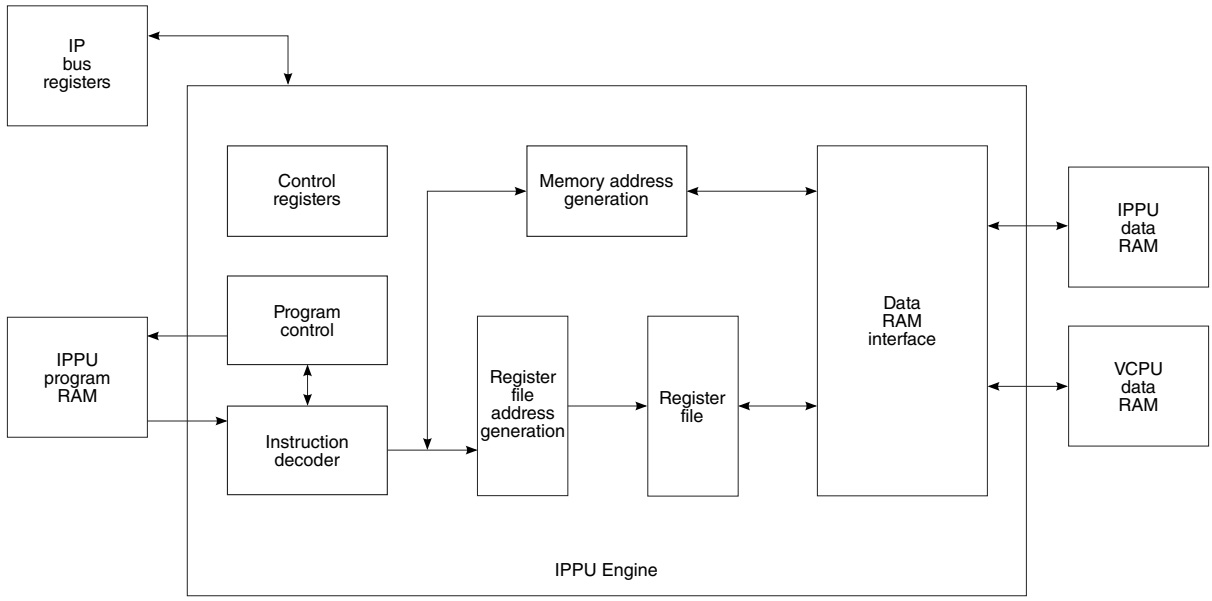



**Figure 50. IPPU Core Block Diagram**

**Table 87. 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.<br>• 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 87. IPPU Core Components (continued)**

| Component | Description |
|---|---|
| | • Each register can contain either 64 real half-precision elements or 32 complex half-precision elements.<br>• Each register in RF is 1024 bits wide.<br>• r1/r0 concatenate to present a virtual 2048 bit register.<br>• r2 and r3 can be used only in a store (st.mem) command in conjunction with the ld.brmr instruction |
| 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 51.

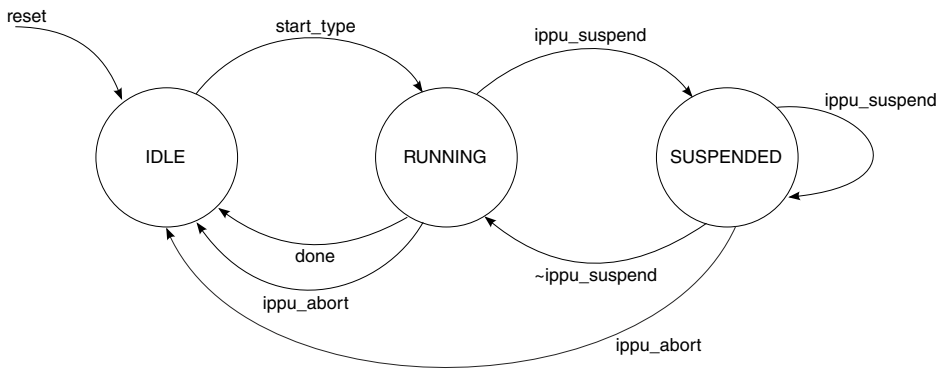



**Figure 51. IPPU Operating State Diagram**

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, 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 88 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 88. 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 89. Hardware definitions**

| What | Name | Size | Description |
|---|---|---|---|
| DMEM address pointers | a0, a1, a2, a3 | 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<br>• Fetches either (i) a 32-bit DMEM element, or (ii) a DMEM line consisting of N_elem 32-bit elements<br>• These are general memory access pointers. The address contained in a0, a1, a2, a3 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. |

*Table continues on the next page...*

**Table 89. Hardware definitions (continued)**

| What | Name | Size | Description |
|---|---|---|---|
| | | | • Registers a0 and a1 support modulo range. Registers a2 and a3 do not support modulo range. Registers a2,a3 are used only in the following load instructions:<br>— ld.index<br>— ld.mask |
| 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.<br>• Elements can be moved to/from these registers DMEM address registers, offset registers. |
| Data Register file | r0, r1 , r3 | N_bits_line | • Storage for elements fetched from memory.<br>• 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.<br>• r3 can be used only in a store (st.mem) command in conjunction with the ld.brmr instruction |
| 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.<br>• 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.<br>• 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, a2, a3. |
| 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.<br>• Used for vectorized index addressing (see Vectorized indirect addressing for details). |

## 6.3 IPPU instructions summary

**Table 90. IPPU instructions summary**

| Instruction | Cycles | Description |
|---|---|---|
| ld.type [a*X*]+/-m*X*, elem_offset, ld_mode, wr_offset, latch_mode | 4 | Load instructions |
| ld.type [a*X*]+/-Is8, elem_offset, wr_offset, latch_mode | | |
| ld.type [a*X*]+/-Is5, elem_offset,ld_mode, wr_offset, latch_mode | | |
| ld.mask.type [a*X*]+/-m*X* | | |
| ld.mask.type [a*X*]+/-Is9 | | |
| ld.index.type [a*X*]+/-m*X* | 3 | Load memory index instructions |
| ld.index.type [a*X*]+/-Is9 | | |
| st.type [a*X*]+/-m*X*, elem_offset, st_mode, rd_offset, write_mode | 2 | Store instructions |
| st.type [a*X*]+/-Is8, elem_offset, rd_offset, write_mode | | |
| st.type [a*X*]+/-Is5, elem_offset, st_mode, rd_offset, write_mode | | |
| set.range a*Y*, as*A*, as*B* | 1 | Set range instructions |
| set.range a*Y*, as*A*, Iu18 | | |
| set.br a*X*, 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.arg.type as*X*, 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 | | |

*Table continues on the next page...*

**Table 90. IPPU instructions summary (continued)**

| Instruction | Cycles | Description |
|---|---|---|
| `rts` | | |
| `rts.z` | | |
| `rts.nz` | | |
| `set.loop Iu1, Iu8` | 2 | Loop instructions<br>**NOTE**<br>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 Is8 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 91. Load instruction arguments**

| Argument | Description |
|---|---|
| type | Indicates which DMEM to access.<br>ippu<br>vcpu |
| a*X* | Pointer to DMEM word location (X = 0, 1, 2, 3).<br>0 a0<br>1 a1<br>2 a2<br>3 a3<br>**NOTE**<br>a2 and a3 can be used only in the ld.mask instructions. |
| +/- | 0 inc (add)<br>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).<br>0 m0<br>1 m1 |
| Is8 | 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). |

*Table continues on the next page...*

**Table 91. Load instruction arguments (continued)**

| Argument | Description |
|---|---|
| | **NOTE**<br>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).<br>**NOTE**<br>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. |
| ld_mode | Loading mode specifying that the address generation that can take following values:<br>000 normal: linear address mapping<br>001 br: bit-reversed address mapping<br>010 dr: digit-reversed address mapping<br>011 br_dr: digit and bit reversed mapping<br>100 vindirect: vectorized indirect mapping (see Vectorized indirect addressing for details)<br>101 ind: indirect mapping (see Indirect addressing for details)<br>• 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.<br>110 ri: (relative indirect), relative indirect mapping<br>• 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.<br>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. |

*Table continues on the next page...*

**Table 91. Load instruction arguments (continued)**

| Argument | Description |
|---|---|
| | 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.<br>• The destination location is specified by the r_wr_ptr. The least significant 6 bits of the write pointer are ignored.<br>• The least significant 3 bits of the DMEM address (read) pointer are ignored.<br>**NOTE**<br>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.<br>• The destination location is specified by the r_wr_ptr. The least significant 5 bits of the write pointer are ignored.<br>• The least significant 2 bits of the DMEM address (read) pointer are ignored. |
| | 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.<br>• The destination location is specified by the r_wr_ptr. The least significant 4 bits of the write pointer are ignored.<br>• 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.<br>• 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.<br>• The 16-bit source element is specified by the most significant bit of mem_elem_rd_ptr.<br>• 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.<br>• The 8-bit source element is specified by the most significant 2 bits of mem_elem_rd_ptr.<br>• 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.<br>• The 4-bit source element is specified by mem_elem_rd_ptr. |

*Table continues on the next page...*

**Table 91. Load instruction arguments (continued)**

| Argument | Description |
|---|---|
| | • 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.<br>**NOTE**<br>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.<br>**NOTE**<br>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.<br>**NOTE**<br>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.<br>**NOTE**<br>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. |
| | 1100 brmr: Bit Reverse Multiple Registers. Latches multiple 32-bit elements from the read bus onto multiple destination registers. Refer to section "BRMR - Bit-reversal-multiple-registers" for more details. |
| | 1101 er: Element Reverse. Latches multiple elements from the read bus onto the destination register. Swaps 64/32/16 bit elements as they are loaded from the DMEM into the IPPU registers R1/R0. Refer to section "Element-reversal" for more details. |
| | 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 92. Load memory index instruction arguments**

| Argument | Description |
|---|---|
| type | Indicates which DMEM to access.<br>ippu<br>vcpu |
| a*X* | Pointer to DMEM location (X = 0, 1).<br>0 a0<br>1 a1<br>When pointer a*X*==a0 load into register mem_index0<br>When pointer a*X*==a1 load into register mem_index1<br>Note: The index register loaded X is the same X as the a*X* used in this instruction. |
| +/- | 0 inc (add)<br>1 dec (subtract) |
| m*X* | Offset register specifying a post-fetch (increment or decrement) offset for the DMEM access pointer a*X*.<br>0 m0<br>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 93. Store instruction arguments**

| Argument | Description |
|---|---|
| type | Indicates which DMEM to access:<br>{ippu,vcpu} |
| a*X* | Pointer to DMEM word location (X = 0, 1, 2, 3).<br>0 a0<br>1 a1<br>2 a2<br>3 a3 |
| +/- | 0 inc (add)<br>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).<br>0 m0<br>1 m1 |
| Is8 | Signed 8-bit immediate field, indicating a post-write (increment or decrement) offset for the DMEM access pointer aN (ignored in indirect mode).<br>This field CANNOT be used in conjunction with the st_mode field. |
| Is5 | Signed 5-bit immediate field, indicating a post-write (increment or decrement) offset for the DMEM access pointer aN (ignored in indirect mode).<br>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:<br>000 normal: linear address mapping<br>001 br: bit-reversed address mapping<br>010 dr: digit-reversed address mapping |

*Table continues on the next page...*

**Table 93. Store instruction arguments (continued)**

| Argument | Description |
|---|---|
| | 011 br_dr: digit and bit reversed mapping<br>101 indirect: indirect mapping (see Indirect addressing for details)<br>• 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<br>110 rind: relative indirect mapping<br>• 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.<br>100, 111 Reserved |
| 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).<br>• 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).<br>• The least significant 2 bits of r_rd_ptr are ignored.<br>• The destination 16-bit element within the 32-bit DMEM location is specified by the MS bit of mem_elem_wr_ptr. |
| | 110 r3: Writes the entire data register line R3 to the DMEM address (specified by a*X*). |
| | 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 94. Set Range instruction arguments**

| Argument | Description |
|---|---|
| a*Y* | Specifies the pointer to which the set range instruction applies (Y = 0, 1).<br>0 a0<br>1 a1 |
| as*A* | A general register (as0, as1, as2 or as3) which specifies the start address of the circular buffer.<br>00 as0<br>01 as1<br>10 as2<br>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).<br>00 as0<br>01 as1<br>10 as2<br>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 95. 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 96. Configure bit-reversal, digit-reversal engine instruction arguments**

| Argument | Description |
|---|---|
| a*X* | X = 0, 1, 2, 3.<br>0 a0<br>1 a1<br>2 a2<br>3 a3 |
| br_mode | Bit-reversal mode.<br>0000 fft32<br>0001 fft64<br>0010 fft128<br>0011 fft256<br>0100 fft512<br>0101 fft1024<br>0110 fft2048<br>0111 fft4096<br>1000 fft8192<br>1001 fft16384<br>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).<br>000 Reset radix registers when lu11 also is zero.<br>001 Not valid<br>010 2 prime radix factor<br>011 3 prime radix factor<br>100 Not valid |

*Table continues on the next page...*

**Table 96. Configure bit-reversal, digit-reversal engine instruction arguments (continued)**

| Argument | Description |
|---|---|
| | 101 5 prime radix factor<br>110 Not valid<br>111 Not valid |
| Iu3 | 3-bit immediate value indicating the counter values for address generation. |
| Iu11 | *set.dr.radix digit_index, radix, Iu11* : 11-bit unsigned immediate value indicating the address increment. Value of zero together with radix=0 resets the radix registers.<br>*set.dr.config digit_index, Iu3, Iu11* : 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. |
| 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, Iu19
```

*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 97. Move Register instruction arguments**

| Argument | Description |
|---|---|
| ippu_regA | Move destination register<br>One of:<br>• aX (X = 0, 1, 2, 3)<br>• mX (X = 0, 1)<br>• asX (X = 0, 1, 2, 3)<br>• vindx_ptr<br>• r_rd_ptr<br>• r_wr_ptr<br>• elem_mask_ptr<br>• mem_elem_rd_ptr<br>• mem_elem_wr_ptr |
| ippu_regB | Move source register<br>One of:<br>• aX (X = 0, 1, 2, 3)<br>• mX (X = 0, 1)<br>• asX (X = 0, 1, 2, 3)<br>• vindx_ptr<br>• r_rd_ptr<br>• r_wr_ptr<br>• elem_mask_ptr<br>• mem_elem_rd_ptr<br>• mem_elem_wr_ptr |
| Iu19 | 19-bit immediate value |

## 6.10 Load input argument instruction

IPPU instructions summary

*Syntax*:

```
ld.arg.type 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 or VCPU 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.arg.ippu 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 98. Load input argument instruction arguments**

| Argument | Description |
|---|---|
| type | Indicates which DMEM to access: (ippu, vcpu) |
| 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.<br>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 99. 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 100. Jump engine instructions**

| Instruction | Description |
|---|---|
| jmp<br>jsr | Unconditional jump |

*Table continues on the next page...*

**Table 100. Jump engine instructions (continued)**

| Instruction | Description |
|---|---|
| rts | |
| jmp.z<br>jsr.z<br>rts.z | Jump is taken if ZERO flag is set. |
| jmp.nz<br>jsr.nz<br>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 101. 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 102. Loop instruction arguments**

| Argument | Description |
|---|---|
| lu1 | A 1-bit immediate value which specifies which level of loop is being configured (0, 1). |
| lu8 | 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 103. 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 104. 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 105. Add instruction arguments**

| Argument | Description |
|---|---|
| a*X* | Pointer to DMEM location (X = 0, 1) |

*Table continues on the next page...*

**Table 105. Add instruction arguments (continued)**

| Argument | Description |
|---|---|
| | 0 - a0<br>1 - a1 |
| ls8 | 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)<br>0 - m0<br>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 BRMR - Bit-reversal-multiple-registers

The bit-reversal-multiple-registers feature allows data to be read from Data Memory in a bit-reversed order into multiple (2 or 4) registers, thus accelerating the process versus the Bit Reversal mode. Similar to the bit reversal mode, in this mode the bit reversal the engine 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 - however using multiple internal registers thus using less clock cycles.

- 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` instruction in brmr mode to read data. Note, enabling the bit-reversal addressing mode is done with the set.br instruction. The Bit-Reversal (br) mode is sticky.
- Use the `st.type` instruction in normal mode to store the data to Data Memory.

The following examples are gives as sample code utilizing the brmr mode for several different FFT sizes. The element size in these examples is 32-bit, which can be a 2*(16-bit (HP,or HF)) complex number, a 32-bit SP real number or a 32-bit integer number.

**Example 1: 16AU VSPA using brmr mode with FFT of size 16384 (16K)**

```
 .set LINE, NUM_AU*4

  mv m0, LINE*16;     // 16 lines

// BRMR
  mv as1, 0x0;        // Read  Buffer Address
  set.range a1, as1,16384*2;
  clr r1,r0,r2,r3;
  set.br a1, fft16384;
  mv  a0, 0x8000;     // Write Buffer Address
  mv a1, as1;         // Read  Buffer Address
  mv r_wr_ptr, 0x0;
  mv mem_elem_rd_ptr, 0x0;
  set.loop 1,  8;     // Num repeat the loop
  set.loop 0, 32;     // Num Elem per Reg

  loop_begin 1;
//#0
  nop;                           // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                           // wait needed
  nop;                           // wait needed
  mv m1, LINE*47;                // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#1
  nop;                           // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                           // wait needed
  nop;                           // wait needed
  mv m1, LINE*47;                // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#2
  nop;                           // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                           // wait needed
  nop;                           // wait needed
```

```
  mv m1, LINE*47;               // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#3
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE*47;               // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#4
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE*47;               // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#5
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE*47;               // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#6
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE*47;               // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#7
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE*47;               // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#8
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE*47;               // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#9
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
```

```
  nop;                          // wait needed
  mv m1, LINE*47;               // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#10
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE*47;               // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#11
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE*47;               // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#12
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE*47;               // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#13
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE*47;               // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#14
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE*47;               // 47 lines (3*16)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#15
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE;                  // 1 lines
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]+m1,0,normal, 0,r3; loop_end 1;

  done;
```

**Example 2: 16AU VSPA using brmr mode with FFT of size 8198 (8K)**

```
  .set LINE, NUM_AU*4

  mv m0, LINE*8;      // 8  lines

// BRMR
  mv as1, 0x0;        // Read  Buffer Address
  set.range a1, as1,8192*2;
  clr r1,r0,r2,r3;
  set.br a1, fft8192;
  mv  a0, 0x4000;     // Write Buffer Address
  mv a1, as1;         // Read  Buffer Address
  mv r_wr_ptr, 0x0;
  mv mem_elem_rd_ptr, 0x0;
  set.loop 1,  8;     // Num repeat the loop
  set.loop 0, 32;     // Num Elem per Reg

  loop_begin 1;
//#0
  nop;                           // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                           // wait needed
  nop;                           // wait needed
  mv m1, LINE*23;                // 11 lines (3*8)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#1
  nop;                           // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                           // wait needed
  nop;                           // wait needed
  mv m1, LINE*23;                // 11 lines (3*8)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#2
  nop;                           // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                           // wait needed
  nop;                           // wait needed
  mv m1, LINE*23;                // 11 lines (3*8)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#3
  nop;                           // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                           // wait needed
  nop;                           // wait needed
  mv m1, LINE*23;                // 11 lines (3*8)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#4
  nop;                           // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                           // wait needed
  nop;                           // wait needed
  mv m1, LINE*23;                // 11 lines (3*8)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
```

```
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#5
  nop;                            // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                            // wait needed
  nop;                            // wait needed
  mv m1, LINE*23;                 // 11 lines (3*8)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#6
  nop;                            // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                            // wait needed
  nop;                            // wait needed
  mv m1, LINE*23;                 // 11 lines (3*8)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#7
  nop;                            // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                            // wait needed
  nop;                            // wait needed
  mv m1, LINE;                    // 1 lines
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]+m1,0,normal, 0,r3; loop_end 1;

  done;
```

**Example 3: 16AU VSPA using brmr mode with FFT of size 4096 (4K)**

```
  .set LINE, NUM_AU*4

  mv m0, LINE*4;      // 4 lines

// BRMR
  mv as1, 0x0;        // Read  Buffer Address
  set.range a1, as1,4096*2;
  clr r1,r0,r2,r3;
  set.br a1, fft4096;
  mv  a0, 0x4000;     // Write Buffer Address
  mv a1, as1;         // Read  Buffer Address
  mv r_wr_ptr, 0x0;
  mv mem_elem_rd_ptr, 0x0;
  set.loop 1,  8;     // Num repeat the loop
  set.loop 0, 32;     // Num Elem per Reg

  loop_begin 1;
//#0
  nop;                            // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                            // wait needed
  nop;                            // wait needed
  mv m1, LINE*11;                 // 11 lines (3*4)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#1
  nop;                            // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                            // wait needed
  nop;                            // wait needed
```

```
  mv m1, LINE*11;               // 11 lines (3*4)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#2
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE*11;               // 11 lines (3*4)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#3
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE;                  // 1 line
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]+m1,0,normal, 0,r3; loop_end 1;

  done;
```

**Example 4: 16AU VSPA using brmr mode with FFT of size 2048 (2K)**

```
  .set LINE, NUM_AU*4

  mv m0, LINE*2;      // 2 lines

// BRMR
  mv as1, 0x0;        // Read  Buffer Address
  set.range a1, as1,2048*2;
  clr r1,r0,r2,r3;
  set.br a1, fft2048;
  mv  a0, 0x4000;     // Write Buffer Address
  mv a1, as1;         // Read  Buffer Address
  mv r_wr_ptr, 0x0;
  mv mem_elem_rd_ptr, 0x0;
  set.loop 1,  8;     // Num repeat the loop
  set.loop 0, 32;     // Num Elem per Reg

  loop_begin 1;
//#0
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE*5;                // 5 line (3*2)-1
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]-m1,0,normal, 0,r3;
//#1
  nop;                          // needed
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                          // wait needed
  nop;                          // wait needed
  mv m1, LINE;                  // 1 line
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]+m1,0,normal, 0,r3; loop_end 1;
```

```
  done;
```

**Example 5: 16AU VSPA using brmr mode with FFT of size 1024 (1K)**

```
  .set LINE, NUM_AU*4

  mv m0, LINE;        // 1 line

// BRMR
  mv as1, 0x0;        // Read  Buffer Address
  set.range a1, as1,1024*2;
  clr r1,r0,r2,r3;
  set.br a1, fft1024;
  mv  a0, 0x4000;     // Write Buffer Address
  mv a1, as1;         // Read  Buffer Address
  mv r_wr_ptr, 0x0;
  mv mem_elem_rd_ptr, 0x0;
  set.loop 1,  8;     // Num repeat the loop
  set.loop 0, 32;     // Num Elem per Reg

  loop_begin 1;
  nop;                // nedded
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                // wait needed
  nop;                // wait needed
  nop;                // wait needed
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]+m0,0,normal, 0,r3; loop_end 1;

  done;
```

**Example 6: 16AU VSPA using brmr mode with FFT of size 512**

```
  .set LINE, NUM_AU*4

  mv m0, LINE;        // 1 line

// BRMR
  mv as1, 0x0;        // Read  Buffer Address
  set.range a1, as1,1024*2;
  clr r1,r0,r2,r3;
  set.br a1, fft1024;
  mv  a0, 0x4000;     // Write Buffer Address
  mv a1, as1;         // Read  Buffer Address
  mv r_wr_ptr, 0x0;
  mv mem_elem_rd_ptr, 0x0;
  set.loop 1,  8;     // Num repeat the loop
  set.loop 0, 32;     // Num Elem per Reg

  loop_begin 1;
  nop;                // nedded
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                // wait needed
  nop;                // wait needed
  nop;                // wait needed
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]+m0,0,normal, 0,r3; loop_end 1;

  done;
```

**Example 7: 16AU VSPA using brmr mode with FFT of size 256**

```
  .set LINE, NUM_AU*4

  mv m0, LINE;        // 1 line

// BRMR
  mv as1, 0x0;        // Read  Buffer Address
  set.range a1, as1,256*2;
  clr r1,r0,r2,r3;
  set.br a1, fft256;
  mv  a0, 0x4000;     // Write Buffer Address
  mv a1, as1;         // Read  Buffer Address
  mv r_wr_ptr, 0x0;
  mv mem_elem_rd_ptr, 0x0;
  set.loop 1,  2;     // Num repeat the loop
  set.loop 0,  8;     // Num Elem per Reg

  loop_begin 1;
  nop;                // nedded
  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                // wait needed
  nop;                // wait needed
  nop;                // wait needed
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]+m0,0,normal, 0,r3; loop_end 1;

  done;
```

**Example 8: 16AU VSPA using brmr mode with FFT of size 128**

```
  .set LINE, NUM_AU*4

  mv m0, LINE;        // 1 line

// BRMR
  mv as1, 0x0;        // Read  Buffer Address
  set.range a1, as1,128*2;
  clr r0,r1,r2,r3;
  set.br a1, fft128;
  mv  a0, 0x4000;     // Write Buffer Address
  mv a1, as1;         // Read  Buffer Address
  mv r_wr_ptr, 0x0;
  mv mem_elem_rd_ptr, 0x0;
  set.loop 0,  4;     // Num Elem per Reg
  nop;                // nedded
  nop;                // nedded

  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                // wait needed
  nop;                // wait needed
  nop;                // wait needed
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;
  st.vcpu [a0]+m0,0,normal, 0,r2;
  st.vcpu [a0]+m0,0,normal, 0,r3;

  done;
```

**Example 9: 16AU VSPA using brmr mode with FFT of size 64**

```
  .set LINE, NUM_AU*4

  mv m0, LINE;        // 1 line
```

```
// BRMR
  mv as1, 0x0;        // Read  Buffer Address
  set.range a1, as1,64*2;
  clr r0,r1,r2,r3;
  set.br a1, fft64;
  mv  a0, 0x4000;     // Write Buffer Address
  mv a1, as1;         // Read  Buffer Address
  mv r_wr_ptr, 0x0;
  mv mem_elem_rd_ptr, 0x0;
  set.loop 0,  2;     // Num Elem per Reg
  nop;                // nedded
  nop;                // nedded

  ld.vcpu [a1]+2,0,br, 8, brmr; loop_begin 0; loop_end 0;
  nop;                // wait needed
  nop;                // wait needed
  nop;                // wait needed
  st.vcpu [a0]+m0,0,normal,-1,normal;
  st.vcpu [a0]+m0,0,normal,+1,normal;

  done;
```

**Example 10: 16AU VSPA using brmr mode with FFT of size 32**

```
  .set LINE, NUM_AU*4

  mv m0, LINE;        // 1 line

// BRMR
  mv as1, 0x0;        // Read  Buffer Address
  set.range a1, as1,32*2;
  clr r0,r1,r2,r3;
  set.br a1, fft32;
  mv  a0, 0x4000;     // Write Buffer Address
  mv a1, as1;         // Read  Buffer Address
  mv r_wr_ptr, 0x0;
  mv mem_elem_rd_ptr, 0x0;
  ld.vcpu [a1]+2,0,br, 8, brmr;
  nop;                //wait needed
  nop;                //wait needed
  nop;                //wait needed
  st.vcpu [a0]+m0,0,normal,-1,normal;

  done;
```

### 6.18.4 Element-reversal

The Element-Reversal feature allows data to be read from Data Memory in an element reversed order. Swaps 64, 32, or 16 bit elements as they are loaded from the DMEM into the IPPU registers R1/R0. The MSB of the register r_wr_ptr selects whether R1 or R0 is selected as a destination register. VRA in the following description is R1 or R0. There is no bit reversal within each element.

- 16 AU configuration
  - 64-bit elements:
    - Bits 63: 0 of read buffer load into VRA[1023:960] (64-bit element 0 goes to element 15)
    - Bits 127:64 of read buffer load into VRA[ 959:896] (64-bit element 1 goes to element 14)
    - ...
    - Bits 1023:960 of read buffer load into VRA[63: 0] (64-bit element 15 goes to element 0)
  - 32-bit elements:

- Bits 31: 0 of read buffer load into VRA[1023:992] (32-bit element 0 goes to element 31)
- Bits 63:32 of read buffer load into VRA[ 991:960] (32-bit element 1 goes to element 30)
- ...
- Bits 1023:992 of read buffer load into VRA[31: 0] (32-bit element 31 goes to element 0)

— 16-bit elements:

- Bits 15: 0 of read buffer load into VRA[1023:1008] (16-bit element 0 goes to element 63)
- Bits 31:16 of read buffer load into VRA[ 1007:992] (16-bit element 1 goes to element 63)
- ...
- Bits 1023:1008 of read buffer load into VRA[15: 0] (16-bit element 63 goes to element 0)

- 2 AU configuration

To operate the element reversal engine, the following steps must be followed:

- Configure the Element reverse mode: `set.er er64/er32/er16;`
- Use the `ld.type` instruction in er mode to read the data. example: `ld.vcpu [a0]+m0, 0, normal, -256, er;`

For example, refer the following IPPU code sample for element reversal:

```
  .ippu
  .section .itext

  .set R_WR_PTR_INC,   -(NUM_AU*16)

  mv  as0, 0x00;
  set.range a0, as0, 0x0;
  set.range a1, as0, 0x0;
  clr r1,r0;

  mv a0, ERL_IN;                // input vector (1st line)
  mv a1, ERL_ACT;               // dest vector @ idmem line 0x11
  mv m0, NUM_AU*4;              // 1 line

  jmp er<sz>;                    // set sz to one of {64, 32, 16}
  nop;
  nop;


er64:
  mv r_wr_ptr,   0;
  mv r_rd_ptr,   0;
  set.er  er64;
  ld.vcpu [a0]+m0, 0, normal, -256, er; // ld r0
  ld.vcpu [a0]+m0, 0, normal, -256, er; // ld r1
  ld.vcpu [a0]+m0, 0, normal, -256, er; // ld r0
  ld.vcpu [a0]+m0, 0, normal, -256, er; // ld r1
  st.vcpu [a1]+m0, 0, normal, -256, normal;
  st.vcpu [a1]+m0, 0, normal, -256, normal;
  st.vcpu [a1]+m0, 0, normal, -256, normal;
  st.vcpu [a1]+m0, 0, normal, -256, normal;
  nop;
  nop;
  nop;
  nop;

er32:
  mv r_wr_ptr,   0;
  mv r_rd_ptr,   0;
  set.er  er32;
  ld.vcpu [a0]+m0, 0, normal, -256, er; // ld r0
```

```
  ld.vcpu [a0]+m0, 0, normal, -256, er; // ld r1
  ld.vcpu [a0]+m0, 0, normal, -256, er; // ld r0
  ld.vcpu [a0]+m0, 0, normal, -256, er; // ld r1
  st.vcpu [a1]+m0, 0, normal, -256, normal;
  st.vcpu [a1]+m0, 0, normal, -256, normal;
  st.vcpu [a1]+m0, 0, normal, -256, normal;
  st.vcpu [a1]+m0, 0, normal, -256, normal;
  nop;
  nop;
  nop;
  nop;

er16:
  mv r_wr_ptr,   0;
  mv r_rd_ptr,   0;
  set.er  er16;
  ld.vcpu [a0]+m0, 0, normal, -256, er; // ld r0
  ld.vcpu [a0]+m0, 0, normal, -256, er; // ld r1
  ld.vcpu [a0]+m0, 0, normal, -256, er; // ld r0
  ld.vcpu [a0]+m0, 0, normal, -256, er; // ld r1
  st.vcpu [a1]+m0, 0, normal, -256, normal;
  st.vcpu [a1]+m0, 0, normal, -256, normal;
  st.vcpu [a1]+m0, 0, normal, -256, normal;
  st.vcpu [a1]+m0, 0, normal, -256, normal;
  nop;
  nop;
  nop;
  nop;

  nop;
  nop;
  nop;
  nop;
  nop;
  nop;
  nop;
  nop;
  done
  nop;
  nop;
```

### 6.18.5 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.

### 6.18.6 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 Controllers

## 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 32 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.

When the DMA is reading from DMEM and writing to AXI the DMA will always speculatively fetch when the last 64 bytes of a DMEM line are being sent on the AXI wdata bus and wvalid and wready are high. If the size of an AXI transfer is such that the final bytes that are sent are from that last 64 bytes of a DMEM line, the speculative DMEM fetch is done but it is not needed and will be ignored. In some cases this fetch may be from a DMEM line which has already been read.

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 the byta address is 0, then di_mode 010b is used. If the MSB of the the byta 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.

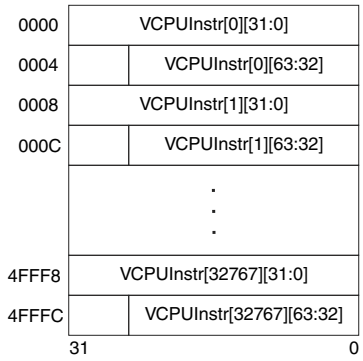



**Figure 52. VCPU instruction (Rinstr) format in system memory**

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.

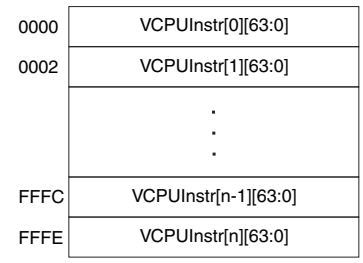



**Figure 53. VCPU instruction (Rinstr) format in program RAM**

System memory to IPPU program RAM (IPPU PRAM):

IPPU instructions must be "firmly" packed into system memory (external to IPPU).

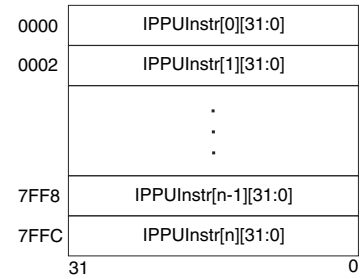



**Figure 54. IPPU instruction (Rinstr) format in system memory**

IPPU instructions that are fetched from system memory are packed into IPPU's program RAM (from the perspective of the IPPU program counter).

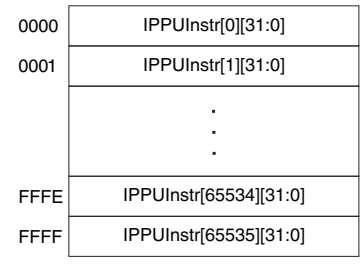



**Figure 55. IPPU instruction (RUinstr) format in program IPPU PRAM**

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 58, Figure 59). 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.

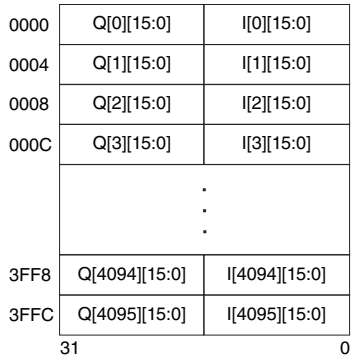



**Figure 56. Complex Data Format in System Memory**

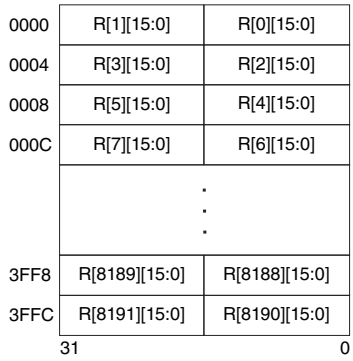



**Figure 57. Real data format in system memory**

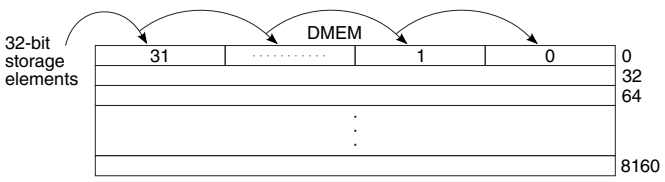



**Figure 58. Data format in internal DMEM**

Data that is "dumped" from internal DMEM to external system memory is packed into external system memory, as shown in Figure 58. 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.

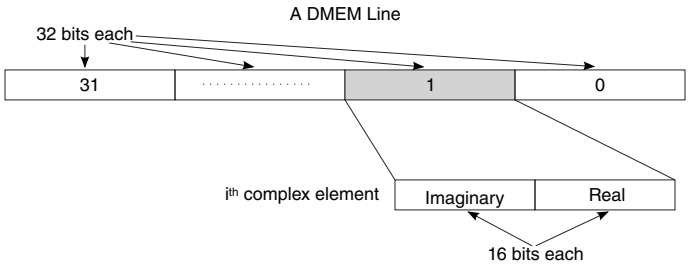



**Figure 59. Data line in internal DMEM**

## 7.2 Auxilary direct memory access unit (AUX_DMA)

The VSPA AUX_DMA is capable of transferring data between the VCPU program memory and a private AXI bus interface. It is intended to load VCPU's program RAM (PRAM).

### 7.2.1 AUX_DMA module operation

The VSPA AUX_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 AUX_DMA controls a private AXI bus master interface, so using it does not directly interfere with the VSPA DMA.

The AUX_DMA can overlay new program code onto the VCPU PMEM while the VCPU is running, without slowing down or interfering with the operation of the VCPU in any way. This is made possible by using a 512-bit VCPU PMEM read buffer. The VCPU PMEM read buffer is loaded from PMEM whenever the VCPU does an instruction fetch targeting an address that does not already reside in the read buffer. So, the VCPU can do multiple instruction fetches per VCPU PMEM read since the instruction fetches are 64 bits, and the read buffer width is 512-bits. The VCPU has priority access to the PMEM, but the read buffer allows the AUX_DMA to write to PMEM while the VCPU simultaneously fetches instructions from the read buffer.

The AUX_DMA has 1 read channel, and it is driven by a set of 4 IP registers which hold all the information required for a AUX_DMA transfer, including starting addresses, byte count, and VSPA go mode.

There is a status register indicating whether the channel's transfers are active, have completed, or had any type of error.

### 7.2.2 Issuing AUX_DMA commands

An entity desiring an AUX_DMA operation programs the command/control registers. Note that this is a command based interface, with a four register set (AD_PRAM_ADDR, AD_AXI_ADDR, AD_BYTE_COUNT, and AD_CTRL) that acts as the command buffer. The interface must be programmed by writing last to the AD_CTRL register, which triggers the AUX_DMA to begin the transaction.

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 AUX_DMA command interface acts as a command buffer, the VCPU can issue an AUX_DMA command that exactly duplicates the last AUX_DMA command issued by the VCPU merely by writing to the AD_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 AUX_DMA in the same fashion as the VCPU, using the same registers. Unlike the primary DMA, the host and VCPU are sharing control over the same AUX_DMA channel. If the host and VSPA will both be programming the AUX_DMA, a software semaphore should be used to control access to the AUX_DMA command registers. The host interface 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 AUX_DMA registers should not be interruptible by another host or host routine that can program the AUX_DMA registers.

# 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), 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 DTU is responsible for generation of non-invasive debug trace messages and is composed of multiple trace generation units: program trace messaging (PTM), watchpoint messaging (WPM), data acquisition messaging (DQM), timestamp messaging (TSM), and data trace messaging (DTM). These units interface with a 4-port FIFO coupled to a 32-bit AMBA Trace Bus (ATB) for transmitting the trace data.

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 60 depicts the high level block diagram of the debug components and how they integrate with the VCPU.

### 10.1.1 VSPA debug block diagram

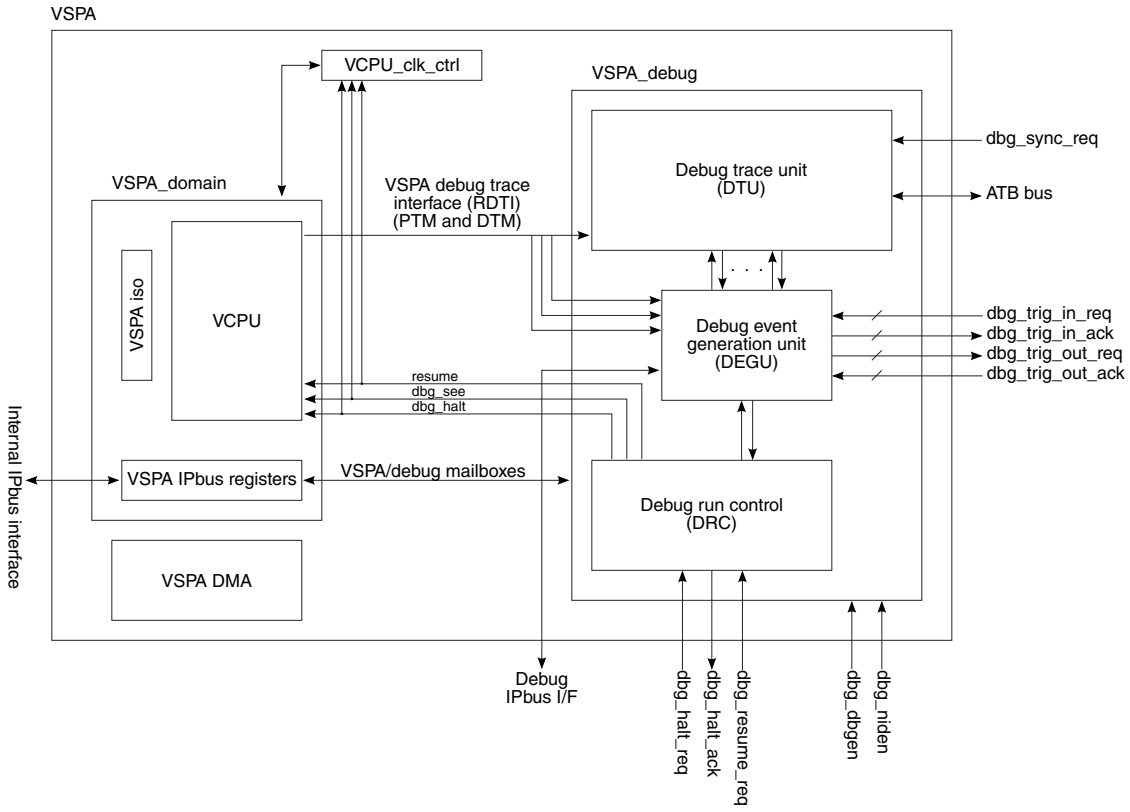



**Figure 60. VSPA debug block diagram**

### 10.1.2 Debug functional description

#### 10.1.2.1 Debug trace unit (DTU) subblock

The DTU block is responsible for generating real-time trace of the VCPU. It will monitor the behavior of the VCPU via the VSPA debug trace interface (RDTI) bus, as well as events generated by the DEGU unit, to generate predefined messages that external trace tools can use to reconstruct VCPU program execution. Refer to VSPA messages for detailed information on the messages produced by the DTU. As illustrated in the figure below, the generated messages are queued in a 4-port message FIFO, capable of writing one or more messages from each trace source in a single VSPA clock cycle. This eliminates collisions between trace sources and thus eliminates the need for any form of error message indicating "lost message due to collision". The 4 ports have a priority structure providing PTM the highest priority with DTM having the lowest. This priority mechanism establishes the temporal order of the messages in the FIFO, if multiple trace sources generate a message in the same clock cycle.

The FIFO drains to a standard 32-bit ARM ATB bus and adheres to the ATB ATVALID/ATREADY, AFVALID/AFREADY signal protocol. Note that unlike normal trace messages, the trace alignment marker (TAM) is directly inserted in the ATB trace stream by the ATB interface.

The DTU unit will make use of multiple DEGU resources (address comparators, and cross-triggers) which can be configured to start and stop program trace (PTM) and/or data trace (DTM). This can be done either at single specific instruction/data addresses or by combining pairs of single address comparators to define a window in which program and/or data trace is either enabled or disabled (include/exclude window).

The configuration bits in the TC1 control register exist to independently enable/disable the PTM and/or DTM trace units, and also configure whether hardware events/triggers can dynamically control trace start/stop of PTM and/or DTM. The WPM and DQM

trace sources are also individually enabled/disabled via corresponding bits in the TC1 register, but do not have the capability to be dynamically triggered on/off. Note that the DTU unit can be disabled entirely by clearing the GDBEN.nidbg_en bit.

The trace output generated by the DTU may be disabled (overridden), irrespective of the GDBEN.nidbg_en bit setting, by the external debug system as deemed necessary from a security perspective by driving the dbg_niden input low (0).

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.

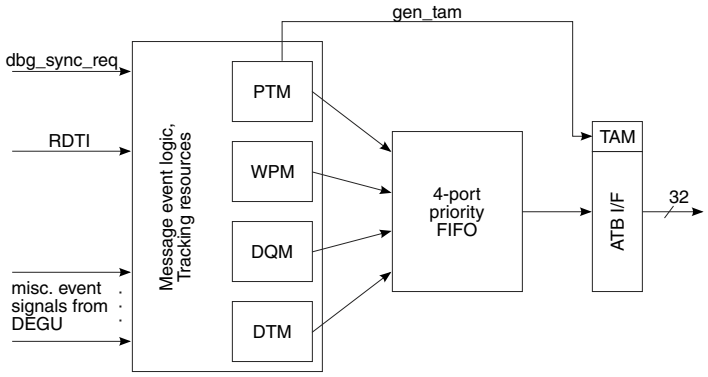



**Figure 61. DTU block diagram**

#### 10.1.2.1.1 Program trace message (PTM) source

The program trace message unit is responsible for generating indirect branch (PTIBw/sync message), address synchronization (SYNC message), and program correlation (predicate and branch history information; hardware loop correlation information) messages, which allows the external debug system to trace the VCPU program flow. The control and data signals from the RDTI are used to trigger the generation of these messages. The Program SYNC message can be produced either periodically (when the TAM is generated) or upon request from the external system via the dedicated dbg_sync_req input. In addition, the SYNC message can also be produced due to an event from the DEGU unit (which was configured to cause the action of generating an address synchronization).

#### 10.1.2.1.2 Watchpoint message (WPM) source

The WPM unit is responsible for producing watchpoint messages based on events generated in the DEGU. On occurrence of any DEGU event, which is configured as having the action of producing a WPM, the WPM unit will produce a watchpoint message if the WPM trace source is not disabled in the TC1 control register. There are upto 20 watchpoint sources as illustrated in Table 110.

Note that any of the DEGU events may be configured to cause the action of generating a WPM through the WPMACR register.

#### 10.1.2.1.3 Data acquisition message (DQM) source

The DQM unit is responsible for producing data acquisition messages, which are useful for observing the architectural state of the machine through software instrumentation, without the need to halt the core. This is accomplished by writing to a set of DQM registers located on the VSPA IPbus. Currently two forms exist, a small form (1 32-bit register), and a long form (set of 2 32-bit registers). Refer to the VSPA DQM registers for details. Whenever these registers are written, a DQM message is generated.

#### 10.1.2.1.4 Data trace message (DTM) source

The DTM unit is responsible for producing data trace messages from comparator events (which are configured to cause the action of generating a DTM) in the DEGU unit. Only the VCPU DMEM memory is allowed as a source for data trace messaging. A DEGU comparator configured for matching VSPA IPbus address or IPPU DMEM address with comparator event configured to cause the action of generating a DTM would produce unpredictable data in the message. Other scenarios, accessing VCPU DMEM, would cause unpredictable data in the DTM messag are descriced in table VCPU DMEM DTM data validity.

**Table 106. VCPU DMEM DTM data validity**

| Comparator address match configuration | Memory access type | Validity of DTM data |
|---|---|---|
| Single address | single word access | Valid DTM message data generated. |
| Address range | single word access | Valid DTM message data generated. |
| Single address | full line access | Valid DTM message data generated. |
| Address range | full line access | Unpredictable message data generated in DTM message. In this case, multiple addresses match the comparator criteria; thus, there is no way to select a single 32-bit data element for the DTM. |

#### 10.1.2.2 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 107. 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. |
| 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.

##### 10.1.2.2.1 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.

#### 10.1.2.2.2 Cross-trigger resources

The DEGU contains 8 general purpose cross-trigger inputs and 8 cross-trigger outputs. Any of the DEGU events can be configured to cause any of the *dbg_trig_out_req[n]* (where n is 0-7) 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:

- Start/stop of program and/or data trace, by sending the appropriate control signal(s) to the DTU unit. Note that the PTM and/or DTM units must be enabled for trace start/stop via hardware in the TC1 control register.
- Halt/resume the VSPA, by sending the appropriate control signal(s) to the DRU unit to perform the halt/resume function.
- Generation of WPM and/or SYNC messages by sending the WPM and/or SYNC event signals to the DTU unit.

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 0-7) 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.

#### 10.1.2.2.3 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.

### 10.1.2.3 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.

#### 10.1.2.3.1 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.

#### 10.1.2.3.2 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.

#### 10.1.2.3.3 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.

### 10.1.2.4 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 62.

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.

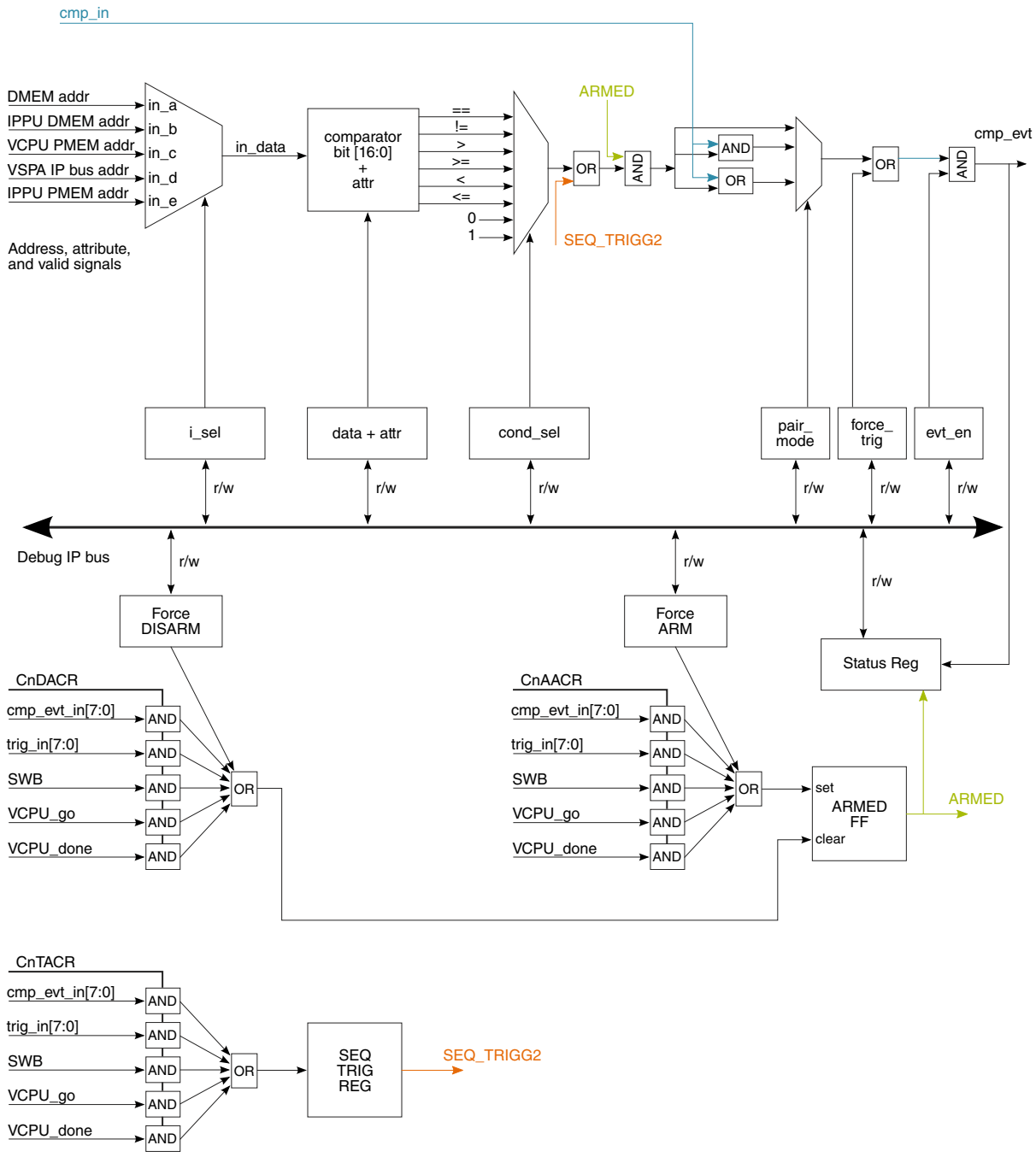



**Figure 62. Debug comparator and sequencer**

### 10.1.2.5 Timestamps

VSPA can output timestamp messages (TSM) to the trace message stream. This adds the capability to indicate the order of events and provides a way to correlate events in a multi-VSPA environment.

The VSPA timestamp message uses a 48-bit timestamp which is provided by a centralized timestamp generator on the SoC. The timestamp can be coherently sampled and read by the debugger by performing a read access to the TSMCNT_U (upper bits) register first followed by a read of the TSMCNT_L (lower bits) register.

VSPA can send two forms of the TSM message:

- Short TSM: 32-bit Message, includes 24-bit Timestamp value.
- Long TSM: 64-bit message, includes 48-bit Timestamp value.

The format of the Timestamp messages is described in Figure 63. For a description of the timestamp message formats, refer to Timestamp message (TCODE=2).

Each trace message has a TS (Timestamp) bit. The TS bit indicates the message was sent concurrently with a timestamp message (TSM). Note that while both messages are created and sent to the fifo at the same time, the TSM message will appear first on the ATB, followed by messages with the TS bit set. A set TS bit in a message means the message correlates to a preceding TSM.

The following conditions, when enabled, will cause a TSM message to be set:

- Taken JSR instruction
- Taken RTS instruction
- Start Loop instruction
- End Loop
- SoC synchronization request
- Timestamp tap point rollover (see TSMCR[15:12])
- Any DTM, PTM SYNC, DQM, WPT trace messages

Whenever possible, the short form of TSM is sent. Under certain conditions, the long form of TSM is transmitted. Long TSM message is sent if any of the following conditions occur:

- SoC synchronization request,
- Periodically due to timestamp tap point rollover or trace alignment message,
- Multiple TTYPE conditions occur at the same time,
- FIFO overflow exit.

All other events that cause a TSM to be sent will result in a short TSM.

Timestamp is enabled when non-invasive debug mode enabled and the TSM_EN bit is set.

### 10.1.3 VSPA messages

VSPA trace messages will be output on a standard 32-bit ATB bus. All messages will be transferred in either one or two 32-bit beats, indicated by the 1-bit EXTensioN field (EXTN). For convenience the Nexus IEEE-ISTO 5001 standard TCODE field will be used for encoding the message type. All fields that make up the VSPA trace messages will be fixed width. The following table lists the currently supported messages:

**Table 108. VSPA trace messages**

| Message type | Description | Width Bits | Num Beats |
|---|---|---|---|
| Special Purpose | | | |
| TAM | Trace Alignment Marker. Identifies the start of trace data and provides 32-bit word and message boundary alignment. | 64 | 2 |

*Table continues on the next page...*

**Table 108. VSPA trace messages (continued)**

| Message type | Description | Width Bits | Num Beats |
|---|---|---|---|
| Watchpoint Messages (WPM) | | | |
| WPM | Watchpoint Message - simple occurrence of up to upto 20 fixed events. | 32 | 1 |
| Program Trace Messages (PTM) | | | |
| SYNC | Program Trace Synchronization Message - debug/low-power mode entry/exit; FIFO overflow; program/data trace start/stop; periodic sync. | 32 | 1 |
| PTIBw/sync | Program Trace Indirect Branch Message with sync - taken indirect/direct branch, rts, VCPU "Go". | 32 | 1 |
| PCM Hist | Program Correlation Message - predicate/branch history. | 32 | 1 |
| PCM Loop | Program Correlation Message - hardware loops. | 32 | 1 |
| Data Acquisition (DQM) | | | |
| DQM Short | Data Acquisition Message - outputs data written to DQM register. | 32 | 1 |
| DQM Long | Data Acquisition Message - outputs data written to DQM registers. | 64 | 2 |
| Data Trace Messages (DTM) | | | |
| DTM WR w/sync | Data Trace with sync (full 17-bit address) - write transaction. | 64 | 2 |
| DTM WR | Data Trace with (no address) - write transaction. | 32 | 1 |
| DTM RD w/sync | Data Trace with sync (full 17-bit address) - read transaction. | 64 | 2 |
| DTM RD | Data Trace with sync (no address) - read transaction. | 32 | 1 |
| TimeStamp Messages (TSM) | | | |
| TSM Short | Short Timestamp Message with 24-bit timestamp value. | 32 | 1 |
| TSM Long | Long Timestamp Message with 48-bit timestamp value. | 64 | 2 |

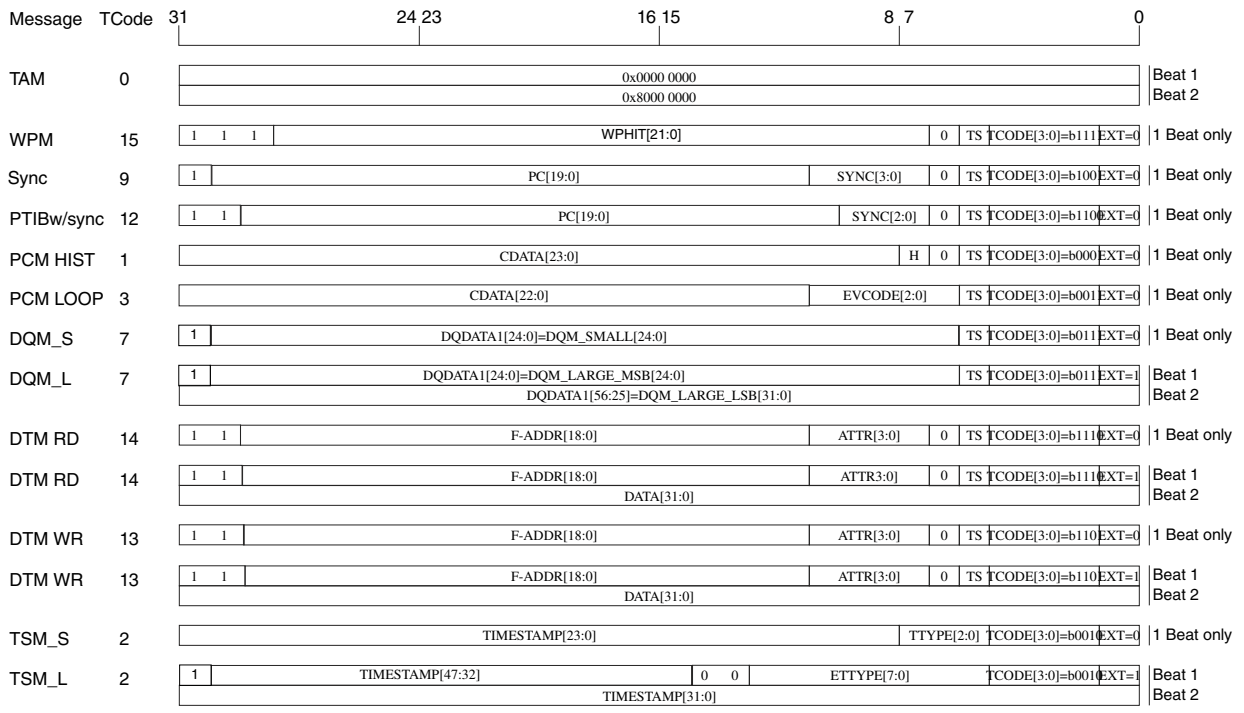



**Figure 63. Trace Message Format**

The following sections provide a detailed description of the VSPA trace messages and the fields that make them up.

### 10.1.3.1 Trace alignment marker

The VSPA trace alignment marker (TAM) is a 64-bit data item (7 consecutive bytes of zeros, 8th byte 0x80) which the trace interpreter/decompressor can uniquely identify in a given trace stream for determining the start of trace as well as word and message boundary alignment. Once this alignment marker is confirmed, the trace interpreter/decompressor can expect that the following 32-bit piece of data is the start of a valid VSPA trace message.

Given that no possible VSPA trace message can contain 63 consecutive zeros, this simple alignment value is guaranteed to be unique within a VSPA trace stream, and is also compatible with ARM CoreSight ETM architecture. This marker will be directly output to the ATB interface and will not be queued in the FIFO as with other trace messages. Refer to Table 109 for a summary of TAM generation conditions. The format of the trace alignment marker is described in Figure 63.

**Table 109. TAM generation conditions**

| Reason | Notes |
|---|---|
| First enabled trace unit | The TAM will be output to the ATB immediately after any trace unit (PTM, WPM, DQM or DTM) becomes enabled via the TC1 register. The TAM is always emitted directly to the ATB and is never placed in the FIFO.<br>**NOTE**<br>A TAM is NOT produced when every trace unit becomes enabled. Rather, the TAM is produced only when the FIRST trace unit becomes enabled. |

*Table continues on the next page...*

**Table 109. TAM generation conditions (continued)**

| Reason | Notes |
|---|---|
| Periodic | Generated after a programmable number of trace messages have been generated. In this case, the ATB interface will pause a cycle from accessing the FIFO, in order to insert the TAM directly on the ATB bus. |
| SoC sync request | Upon an external sync request from the SoC and/or development tool, or on occurrence of an event configured to produce a SYNC message, the TAM will pend to be directly output on the ATB. A PTM SYNC message will also be generated (if PTM is enabled) and will enter the FIFO as usual. The development tool will need to be aware that the corresponding SYNC message will follow the TAM, with potentially many interleaving trace messages (depending on the FIFO state at the time of the synchronization request).<br>**NOTE**<br>If PTM is not enabled, then only the TAM will be produced. |
| TSM rollover | Upon TSM tap point rollover when TSM is enabled. |

The following fields are common to all remaining VSPA trace messages and are described here:

- TCODE - Transfer code. This 4-bit field uniquely identifies the message type and therefore, the number and type of fields that make up the message. Note that all VSPA message fields are fixed width.
- EXTN - Extension. This field specifies the number of 32-bit beats required to transmit the given message on the ATB bus: one 32-bit beat (EXTN = 0) or two 32-bit beats (EXTN = 1).
- TS - Timestamp. This 1-bit field indicates an associated timestamp message (TSM) was created and sent at the time this message was created.
  - 0: No TSM was generated.
  - 1: A TSM was also created.

### 10.1.3.2 Watchpoint message (TCODE=15)

The VSPA watchpoint message (WPM) includes up to 19 fixed watchpoint sources. When any of the possible sources assert, a watchpoint message is generated and sent to the queue for eventual output. For VSPA debug, watchpoint inputs are positive edge sensitive and a watchpoint message will be generated only on cycles in which the pos-edge assertion of one or more watchpoint sources is detected. Because of this, any event(s) which remains asserted on back-to-back cycles will only produce a single watchpoint message. The format of the Watchpoint message is described in Figure 63.

When an event occurs and the corresponding bit in WPMCR is set, a WPM message is generated. The WPM message will indicate ALL the events that occurred at the time the message was fired (including comparators that fired, and trigger_in rising edge, even if the WPMCR bit is cleared).

WPHIT - This 19-bit field (also known as the watchpoint payload) indicates which watchpoint source(s) have triggered on the current cycle. The following table details the format of this field. Refer to Debug WPM Generation Action Control register (WPMACR) for the list of actual watchpoint sources and to configure the events which cause a WPM.

**Table 110. Watchpoint Source Description**

| WPHIT | Description | Enable Bit | Source |
|---|---|---|---|
| XXXXXXXXXXXXXXXXXX XX1 | Watchpoint 0 | WPMACR[0] | Comparator 0 match event |

*Table continues on the next page...*

**Table 110. Watchpoint Source Description (continued)**

| WPHIT | Description | Enable Bit | Source |
|---|---|---|---|
| XXXXXXXXXXXXXXXXXXX X1X | Watchpoint 1 | WPMACR[1] | Comparator 1 match event |
| XXXXXXXXXXXXXXXXXXX1 XX | Watchpoint 2 | WPMACR[2] | Comparator 2 match event |
| XXXXXXXXXXXXXXXXXX1X XX | Watchpoint 3 | WPMACR[3] | Comparator 3 match event |
| XXXXXXXXXXXXXXXXX1XX XX | Watchpoint 4 | WPMACR[4] | Comparator 4 match event |
| XXXXXXXXXXXXXXXX1XXX XX | Watchpoint 5 | WPMACR[5] | Comparator 5 match event |
| XXXXXXXXXXXXXXX1XXXX XX | Watchpoint 6 | WPMACR[6] | Comparator 6 match event |
| XXXXXXXXXXXXXX1XXXXX XX | Watchpoint 7 | WPMACR[7] | Comparator 7 match event |
| XXXXXXXXXXXXX1XXXXXX XX | Watchpoint 8 | WPMACR[16] | dbg_trig_in_req[0] event |
| XXXXXXXXXXXX1XXXXXXX XX | Watchpoint 9 | WPMACR[17] | dbg_trig_in_req[1] event |
| XXXXXXXXXXX1XXXXXXXX XX | Watchpoint 10 | WPMACR[18] | dbg_trig_in_req[2] event |
| XXXXXXXXXX1XXXXXXXXX XX | Watchpoint 11 | WPMACR[19] | dbg_trig_in_req[3] event |
| XXXXXXXXX1XXXXXXXXXX XX | Watchpoint 12 | WPMACR[20] | dbg_trig_in_req[4] event |
| XXXXXXXX1XXXXXXXXXXX XX | Watchpoint 13 | WPMACR[21] | dbg_trig_in_req[5] event |
| XXXXXXX1XXXXXXXXXXXX XX | Watchpoint 14 | WPMACR[22] | dbg_trig_in_req[6] event |
| XXXXXX1XXXXXXXXXXXXX XX | Watchpoint 15 | WPMACR[23] | dbg_trig_in_req[7] event |
| XXXXX1XXXXXXXXXXXXXX XX | Watchpoint 16 | WPMACR[29] | Software breakpoint event |
| XXXX1XXXXXXXXXXXXXXX XX | Watchpoint 17 | WPMACR[30] | VCPU "Done" event |

*Table continues on the next page...*

**Table 110. Watchpoint Source Description (continued)**

| WPHIT | Description | Enable Bit | Source |
|---|---|---|---|
| XXX1XXXXXXXXXXXXXXXXX | Watchpoint 18 | WPMACR[31] | VCPU "Go" event |
| XX1XXXXXXXXXXXXXXXXX | Watchpoint 19 | WPMACR[28] | "Illegal Instruction Trap" event |
| X1XXXXXXXXXXXXXXXXX | Watchpoint 20 | | Protection fault |
| 1XXXXXXXXXXXXXXXXXX | Watchpoint 21 | | IRQ input event20 |

#### 10.1.3.3 SYNC - program trace synchronization message (TCODE=9)

The Program trace synchronization message (SYNC) is used at multiple events and provides the development tools the full PC address at various times for program flow synchronization purposes. Note that the LSB (bit 0) of the provided 17-bit value can be thought of as a virtual address bit (as the VCPU Program Memory- PMEM - address width is only 16-bits). LSB (bit 0) equal to 1 indicates the given PC value is the address of an upper half word instruction, whereas an LSB equal to 0 indicates the PC value is either a lower half word or full-word instruction address. The format of the SYNC message is described in Figure 63.

SYNC - This 4-bit field indicates which synchronization condition occurred to cause the generation of the PTM Sync message. Refer to Table 111. Note that if multiple events occur at the same time (the same clock cycle), only the highest priority event is reflected in the SYNC code (higher priority has lower SYNC code).

PC - This is the full 17 bits of the instruction address (PC value).

**Table 111. SYNC Encoding**

| SYNC Value | SYNC Condition | Description |
|---|---|---|
| 0000 | Error - FIFO Overrun | Indicates the FIFO has overflowed. Message contains the PC value at the time the FIFO is re-enabled. |
| 0001 | VCPU "Done" | Indicates VCPU has completed a task and will go dormant (until the next "Go" event). |
| 0010 | Entry to Debug | Indicates VSPA has been halted in debug mode. |
| 0011 | Trace Stop | Indicates the address when instruction trace was disabled while the VCPU continues to execute instructions (either via *stop trace* trigger, or a write to the instruction trace enable bit in the TC1 register). |
| 0100 | VCPU "Go" | Indicates the VCPU has started execution (trace already enabled). This will provide the full PC value of the start address. |
| 0101 | Exit from Reset | Exit from reset. Note that future versions of the VCPU may start at addresses other than 0. Can also be used as an indicator that the VCPU has exited reset, as well as determining how long it takes to load code and start the VCPU. |
| 0110 | Exit from Debug | Indicates VSPA has exited from debug mode (resume execution). |

*Table continues on the next page...*

**Table 111. SYNC Encoding (continued)**

| SYNC Value | SYNC Condition | Description |
|---|---|---|
| 0111 | Trace Start | Indicates the start address when instruction trace was enabled while the VCPU is already executing instructions or halted in debug mode. The enable can be via a write to the instruction trace enable bit in the TC1 register or on occurrence of a start trace trigger. The occurrence of this event will also cause a TAM to be generated, if not already generated as a result of another trace unit message. |
| 1000 | External SYNC Request | External dbg_sync_req input was driven (synchronization request). A trace alignment message (TAM) will be directly placed on the ATB bus and the SYNC message (if PTM is enabled) will be queued in the FIFO. Given this, multiple messages could potentially be output between the TAM and the PTM SYNC message. |
| 1001 | Debug event generated SYNC | A debug event configured to generate trace synchronization occurred (Refer to the program trace sync action control register - PTSACR for details). A trace alignment Message (TAM) will be directly placed on the ATB bus and the SYNC message (if PTM is enabled) will be queued in the FIFO. Given this, multiple messages could potentially be output between the TAM and the PTM SYNC message. |
| 1010 | Periodic | Periodic TAM counter resource has reached zero, causing a TAM to be inserted into the trace stream, followed by a SYNC message (if PTM is enabled) being queued in the FIFO for eventual output. |
| 1011-1111 | Reserved | |

#### 10.1.3.4 Program trace indirect branch with sync message (TCODE=12)

The program trace indirect branch with synchronization message (PTIBw/sync) provides the development tools the full PC address on change of program flow events such as taken branches and jump to/return from subroutines. The format of the PTIBw/sync message is described in Figure 63.

SYNC - This 3-bit field indicates which synchronization condition occurred to cause the generation of the PTIBw/sync message. Refer to the following table for more information.

PC - This is the 20-bit PC target address.

**Table 112. PTIB SYNC Encoding**

| SYNC Value | SYNC Condition | Description |
|---|---|---|
| 000 | Taken Indirect branch | Indicates taken indirect branch occurred.<br>jsr(.cc) gX<br>jmp(.cc) gX<br>rts<br>When return stack compression is disabled, a PTIBw/sync message is always sent for rts. When return stack compression is enabled, since the VCPU's return address stack is always "well behaved", a PTIBw/sync message is only sent if trace is enabled after a return address has already been pushed onto the stack (that is, when a pop occurs and trace was not enabled to see the push). The option to enable return stack compression would be enabled by the RSE bit in the TC1 configuration register. |

*Table continues on the next page...*

**Table 112. PTIB SYNC Encoding (continued)**

| SYNC Value | SYNC Condition | Description |
|---|---|---|
| 001 | Taken direct branch | Indicates a taken direct branch occurred.<br>jsr(.cc) #I<br>jmp(.cc) #I<br>Normally both taken and not-taken direct branches would be tallied as HISTORY bits (refer to the PCM message). The option to have taken direct branches generates a PTIBw/sync message is enabled by the DBSUE bit in the TC1 configuration register. |
| 010-111 | Reserved | Reserved |

### 10.1.3.5 PCM HIST - program correlation message history (TCODE=1)

The program correlation history (PCM HIST) message is used to correlate processor events to instructions in the program flow. Program Correlation messages provide branch/predicate history at the time the event is detected.

In the VCPU, the loop program correlation message will also be used for providing development tools with hardware loop (refer to VSPA Instruction Set Manual for details on hardware loop instructions) execution.

The format of the program trace - program correlation message is described in Figure 63.

H-CODE - This 1-bit field indicates which event the Program Correlation message was generated for. Refer to the following table for H-CODE details.

**Table 113. History Event Code (H-CODE)**

| H-CODE | Description | CDATA |
|---|---|---|
| 0 | HISTORY buffer flush, if not empty and evcoce_mask[0] is not set, when any of the following occur:<br>Entry in debug mode.<br>Program trace disable.<br>Execution of VCPU "done" instruction.<br>Taken indirect branch.<br>Taken direct branch, when dbsue bit in the TC1 register is set.<br>When the dbsue bit is cleared, both taken and not-taken direct branches would be tallied as HISTORY bits – 0 for non-taken branch and 1 for taken branch.<br>When the dbsue bit is set, only taken direct (excluding non-taken) branches would result in a PTIB messages. When dbsue bit is set, no HISTORY bit is tallied in the history buffer. When the PTIB message is sent, the History buffer is flushed (if not empty), and a PCM HIST message is sent with the H-CODE=0. | HISTORY buffer. |

*Table continues on the next page...*

**Table 113. History Event Code (H-CODE) (continued)**

| H-CODE | Description | CDATA |
|---|---|---|
| | Any generated SYNC message, whether by external request (dbg_sync_req pin), periodic, or debug event (refer to the PTSACR register).<br>Start of Hardware loop (loop_start instruction). | |
| 1 | HISTORY buffer full. | HISTORY buffer. |

CDATA - This 24-bit field is the data associated with the EVCODE. It is branch/predicate history information.

PCM for branch/predicate history - The branch/predicate HISTORY buffer stores information about branch and predicate instruction execution. The buffer is implemented as a left-shifting register. The buffer is preloaded with a one (1) which acts as a stop bit (the most significant 1 in the history field is a termination bit for the field and not an inherent part of the history data).

A value of one (1) is shifted into the history buffer for each taken direct branch, taken indirect branch (when the target address matches the address being tracked on the return stack, if implemented), or non-branch conditional instruction whose condition evaluates to TRUE.

A value of zero (0) is shifted into the history buffer for each non-taken branch (including indirect branch instructions) or conditional instructions whose condition evaluates to FALSE. VCPU implements a 24-bit history buffer (23 history bits plus 1 stop bit). Refer to the following table for more information.

**Table 114. Branch/Predicate History Events**

| Branch/Predicate History Event | History Bit | Relevant Instructions | Notes |
|---|---|---|---|
| Not-taken indirect branches | 0 | jsr(.cc) gX<br>jmp(.cc) gX | |
| Not-taken direct branches | 0 | jsr(.cc) #I<br>jmp(.cc) #I | |
| Taken direct branches | 1 | | If the DBSUE configuration bit is set in the TC1 control register, taken direct branches will generate a PTIBw/sync message and will not be logged in the HISTORY buffer. |
| Taken indirect branches | 1 | rts | If the RSE configuration bit is set in the TC1 control register when return stack compression is enabled, rts instructions (which can be tracked on the return stack), are logged in the HISTORY buffer. When trace is enabled after return addresses have already been pushed on the stack, the indirect branch for those return instructions are not logged in the history buffer, but will generate a PTIBw/sync message instead (since the address pushed on the stack is not known). |
| Not executed conditional non-branch instructions | 0 | mpy.cc, div.cc,<br>mod.cc, sr.cc, | |

*Table continues on the next page...*

**Table 114. Branch/Predicate History Events (continued)**

| Branch/Predicate History Event | History Bit | Relevant Instructions | Notes |
|---|---|---|---|
| Executed conditional non-branch instructions | 1 | add.cc, sub.cc,<br>cmp.cc, sl.cc,<br>and.cc, or.cc,<br>xor.cc, bclr.cc,<br>mv.cc, not.cc | |

### 10.1.3.6 PCM LOOP - program trace-program correlation message LOOP (TCODE=3)

The program correlation loop (PCM LOOP) message is used for providing program flow trace information on hardware loop execution (refer to the *VSPA Instruction Set Manual* for details on hardware loop instructions). The format of the Program Trace - Program Correlation message is described in Figure 63.

EVCODE - This 4-bit field indicates which event the program correlation message was generated for. Refer to Table 115.

CDATA - This 20-bit field is the data associated with the EVCODE. In the case of hardware loops, this 20-bit field is further sub-divided as follows: bits [19:16] provide loop nesting level information, while bits [15:0] provide either the loop iteration count since PTM trace was enabled, or the 16 MSBs of address. Note that the 17th bit (LSB) is not provided, as it is always 0 for these EVCODEs. Furthermore, on completion of loops of length one, the nesting level bits [19:16] will be set to 1111 and to 1110 on completion of loops of length two.

**Table 115. Event Code (EVCODE)**

| EVCODE | Description | CDATA[22:0] |
|---|---|---|
| 001 | Start of VCPU hardware loop (loop_begin instruction executed for most recent set.loop instruction) for loops of length >=3. | Address of beginning of hardware loop for loops of length greater than 2. This is the address of the macroinstruction containing the loop_begin microinstruction.<br>Note that this message will be generated on the first occurrence of the loop_begin microinstruction after program trace is enabled or after the FIFO has been emptied due to a FIFO overflow condition. |
| 010 | Start of VCPU hardware loop (loop_begin instruction executed for most recent set.loop instruction) for loops of length one. | Address of hardware loop for loops of length one. This is the address of the single macroinstruction containing the loop_begin microinstruction of the loop. Note that this message is only generated when program trace is enabled on the first occurrence of the loop beginning (1st iteration). In addition, if trace is enabled after the loop has already started, this message will not be generated. |
| 011 | Start of VCPU hardware loop (loop_begin instruction executed for most recent set.loop instruction) for loops of length two. | Address of hardware loop for loops of length two. This is the address of the first of two macroinstructions of the loop. Note that this message is only generated when Program Trace is enabled on the first occurrence of the loop beginning (1st iteration). In addition, if trace is enabled after the loop has already started, this message will not be generated. |

*Table continues on the next page...*

**Table 115. Event Code (EVCODE) (continued)**

| EVCODE | Description | CDATA[22:0] |
|---|---|---|
| 100 | End of VCPU hardware loop (last macroinstruction of the loop indicated by the loop_end instruction) for loops of length >=3. | Address of last macroinstruction (wrap point indicated by the loop_end microinstruction) of hardware loop for loops of length three or greater. Note that this message is generated on the first occurrence of the end of the loop (wrap point denoted by loop_end microinstruction). In addition, if trace is enabled after the loop has already started, this message will be generated prior to the corresponding start-of-loop message (EVCODE 0001). In addition, this message is never produced for loops of length one or two. |
| 101 | Completion of VCPU hardware loop (loop completion, whether terminated early - loop_stop - or not) for loops of length >=3. | Loop count at hardware loop completion. Note that this count only represents the number of times the loop iterated while PTM trace was enabled. |
| 110 | Completion of VCPU hardware loop (loop completion) for loops of length one or two. | Loop count at hardware loop completion. Note that this count only represents the number of times the loop iterated while PTM trace was enabled. |
| 111 | loop break | VSPA executed loop break. |

PCM for hardware loops - In the general case of VCPU hardware loops, a PCM will be generated upon execution of a loop_begin instruction with EVCODE indicating start of hardware loop and CDATA providing the address of the first macroinstruction in the loop (which includes, or is, the loop_begin microinstruction). At the end of the loop (wrap point) another PCM with EVCODE indicating end of hardware loop and CDATA providing the address of this last macroinstruction of the loop is generated. Finally, when the loop has completed, a PCM with EVCODE indicating loop completion and CDATA providing the loop iteration count, is output.

For loops of length 1 or 2, the start of hardware loop of length one/two PCM (EVCODE=0010/0011) will be generated and the loop completion PCM (EVCODE=0110) will be generated at completion of the loop as usual. However, the end (wrap point) of hardware loop PCM will NOT be generated since the end address is easily calculated from the start of hardware loop PCM address.

In the case PTM trace is enabled (either manually or triggered "on" via hardware event) while the hardware loop has already begun, the PCM messages change slightly. In this scenario:

- The start of hardware loop PCM will be generated upon the first observed (trace enabled) execution of the start of the loop. This implies that this PCM would follow the end of hardware loop PCM. In other words, an end-of-loop PCM will be sent followed by a start-of-loop PCM.
- The loop count reported by the loop completion PCM would reflect only the number of loop iterations which were executed during the period in which trace was enabled, not the total number of iterations the loop actually executed (# iterations prior to trace enable + # iterations after trace enabled).
- In the special case trace is enabled during the final iteration of the loop, both the start of hardware loop and end of hardware loop PCMs are omitted entirely. The special case loop completion PCM, with EVCODE=0101, is used to indicate CDATA is providing the address of the last macroinstruction in the loop. In this case, the actual loop count is implied to be one.

Table 116 summarizes the conditions under which a PCM is generated around hardware loop execution, taking into account the trace enable state at the time of event. In this table, the "Seen" column corresponds to whether trace was enabled (either manually or triggered via hardware event), when the instruction(s) in the "Relevant Instructions" column were executed (Seen=0 indicates trace was off, Seen=1 indicates trace was on).

In the VCPU, hardware loops may be nested up to 8 levels deep. In the case of nested loops, the PCM messaging for the inner loop will be generated as described above. When the end of the next-level outer loop is encountered, the end of hardware loop PCM gets generated for that loop, followed by the start of hardware loop PCM message, after it wraps back to the beginning. (in the case that trace was enabled after the start of this outer-level loop, otherwise the PCM message for the start of this outer loop would have been generated in the normal way). When the inner loop is once again encountered, normal loop-related PCM messages are generated for this inner loop, as described earlier. When this next-level outer loop completes, the loop completion PCM is generated providing the loop count for this next-level outer loop during the period of time trace was enabled. This process continues for each level of nesting as the loop structure naturally unwinds. Note that all normal program trace messages (SYNC, PTIBw/sync, and PCM history messaging) continue to be generated as normal during the body of any given loop. Hardware looping fails when loop nesting reaches 8, and the hardware loop trace only fails because of the loop trace failure.

**Table 116. Hardware Loop PCM Generation Conditions**

| PCM Generation | Seen | Relevant Instructions[1] | Notes |
|---|---|---|---|
| PCM generated (once) at start of loop, providing address of first macroinstruction in the hardware loop.<br>Timing of this "loop start" PCM differs depending on whether trace was on or off when the loop started execution.<br>Note this PCM is generated for all loop sizes (1, 2, >=3). | 0 | set.loop ITER_CNT_1, LOOP_SIZE_R10_1<br>set.loop Iu10, Iu17, Iu17<br>set.loop AS, LOOP_SIZE_R10_1<br>set.loop ITER_CNT_1<br>set.loop AS<br>loop_begin | If PTM trace becomes enabled while currently in hardware loop (set.loop/loop_begin were unobserved), a PCM with EVCODE=00010, and CDATA providing address of first macroinstruction of loop (which includes the loop_begin microinstruction). For loop size > 2, this PCM will be sent on the first occurrence of the observed execution of the beginning of the loop (the point at which the loop has wrapped, and is beginning the next iteration). |
| | 1 | | If PTM trace was enabled at the start of the loop (set.loop/loop_begin were observed), The PCM will be generated immediately at the beginning of the loop (as opposed to after wrapping as in the previous scenario above).<br>In either scenario, this PCM (EVCODE=00010) will only be generated once (on first observed execution of beginning of loop).<br>Note in the case PTM is enabled during the final iteration of the loop, this PCM (EVCODE=00010) will NOT be generated. |
| PCM not generated | 0 | loop_stop | PCM is never generated at time of loop_stop execution, whether trace was enabled or disabled. |
| | 1 | | |
| PCM generated (once) at end of loop (wrap point).<br>Note this PCM is generated only for loops of length 3 or greater (except if trace is enabled during the final iteration of a loop. Refer to the "Notes" column for additional | 0 | set.loop ITER_CNT_1, LOOP_SIZE_R10_1<br>set.loop Iu10, Iu17, Iu17<br>set.loop AS, LOOP_SIZE_R10_1<br>set.loop ITER_CNT_1<br>set.loop AS<br>loop_begin | PCM is generated indicating end of loop (wrap point), EVCODE=00101, and CDATA providing address of the last macroinstruction in the hardware loop. This PCM (EVCODE=00101) will only be generated once (on first observed occurrence of loop wrap point).<br>Note in the case PTM is enabled during the final iteration of the loop, this PCM will not be generated. Rather, a "loop complete" PCM, with EVCODE=00111 and CDATA providing the |
| | 1 | | |

*Table continues on the next page...*

**Table 116. Hardware Loop PCM Generation Conditions (continued)**

| PCM Generation | Seen | Relevant Instructions[1] | Notes |
|---|---|---|---|
| information on this special case scenario). | | | address of the last macroinstruction of the loop will be generated. This is a special case and differs from the normal loop completion message (EVCODE=00110), where CDATA provides loop iteration count during the period of time PTM trace was enabled. Refer to EVCODE 00111. |
| PCM message generated at loop completion. | 0 | set.loop ITER_CNT_1, LOOP_SIZE_R10_1<br>set.loop Iu10, Iu17, Iu17<br>set.loop AS, LOOP_SIZE_R10_1<br>set.loop ITER_CNT_1<br>set.loop AS<br>loop_begin | PCM with EVCODE=00110, and CDATA providing the number of iterations the loop executed during the time trace was enabled.<br>Note a special case exists when trace is enabled during the final loop iteration. In this case, CDATA will provide the address of the last macroinstruction in the hardware loop, and the count will implicitly be understood to be one. EVCODE=00111 will be used to indicate this condition. |
| | 1 | | |

1. See *VSPA Instruction Set Manual* for instruction details.

### 10.1.3.7 Data acquisition message (TCODE=7)

The data acquisition (DQM) message is useful for the external development tool to observe the architectural state of the target through software instrumentation. This is achieved by performing one or more writes to one or more dedicated internal DQM register(s), the value of which is inserted into the data acquisition message. There exists two forms of the DQM message-a short data profile (performed in 1 32-bit beat) and a long data profile (performed in two 32-bit beats).

The short form is generated by a write to a single 32-bit VSPA IPbus register (indicated by EXTN=0). The 25 LSBs of this 32-bit value will be placed in DQDATA field in a single beat message.

The long form is generated after performing a pair of 32-bit writes to a set of VSPA IPbus registers (indicated by EXTN=1). Similar to the short form, the 32-bit write to the first of this set of registers will result in the 25 LSBs of this data value being placed in the DQDATA field of the 1st beat. The 32-bit write to the second register in this set actually causes the DQM message to be generated. In this case, this 32-bit write will result in all 32-bits of data being stored in the second beat (full 32-bit DQDATA) of this message.

In summary, the result of the short form of this message will be a single-beat message containing 25-bits of data - the 25 LSBs of the 32-bit value written - in the DQDATA field. The long form will produce a 2-beat message with 25 bits of data - value written to the 1st of the set of IPbus registers - in the DQDATA field, followed by another beat containing the full 32-bit value written to the 2nd of the set of IPbus registers (that is, DQDATA field takes up full 32-bits of the second beat). The DQM message will only be generated upon a 32-bit write to the 2nd of the set of registers, irrespective of any writes to the 1st register in the set (meaning it may remain the same, while the second 32-bit value changes, while also causing the message to be produced).

For more information, refer to the DQM IPbus registers, also known as the DQM mailbox registers.

The format of the data acquisition message is described in Figure 63.

DQDATA - This fixed width (fixed width of 25-bits for single beat, fixed width of 32-bits for second beat) field reflects the data value written to the DQM register(s).

### 10.1.3.8 Data trace with sync message (Read TCODE=14, Write TCODE=13)

VSPA data trace is accomplished by a comparator triggering on an address value match of a VCPU DMEM memory access. When a qualified transaction is recognized the information is sent out in a Data Trace w/Sync message (DTM w/sync).

Transactions are qualified based on configured features, such as the address, the transaction type (read or write), and the source unit performing the transaction. Refer to the debug comparator control and status register (DCnCS) more for details.

The DTM w/sync message is normally two beats with the 2nd beat containing the 16-bit or 32-bit data that was read or written. However, in some cases, discussed below, the specific data element to be returned is nondeterministic, and thus result in setting Attr[1:0] to 00. In this case, the DTM message becomes a single beat message with the F-ADDR field indicating which comparator(s) triggered the DTM message. Refer to Table 117 for additional details.

Separate DTM w/sync message formats exist for read and write.

The VCPU and the IPPU always read the entire line of data from VCPU DMEM, although they may use only a portion of this data line, as indicated in the instruction. Thus, DTM Read (TCODE=14) would generate a DTM message with Attr[1:0]==00 (Indeterminate data), unless the *cond_sel* in the corresponding comparator is configured to *'001 - data_in == data_reg'*. Refer to Table 117 for additional details.

Writes to VCPU DMEM are always the exact size the user requested. DTM Write (TCODE=13) would generate a DTM message with Attr[1:0]==00 (Indeterminate data), unless the *cond_sel* in the corresponding comaprator is configured to *'001 - data_in == data_reg'* or the memory transaction writes 32-bit or less to the VCPU DMEM. Refer to Table 117 for additional details.

The format of the data trace with sync message is described in Figure 63.

Attr - This 4-bit field encodes the transaction attributes: size and the unit accessing the data memory (VCPU DMEM). Refer to Table 117 for more information.

F-ADDR - Typically, this is the address corresponding to the DATA field. However, For DTM messages with Indeterminate data (Attr[1:0]==00), bits [7:0] of the F-ADDR field indicates the comparator(s) which caused the DTM event.

DATA - This is the data read/written in the transaction monitored by a specific appropriately configured comparator. The data may be either 16 or 32 bits. A 16-bit data value may be read/written to either the upper or lower half of a memory element (depending on the source) and is identified in the Attr field strobes (bits 1-0). The DATA fiels is the second beat of the DTM message. The DATA field does not exist for messages with Indeterminate data (Attr[1:0]==00), thus generating a single beat DTM message.

**Table 117. Attr Field Encoding**

| Source Attr[3:2] | Data Strobes Attr[1:0] |
|---|---|
| 00: VCPU<br>01: IPPU<br>10: DMA<br>11: Reserved | 00: Indeterminate data. The F-ADDR[7:0] field indicates the comparator(s) which caused the DTM event. A '1' in bit location F-ADDR[*'n'*] indicates comparator *'n'* caused the DTM event. In this case (Attr[1:0]==00) the EXTN field is set to 0, indicating a single beat message, as no deterministic data value can be retuned. If multiple comparators caused the DTM event, then Attr[1:0] is set to to 00 and multiple bits in F-ADDR are set. The user can use the comparator indiator to find the comparator setting to identify the address range that caused the comparator match event.<br>01: Lower 16-bit access to 32-bit element.<br>10: Upper 16-bit access to 32-bit element.<br>11: Full 32-bit element access. |

**NOTE**

Some configurations may cause the data in the DTM message to be nondeterministic, which will set Attr[1:0] to 00. The user can use the comparator setting to get the address range that caused the comparator match event. These configurations include:

- Comparator configured for address range (window) whose event is configured to generate a DTM. If the transaction to DMEM is a full-line access, the DEBU cannot determine which 32-bit data element to output in the DATA field of the DTM.
- Multiple armed and enabled comparators whose events are configured to generate a DTM are triggered in the same cycle.

**NOTE**

VSPA data trace is accomplished by a comparator triggering on an address value match (only) on the VCPU DMEM memory. The comparator used to generate the DTM messages must be configured with the i_sel field in the corresponding Debug Comparator Control and Status register set to '001 - VSP DMEM'.

#### 10.1.3.9 Timestamp message (TCODE=2)

Timestamp message (TSM) is used to indicate the time, according to the timestamp counter register, when specific events occur. See Timestamps for more information about timestamps and the events which can trigger a TSM.

The format of the timestamp message is described in Figure 63.

TIMESTAMP - This 24-bit (short TSM) or 48-bit (long TSM) field conveys the value of the timer register at the time the TSM was created.

TTYPE - This 3-bit field encodes the cause for which the TSM message was sent. Note that a DTM message, WPT message, DQM message or SYNC message will not be indicated in the TTYPE encoding. Rather, the TS bit in the corresponding message will be set.

**Table 118. TTYPE Encoding**

| TTYPE Value | TTYPE Condition | Description |
|---|---|---|
| 000 | Trace Message | TSM was caused by a DTM, a WPT, a DQM or a SYNC message. The TS bit in the following corresponding message will be set. |
| 100 | LOOP START | TSM was caused by the VCPU starting a loop |
| 101 | LOOP END | TSM was caused by the VCPU ending a loop |
| 110 | JSR | TSM was caused by the VCPU executing a JSR instruction |
| 111 | RTS | TSM was caused by the VCPU executing a RTS instruction |
| 001-011 | Reserved | - |

ETTYPE - This 8-bit field encodes the cause for the long TSM message. Note that more than one bit may be set, which indicates that multiple events occurred simultaneously. Also, if a DTM, a WPT, a DQM and/or a SYNC message is sent simultaneously with the TSM message, each of those messages will follow the TSM message with its TS bit set.

**Table 119. ETTYPE Encoding**

| ETTYPE Bit | ETTYPE Condition | Description |
|---|---|---|
| 0 | JSR | TSM was caused by the VCPU executing a JSR instruction |
| 1 | RTS | TSM was caused by the VCPU executing a RTS instruction |
| 2 | LOOP START | TSM was caused by the VCPU starting a loop |
| 3 | LOOP END | TSM was caused by the VCPU ending a loop |
| 4 | EXT_SYNC_REQ | TSM was caused by assertion of input signal dbg_sync_req |

*Table continues on the next page...*

**Table 119. ETTYPE Encoding (continued)**

| ETTYPE Bit | ETTYPE Condition | Description |
|---|---|---|
| 5 | TSM_ROLLOVER | TSM was caused by Timestamp Counter bits [23:0] changing from 0xff_ffff to 0x00_0000 (Rollover) |
| 6-7 | Reserved | |

### 10.1.4 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
# VSPA IP Registers

## 13.1 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.

## 13.2 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 120. 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).<br>**NOTE**<br>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 120. Register Conventions (continued)**

| Convention | Description |
|---|---|
| * | See footnote for description. |
| Memory areas not defined should be considered reserved. | |

### 13.2.1 VSPA_CCSR memory map

VSPA0.VSPA0 base address: F900_0000h

VSPA1.VSPA1 base address: F900_4000h

VSPA2.VSPA2 base address: F900_8000h

VSPA3.VSPA3 base address: F900_C000h

VSPA4.VSPA4 base address: F901_0000h

VSPA5.VSPA5 base address: F901_4000h

VSPA6.VSPA6 base address: F901_8000h

VSPA7.VSPA7 base address: F901_C000h

#### 13.2.1.1 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 | WORZ | 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 |
| E0h | AUX DMA status register (AD_STATUS) | 32 | W1C | 0000_0000h |
| 188h | AUX DMA program memory address register (AD_PRAM_ADDR) | 32 | RW | 0000_0000h |

*Table continues on the next page...*

*Table continued from the previous page...*

| Offset | Register | Width (In bits) | Access | Reset value |
|---|---|---|---|---|
| 18Ch | AUX DMA AXI address register (AD_AXI_ADDR) | 32 | RW | 0000_0000h |
| 190h | AUX DMA byte count register (AD_BYTE_COUNT) | 32 | RW | 0000_0000h |
| 194h | AUX DMA control register (AD_CTRL) | 32 | RW | 0000_0000h |

#### 13.2.1.2 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 |
| 600h | VCPU to DQM Trace Small Outbox register (DQM_SMALL) | 32 | WO | See description |
| 604h | VCPU to DQM Trace Large MSB Outbox register (DQM_LARGE_MSB) | 32 | WO | See description |
| 608h | VCPU to DQM Trace Large LSB Outbox register (DQM_LARGE_LSB) | 32 | WO | 0000_0000h |
| 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 |

### 13.2.1.3 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 |
| 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 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Offset | Register | Width (In bits) | Access | Reset value |
|---|---|---|---|---|
| 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 | 0010_0000h |
| 40Ch | VCPU CREG 1 (VCPU_CREG1) | 32 | RO | 0000_0000h |
| 410h | Store Unalign Vector Length (ST_UL_VEC_LEN) | 32 | RO | C000_0000h |
| 414h | Vector Predicate Control (VP_CONTROL) | 32 | RO | 0000_0000h |
| 418h | Extended Condition Control (XC_CONTROL) | 32 | RO | 0000_0000h |
| 41Ch | VCPU CREG 2 (VCPU_CREG2) | 32 | RO | 0000_0000h |
| 480h | RAG set rS0 & rS1 registers (RAG_SET_S0S1_0) | 32 | RO | 0000_0000h |
| 484h | RAG set rS2, rV & rSt registers (RAG_SET_S2VSt_0) | 32 | RO | 0000_0000h |
| 488h | RAG set rS0 & rS1 registers (RAG_SET_S0S1_1) | 32 | RO | 0000_0000h |
| 48Ch | RAG set rS2, rV & rSt registers (RAG_SET_S2VSt_1) | 32 | RO | 0000_0000h |
| 490h | RAG set rS0 & rS1 registers (RAG_SET_S0S1_2) | 32 | RO | 0000_0000h |
| 494h | RAG set rS2, rV & rSt registers (RAG_SET_S2VSt_2) | 32 | RO | 0000_0000h |
| 498h | RAG set rS0 & rS1 registers (RAG_SET_S0S1_3) | 32 | RO | 0000_0000h |
| 49Ch | RAG set rS2, rV & rSt registers (RAG_SET_S2VSt_3) | 32 | RO | 0000_0000h |
| 4A0h | RAG set rS0 & rS1 registers (RAG_SET_S0S1_4) | 32 | RO | 0000_0000h |
| 4A4h | RAG set rS2, rV & rSt registers (RAG_SET_S2VSt_4) | 32 | RO | 0000_0000h |
| 4A8h | RAG set rS0 & rS1 registers (RAG_SET_S0S1_5) | 32 | RO | 0000_0000h |
| 4ACh | RAG set rS2, rV & rSt registers (RAG_SET_S2VSt_5) | 32 | RO | 0000_0000h |
| 4B0h | RAG set rS0 & rS1 registers (RAG_SET_S0S1_6) | 32 | RO | 0000_0000h |
| 4B4h | RAG set rS2, rV & rSt registers (RAG_SET_S2VSt_6) | 32 | RO | 0000_0000h |
| 4B8h | RAG set rS0 & rS1 registers (RAG_SET_S0S1_7) | 32 | RO | 0000_0000h |
| 4BCh | RAG set rS2, rV & rSt registers (RAG_SET_S2VSt_7) | 32 | RO | 0000_0000h |

#### 13.2.1.4 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 |

#### 13.2.1.5 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 - 558h | General Purpose Input registers [23 registers] (GP_IN0 - GP_IN22) | 32 | RO | See description |
| 580h - 5D0h | General Purpose Output registers [21 registers] (GP_OUT0 - GP_OUT20) | 32 | RW | 0000_0000h |

#### 13.2.1.6 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 | 0000_0000h |
| 2Ch | External Go Status (EXT_GO_STAT) | 32 | W1C | 0000_0000h |
| 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 |

### 13.2.1.7 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 |
| 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 |

### 13.2.1.8 Version and Configuration Registers memory map

| Offset | Register | Width (In bits) | Access | Reset value |
|---|---|---|---|---|
| 0h | VSPA Hardware Version (HWVERSION) | 32 | RO | 0301_1900h |
| 4h | VCPU Software Version (SWVERSION) | 32 | RW | 0000_0000h |

*Table continues on the next page...*

*Table continued from the previous page...*

| Offset | Register | Width (In bits) | Access | Reset value |
|---|---|---|---|---|
| 40h | VSPA Parameters 0 (PARAM0) | 32 | RO | See description |
| 44h | VSPA Parameters 1 (PARAM1) | 32 | RO | 4520_1517h |
| 48h | VSPA Parameters 2 (PARAM2) | 32 | RO | 8843_4110h |
| 4Ch | VCPU DMEM Size (DMEM0_SIZE) | 32 | RO | 0002_0000h |

### 13.2.2 VSPA Hardware Version (HWVERSION)

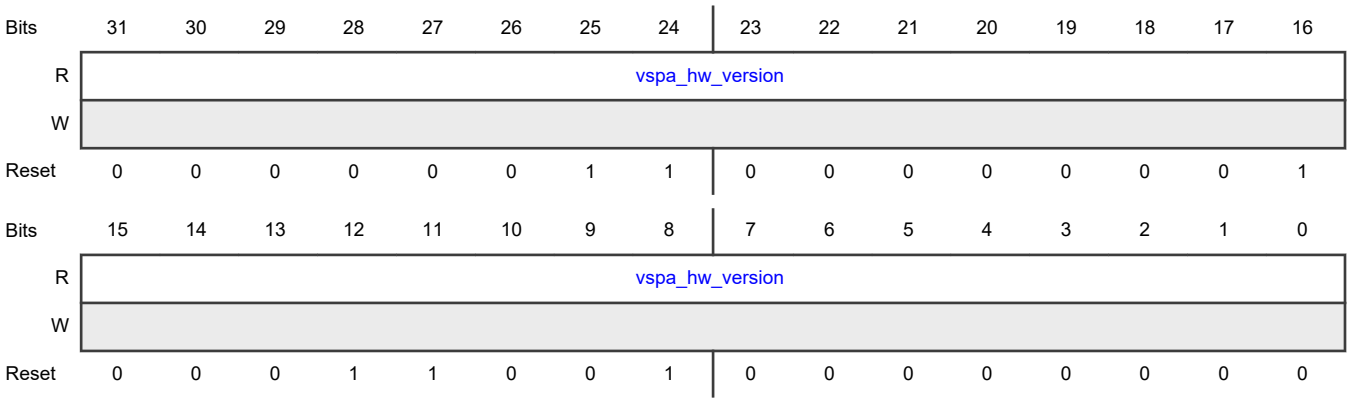
#### Offset

| Register | Offset |
|---|---|
| HWVERSION | 0h |

#### Function

VSPA hardware version

#### Diagram




#### Fields

| Field | Function |
|---|---|
| 31-0<br>vspa_hw_version | vspa_hw_version<br>VSPA hardware version<br>Indicates the version of the VSPA module's hardware. The values in this register are valid immediately after reset. |

### 13.2.3 VCPU Software Version (SWVERSION)

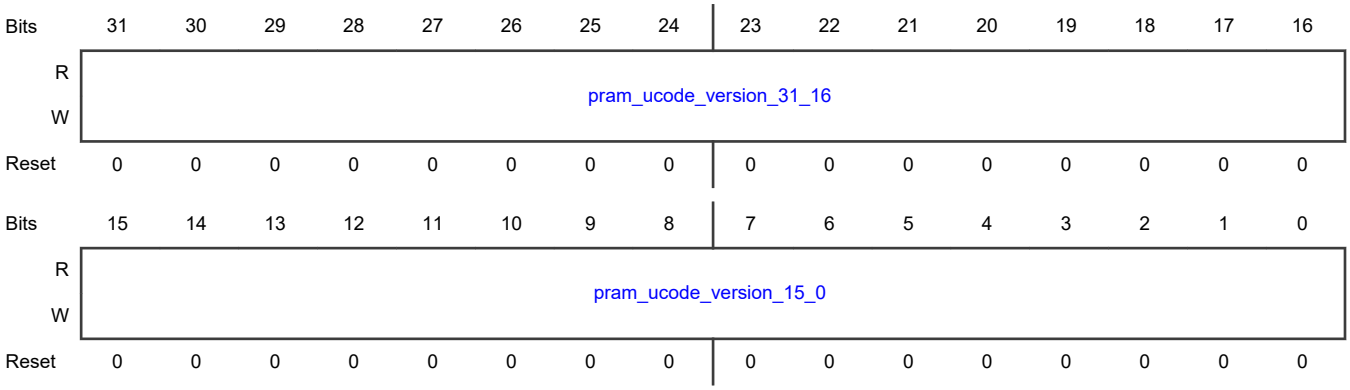
#### Offset

| Register | Offset |
|---|---|
| SWVERSION | 4h |

#### Function

VCPU Software Version

#### Diagram




#### Fields

| Field | Function |
|---|---|
| 31-16<br>pram_ucode_version_31_16 | pram_ucode_version<br>PRAM Software Version<br>This field can be used to indicate the version of the PRAM assembly code.<br>VCPU software configures this register during global initialization after download of a new PRAM image.<br>PRAM version numbers are non-zero. |
| 15-0<br>pram_ucode_version_15_0 | pram_ucode_version<br>PRAM Software Version<br>This field can be used to indicate the version of the PRAM assembly code.<br>VCPU software configures this register during global initialization after download of a new PRAM image.<br>PRAM version numbers are non-zero. |

### 13.2.4 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 | vspa_so... | dma_hal... | Reserved |
| W | | | | | | | | | W1C | W1C | W1C | W1C | | | | |
| Reset | u | u | u | u | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | u | 0 | 0 | u |

| 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_... | Reserved | host_vs... | debug_m... | vcpu_go | ext_go | dma_go | ippu_go | host_go |
| W | | | | | | | | | | | W1C | | | | W1C | |
| Reset | u | u | u | 0 | 0 | u | 0 | 0 | u | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

#### Fields

| Field | Function |
|---|---|
| 31-28<br>— | -<br>Reserved |
| 27<br>host_read_msg1_go_en | host_read_msg1_go_en<br>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.<br>0b - No Go request is generated when the host_read_msg1_go is set.<br>1b - A Go request is generated when the host_read_msg1_go is set. |
| 26<br>host_read_msg0_go_en | host_read_msg0_go_en<br>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. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 0b - No Go request is generated when the host_read_msg0_go is set.<br>1b - A Go request is generated when the host_read_msg0_go is set. |
| 25<br>host_msg1_go_en | host_msg1_go_en<br>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.<br>0b - No Go request is generated when the host_sent_msg1_go is set.<br>1b - A Go request is generated when the host_sent_msg1_go is set. |
| 24<br>host_msg0_go_en | host_msg0_go_en<br>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.<br>0b - No Go request is generated when the host_sent_msg0_go is set.<br>1b - A Go request is generated when the host_sent_msg0_go is set. |
| 23<br>host_read_msg1_go | host_read_msg1_go<br>**NOTE**<br>Access to this field is non-standard and is described in detail below.<br>Host Read Message 1 from VCPU - Go flag.<br>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.<br>0b - VCPU/Host read - No go request is pending from the host mailbox<br>1b - VCPU/Host read - Go request was generated by the host reading a message from the mailbox. |
| 22<br>host_read_msg0_go | host_read_msg0_go<br>**NOTE**<br>Access to this field is non-standard and is described in detail below.<br>Host Read Message 0 from VCPU - Go flag.<br>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.<br>0b - VCPU/Host read - No go request is pending from the host mailbox<br>1b - VCPU/Host read - Go request was generated by the host reading a message from the mailbox |
| 21 | host_sent_msg1_go |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| host_sent_msg1_go | **NOTE**<br>Access to this field is non-standard and is described in detail below.<br>Host Sent Message 1 to VCPU - Go flag.<br>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.<br>**NOTE**<br>VCPU must read the message before clearing this bit. Failing to follow this protocol may result in spurious VCPU_Go events.<br>0b - VCPU/Host read - No go request is pending from the host mailbox<br>1b - VCPU/Host read - Go request was generated by the host writing a message to the mailbox |
| 20<br>host_sent_msg0_go | host_sent_msg0_go<br>**NOTE**<br>Access to this field is non-standard and is described in detail below.<br>Host Sent Message 0 to VCPU - Go flag.<br>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.<br>**NOTE**<br>VCPU must read the message before clearing this bit. Failing to follow this protocol may result in spurious VCPU_Go events.<br>0b - VCPU/Host read - No go request is pending from the host mailbox<br>1b - VCPU/Host read - Go request was generated by the host writing a message to the mailbox |
| 19<br>— | -<br>Reserved |
| 18<br>vspa_soft_rst_req | vspa_soft_rst_req<br>VSPA Soft Reset<br>**NOTE**<br>This VSPA Soft Reset should only be used during de-bugging.<br>Write to 1 causes soft reset. It can be written by the host only - VCPU writes have no effect.. Reads will return status 1 while soft reset is in progress, 0 otherwise.<br>The following registers and bits are cleared by soft reset: |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | • CONTROL: host_go (bit 0), ippu_go (bit 1), vcpu_go (bit 4) debug_msg_go (bit 5), host_vsp_flags_go (bit 6), debug_msg_go_en (bit 9), host_vsp_flags0_go_en (bit 11), host_vsp_flags1_go_en (bit 12), host_sent_msg0_go (bit 20), host_sent_msg1_go (bit 21), host_read_msg0_go (bit 22), host_read_msg1_go (bit 23), host_msg0_go_en (bit 24), host_msg1_go_en (bit 25), host_read_msg0_go_en (bit 26), host_read_msg1_go_en (bit 27)<br>• AXISLV_GOEN0: all bits<br>• AXISLV_GOEN1: all bits<br>• EXT_GO_STAT: all bits<br>• EXT_GO_ENA: all bits<br>• THREAD_CTRL_STAT: thread_change_flag (bit 15)<br>• EXCEPTION_CTRL: vcpu_irq_en (bit 29)<br>• EXCEPTION_STAT: vcpu_irq_go (bit 30) |
| 17<br>dma_halt_req | dma_halt_req<br>DMA Halt Request<br>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.<br>0b - Not requesting the DMA to halt<br>1b - Request for the DMA to halt operation at a clean AXI boundary |
| 16-13<br>— | -<br>Reserved |
| 12<br>host_vsp_flags1_go_en | host_vsp_flags1_go_en<br>Host to VSPA flags 1 GO enable<br>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.<br>0b - Host to VSPA flags 1 GO is disabled<br>1b - Host to VSPA flags 1 GO is enabled |
| 11<br>host_vsp_flags0_go_en | host_vsp_flags0_go_en<br>Host to VSPA flags 0 GO enable<br>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.<br>0b - Host to VSPA flags 0 GO is disabled |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 1b - Host to VSPA flags 0 GO is enabled |
| 10<br>—— | -<br>Reserved |
| 9<br>debug_msg_go_en | debug_msg_go_en<br>Debug message go enable<br>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.<br>It is a read/write register bit. It can be written and read at any time.<br>0b - No Go request is generated when the debug module writes a message to the debug mailbox<br>1b - Go request is generated when the debug module writes a message to the debug mailbox |
| 8<br>axislv_go | axislv_go<br>Read only status bit.<br>0b - Indicates no VCPU go is pending from the AXI slave subsystem.<br>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<br>—— | -<br>Reserved |
| 6<br>host_vsp_flags_go | host_vsp_flags_go<br>**NOTE**<br>Access to this field is non-standard and is described in detail below.<br>This bit is set when either condition occurs:<br>• one or more bits are set in the VSPA_HOST_VCPU_FLAGS0 register and host_vsp_flags0_go_en is set to one, or<br>• one or more bits are set in the VSPA_HOST_VCPU_FLAGS1 register and host_vsp_flags1_go_en is set to one<br>This bit is cleared when both conditions occur:<br>• all bits in the VSPA_HOST_VCPU_FLAGS0 register are zero or the host_vsp_flags0_go_en is cleared, and<br>• all bits in the VSPA_HOST_VCPU_FLAGS1 register are zero or the host_vsp_flags1_go_en is cleared<br>0b - VCPU/Host read - No go request is pending from the host flags registers<br>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 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| debug_msg_go | **NOTE**<br>Access to this field is non-standard and is described in detail below.<br><br>Debug message go<br><br>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.<br><br>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.<br><br>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.<br><br>0 VCPU/Host read - No go request is pending from the debug host mailbox<br><br>1 VCPU/Host read - Go request was generated by the debug host writing a message to the mailbox<br><br>0 Host write - Ignored<br><br>1 Host write - Ignored<br><br>0 VCPU write - Ignored<br><br>1 VCPU write - Clear the status bit (if set) |
| 4<br><br>vcpu_go | vcpu_go<br><br>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.<br><br>The VCPU SUPV or USER can write to 1, only SUPV can write to 0. When 1, a GO event will be pending.<br><br>Host reads and writes have no effect.<br><br>**NOTE**<br>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<br><br>ext_go | ext_go<br><br>External go<br><br>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.<br><br>0b - No go request is pending from an external event<br><br>1b - Go request was generated by an external event |
| 2<br><br>dma_go | dma_go<br><br>DMA unit go |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 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.<br>0b - No go request is pending from the DMA unit<br>1b - Go request was generated by the DMA unit |
| 1<br>ippu_go | ippu_go<br>**NOTE**<br>Access to this field is non-standard and is described in detail below.<br>IPPU go<br>This is a VCPU write-1-to-clear status bit. When set, it reflects that the IPPU issued a VCPU GO command.<br>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.<br>0 VCPU/Host read - No go request is pending from the IPPU<br>1 VCPU/Host read - Go request was generated by the IPPU<br>0 Host write - Ignored<br>1 Host write - Ignored<br>0 VCPU write - Ignored<br>1 VCPU write - Clear the status bit (if set) |
| 0<br>host_go | host_go<br>**NOTE**<br>Access to this field is non-standard and is described in detail below.<br>Host go<br>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.<br>When set, it requests that VCPU start processing instructions at PMEM address 0 (assuming VCPU is stopped).<br>0 VCPU/Host read - VCPU has not been requested to go by the host since it last cleared this bit<br>1 VCPU/Host read - A VCPU host go was requested by the host and was not yet cleared by the VCPU<br>0 Host write - Ignored<br>1 Host write - Start VCPU core processing at PMEM address 0.<br>0 VCPU write - Ignored<br>1 VCPU write - Clear the status bit (if set) |

### 13.2.5 VSPA Interrupt Enable register (IRQEN)

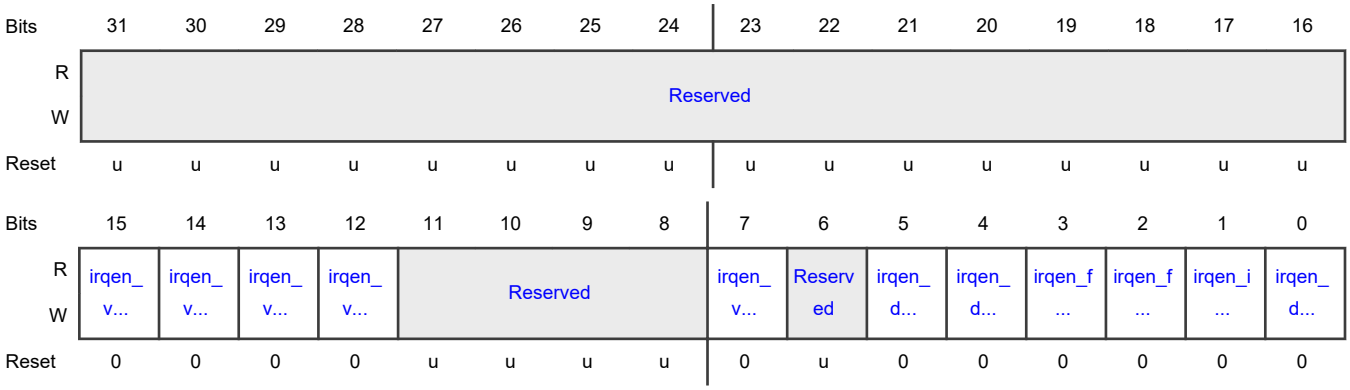
#### Offset

| Register | Offset |
|---|---|
| IRQEN | Ch |

#### Function

VSPA Interrupt Enable register

#### Diagram




#### Fields

| Field | Function |
|---|---|
| 31-16<br>— | -<br>Reserved |
| 15<br>irqen_vcpu_read_msg1 | irqen_vcpu_read_msg1<br>Host to VCPU message 1 read interrupt enable.<br>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.<br>0b - Host to VCPU message 1 read interrupt disabled<br>1b - Host to VCPU message 1 read interrupt enabled |
| 14<br>irqen_vcpu_read_msg0 | irqen_vcpu_read_msg0<br>Host to VCPU message 0 read interrupt enable.<br>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.<br>0b - Host to VCPU message 0 read interrupt disabled<br>1b - Host to VCPU message 0 read interrupt enabled |
| 13 | irqen_vcpu_sent_msg1 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| irqen_vcpu_sent_msg1 | VCPU to Host message 1 sent interrupt enable.<br>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.<br>0b - VCPU to host message 1 sent interrupt disabled<br>1b - VCPU to host message 1 sent interrupt enabled |
| 12<br>irqen_vcpu_sent_msg0 | irqen_vcpu_sent_msg0<br>VCPU to Host message 0 sent interrupt enable.<br>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.<br>0b - VCPU to host message 0 sent interrupt disabled<br>1b - VCPU to host message 0 sent interrupt enabled |
| 11-8<br>— | -<br>Reserved. |
| 7<br>irqen_vcpu_iit | irqen_vcpu_iit<br>VCPU illegal instruction trap interrupt enable.<br>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.<br>0b - VCPU Illegal instruction trap interrupt disabled<br>1b - VCPU Illegal instruction trap interrupt enabled |
| 6<br>— | -<br>Reserved |
| 5<br>irqen_dma_error | irqen_dma_error<br>VSPA DMA error interrupt enable<br>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.<br>0b - DMA error interrupt disabled<br>1b - DMA error interrupt enabled |
| 4<br>irqen_dma_cmp | irqen_dma_cmp<br>VSPA DMA complete interrupt enable<br>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.<br>0b - DMA transfer complete interrupt disabled<br>1b - DMA transfer complete interrupt enabled |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 3<br>irqen_flags1 | irqen_flags1<br>Flags 1 Interrupts Enable<br>Enables the vcpu_flags1_irq_b interrupt. This interrupt is asserted whenever any of the bits in the VCPU_HOST_FLAGS1 register are set.<br>0b - VCPU_HOST_FLAGS1 interrupts disabled<br>1b - VCPU_HOST_FLAGS1 interrupts enabled |
| 2<br>irqen_flags0 | irqen_flags0<br>Flags 0 interrupts enable<br>Enables the vcpu_flags0_irq_b interrupt. This interrupt is asserted whenever any of the bits in the VCPU_HOST_FLAGS0 register are set.<br>0b - VCPU_HOST_FLAGS0 interrupts disabled<br>1b - VCPU_HOST_FLAGS0 interrupts enabled |
| 1<br>irqen_ippu_done | irqen_ippu_done<br>IPPU done interrupt enable<br>Enables the IPPU done interrupt. When enabled, when the IPPU done status flag asserts, this interrupt will be asserted.<br>0b - IPPU done interrupt disabled<br>1b - IPPU done interrupt enabled |
| 0<br>irqen_done | irqen_done<br>VCPU Done interrupt enable<br>Enables the vcpu_done_irq_b interrupt. This interrupt is asserted whenever the DONE bit in the STATUS register is set.<br>0b - Done interrupt disabled<br>1b - Done interrupt enabled |

### 13.2.6 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 | Reserved | irq_pen... | irq_pen... | irq_pen... | irq_pen... | irq_pen... | done |
| W | W1C | W1C | W1C | W1C | | | | | W1C | | | | | | W1C | W1C |
| Reset | 0 | 0 | 0 | 0 | u | u | u | 0 | 0 | u | 0 | 0 | 0 | 0 | 0 | 0 |

### Fields

| Field | Function |
|---|---|
| 31-18<br>— | -<br>Reserved |
| 17<br>dma_halt_ack | dma_halt_ack<br>DMA halt acknowledge.<br>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..<br>0b - The DMA has not halted operation in response to a halt request<br>1b - The DMA has halted operation in response to a halt request |
| 16<br>— | -<br>Reserved |
| 15<br>vcpu_read_msg1 | vcpu_read_msg1<br>VCPU read message 1.<br>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. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 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.<br>0b - The status bit was never set following reset, or was cleared by a write 1 to clear (for read)<br>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<br>vcpu_read_msg0 | vcpu_read_msg0<br>VCPU read message 0.<br>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.<br>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.<br>0b - The status bit was never set following reset, or was cleared by a write 1 to clear (for read)<br>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<br>vcpu_sent_msg1 | vcpu_sent_msg1<br>VCPU sent message 1.<br>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.<br>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.<br>**NOTE**<br>The host must read the message before clearing this bit. Failing to follow this protocol may result in spurious interrupts.<br>0b - The status bit was never set following reset, or was cleared by a write 1 to clear (for read)<br>1b - The VCPU wrote the VSPA_VCPU_OUT_1_LSB register (for read) |
| 12<br>vcpu_sent_msg0 | vcpu_sent_msg0<br>VCPU sent message 0.<br>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.<br>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.<br>**NOTE**<br>The host must read the message before clearing this bit. Failing to follow this protocol may result in spurious interrupts. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 0b - The status bit was never set following reset, or was cleared by a write 1 to clear (for read)<br>1b - The VCPU wrote the VSPA_VCPU_OUT_0_LSB register (for read) |
| 11-9<br>— | -<br>Reserved |
| 8<br>busy | busy<br>VCPU busy<br>Indicates whether the VCPU core is idle or busy executing code.<br>0b - Idle<br>1b - Busy |
| 7<br>vcpu_iit | vcpu_iit<br>VCPU illegal instruction trap.<br>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.<br>0b - No illegal instructions were detected by the VCPU<br>1b - One or more illegal instructions were detected by the VCPU |
| 6<br>— | -<br>Reserved |
| 5<br>irq_pend_dma_error | irq_pend_dma_error<br>IRQ pending DMA error<br>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 and DMA_CFGERR_STAT registers. It can only be cleared by clearing those underlying status register bits.<br>0b - No IRQ pending<br>1b - DMA transfer or configuration error occurred |
| 4<br>irq_pend_dma_comp | irq_pend_dma_comp<br>IRQ pending DMA complete<br>Indicates that one or more bits are set in the DMA_IRQ_STAT register.<br>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.<br>0b - No IRQ pending |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 1b - DMA transfer completed for at least one channel which has its associated DMA_IRQ_STAT bit set |
| 3<br>irq_pend_flags1 | irq_pend_flags1<br>IRQ pending VCPU_HOST_FLAGS1<br>Indicates that one or more bits are set in the VCPU_HOST_FLAGS1 register.<br>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.<br>0b - No VCPU_HOST_FLAGS1 bits set<br>1b - At least one bit in the VCPU_HOST_FLAGS1 register is set |
| 2<br>irq_pend_flags0 | irq_pend_flags0<br>IRQ pending VCPU_HOST_FLAGS0<br>Indicates that one or more bits are set in the VCPU_HOST_FLAGS0 register.<br>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.<br>0b - No VCPU_HOST_FLAGS0 bits set<br>1b - At least one bit in the VCPU_HOST_FLAGS0 register is set |
| 1<br>irq_pend_ippu_done | irq_pend_ippu_done<br>IRQ pending IPPU done<br>Indicates that IPPU done status flag was asserted. This bit can only be cleared by writing one to this bit location.<br>0b - No IRQ pending<br>1b - IPPU done interrupt is pending |
| 0<br>done | done<br>**NOTE**<br>Access to this field is non-standard and is described in detail below.<br>Done<br>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.<br>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).<br>0b - VCPU indicates processing not done<br>1b - VCPU indicates processing done |

### 13.2.7 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

| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R | flagn | | | | | | | | | | | | | | | |
| W | W1C | | | | | | | | | | | | | | | |
| 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 | flagn | | | | | | | | | | | | | | | |
| W | W1C | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

#### Fields

| Field | Function |
|---|---|
| 31-0<br>flagn | flagn<br>**NOTE**<br>Access to this field is non-standard and is described in detail below.<br>Indicates that a VCPU group *n* event occurred, the nature of which is definable by VCPU software.<br>0 VCPU/Host read - No VCPU group *n* event occurred<br>1 VCPU/Host read - VCPU group *n* event occurred |

*Table continues on the next page...*

| Field | Function |
|---|---|
| | 0 Host write - Ignored<br>1 Host write - Clear VCPU group *n* event flag<br>0 VCPU write - Ignored<br>1 VCPU write - Set VCPU group *n* event flag |

### 13.2.8 Host to VCPU Flags register a (HOST_VCPU_FLAGS0 - HOST_VCPU_FLAGS1)

#### Offset

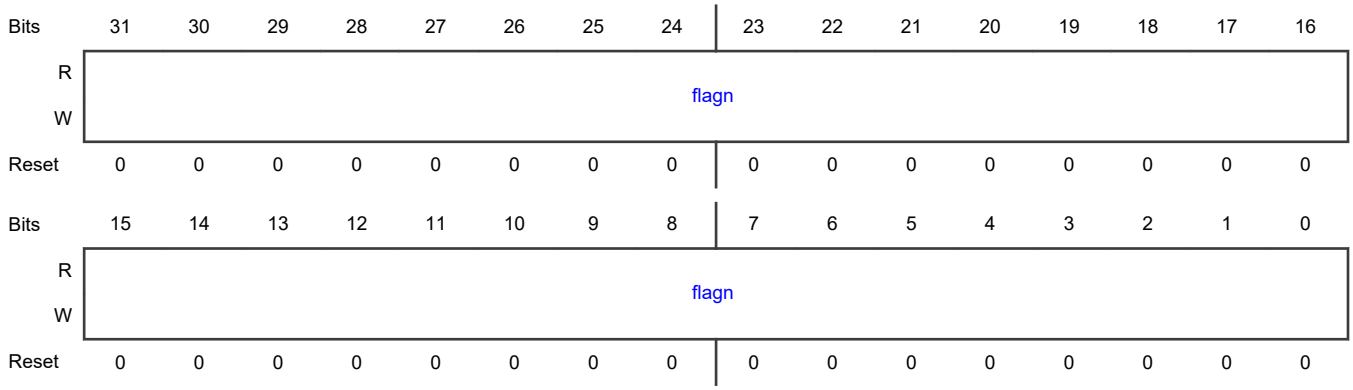
| Register | Offset |
|---|---|
| HOST_VCPU_FLAGS0 | 1Ch |
| HOST_VCPU_FLAGS1 | 20h |

#### 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<br>flagn | flagn<br>Host to VCPU flag.<br>Read behavior is identical for both the host and the VCPU. Write behavior differs between the host and the VCPU. |

*Table continues on the next page...*

| Field | Function |
|---|---|
| | 0 - indicates VCPU wrote the bit to 1, or the bit was never set by the host, since the last reset. (for read behavior)<br>1- indicates the host wrote the bit to 1 (for read behavior)<br>0 - no effect (for host write behavior)<br>1 - sets the bit (for host write behavior)<br>0 - no effect (for VCPU write behavior)<br>1 - clears the bit (for VCPU write behavior) |

### 13.2.9 External Go Enable (EXT_GO_ENA)

#### Offset

| Register | Offset |
|---|---|
| EXT_GO_ENA | 28h |

#### Function

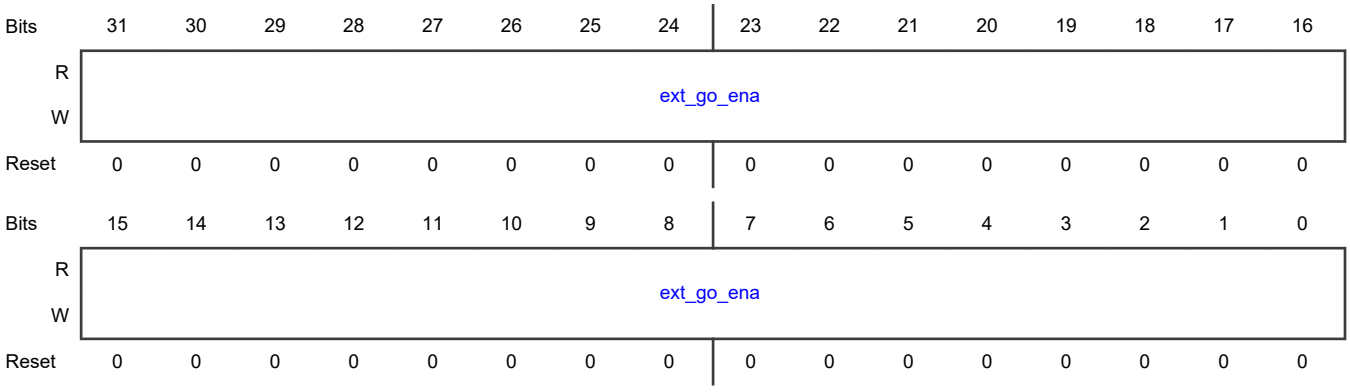
The EXT_GO_ENA register is used to control the generation of VCPU GO events in response to rising edges detected on ext_VCPU_go inputs to the VSPA platform. If a given bit *x* is set in both the EXT_GO_STAT and EXT_GO_ENA registers, then a VCPU GO will be asserted.

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 |
|---|---|
| 31-0<br>ext_go_ena | ext_go_ena<br>External Go Enable<br>Enable/disable a VCPU GO event upon detection of rising edges of the associated ext_VCPU_go inputs of the VSPA platform.<br>00000000000000000000000000000000b - Do not allow generation of a GO event.<br>00000000000000000000000000000001b - Enable generation of a GO event. |

## 13.2.10 External Go Status (EXT_GO_STAT)

**Offset**

| Register | Offset |
|---|---|
| EXT_GO_STAT | 2Ch |

**Function**

**NOTE**

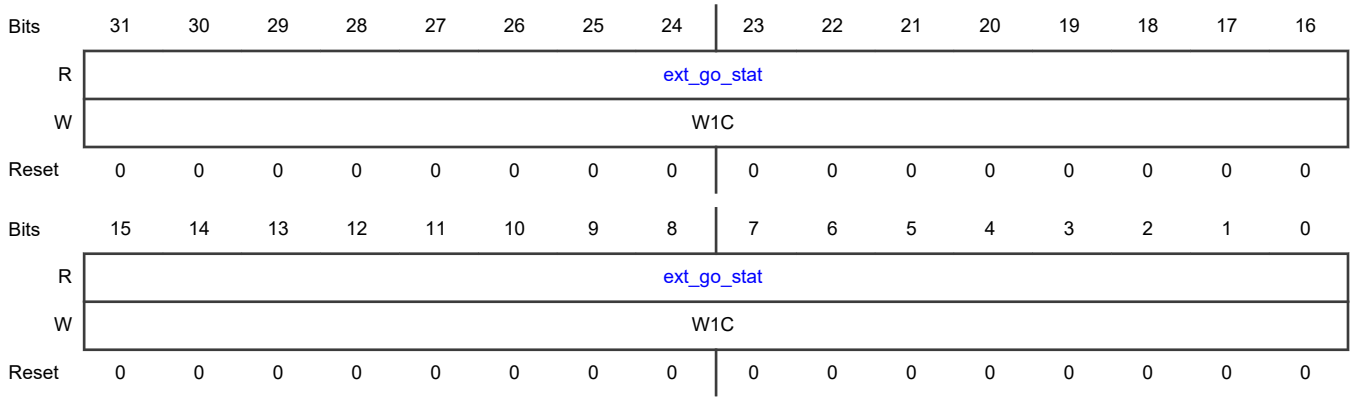
Pay close attention to the register bit field descriptions following the register figure because a field designated with read-write access in the register figure may have additional non-standard behavior described in its corresponding register bit field description.

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 |
|---|---|
| 31-0<br>ext_go_stat | ext_go_stat<br>**NOTE**<br>Access to this field is non-standard and is described in detail below.<br>External Go Stat<br>Rising edges detected on the ext_VCPU_go input.<br>0 VCPU/Host read - No rising edge detected<br>1 VCPU/Host read - At least one rising edge was detected<br>0 VCPU write - Ignored<br>1 VCPU write - Clear status bit (if set)<br>0 Host write - Ignored<br>1 Host write - Ignored |

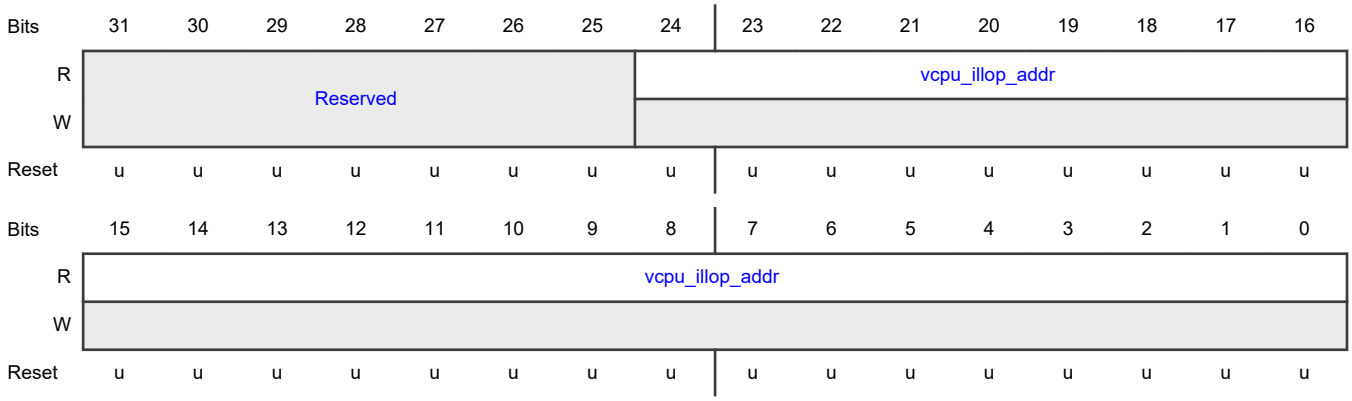
## 13.2.11 VSPA VCPU Illegal Opcode Address (ILLOP_STATUS)

**Offset**

| Register | Offset |
|---|---|
| ILLOP_STATUS | 30h |

**Function**
VSPA VCPU Illegal Opcode Address

**Diagram**



**Fields**

| Field | Function |
|---|---|
| 31-25<br>— | -<br>Reserved |
| 24-0<br>vcpu_illop_addr | vcpu_illop_addr<br>VSPA VCPU Illegal Opcode Address.<br>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. |

## 13.2.12 VSPA Parameters 0 (PARAM0)

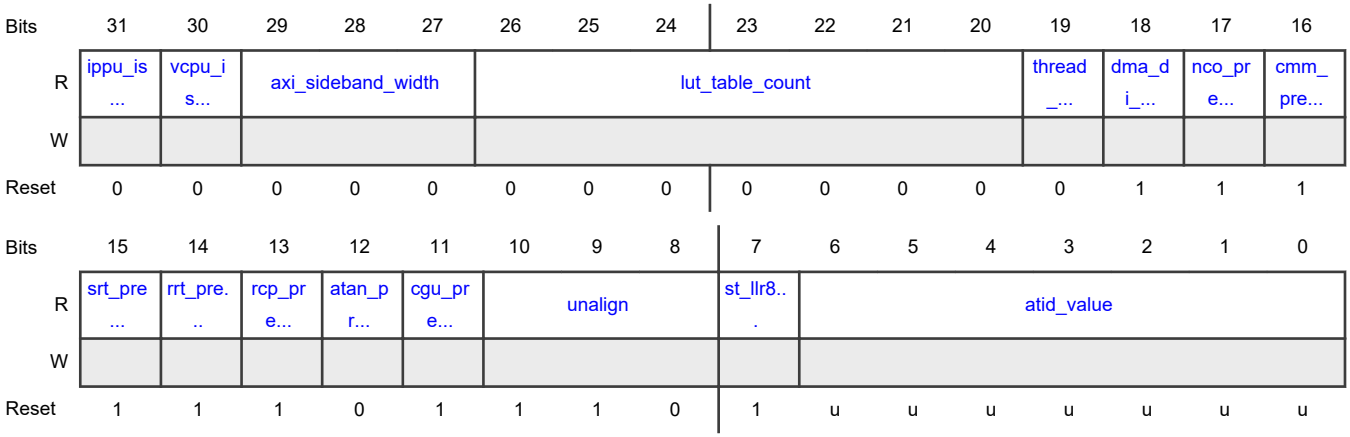
**Offset**

| 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<br>IPPU isolation status |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| ippu_isolate | Indicates whether the IPPU can operate while the VCPU is powered down.<br>The values in this register are valid immediately after reset.<br>0b - The IPPU and VCPU are both powered down together.<br>1b - The IPPU will not be powered down, it can operate when the VCPU is powered down. |
| 30<br>vcpu_isolate | vcpu_isolate<br>VCPU isolate.<br>This status bit indicates that the VCPU can be isolated, and possibly powered down by the system.<br>0b - The VCPU cannot be isolated (and cannot be powered down independently for the VSPA).<br>1b - The VCPU can be isolated (and possibly powered down). |
| 29-27<br>axi_sideband_width | axi_sideband_width<br>These status bits indicate the number of AXI sideband outputs the DMA uses for any given chip. The encodings are as follows:<br>000b - 8 awsideband and 8 arsideband outputs<br>001b - 1 awsideband and 1 arsideband outputs<br>010b - 2 awsideband and 2 arsideband outputs<br>011b - 3 awsideband and 3 arsideband outputs<br>100b - 4 awsideband and 4 arsideband outputs<br>101b - 5 awsideband and 5 arsideband outputs<br>110b - 6 awsideband and 6 arsideband outputs<br>111b - 7 awsideband and 7 arsideband outputs |
| 26-20<br>lut_table_count | lut_table_count<br>Lookup table count<br>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.<br>0000000b - The VCPU has no support for lookup table functionality<br>0111111b - The VCPU has support for 127 full lookup tables (and 254 half tables) |
| 19<br>thread_protection | thread_protection<br>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.<br>0b - thread protection is not supported<br>1b - thread protection is supported |
| 18 | dma_di_eng |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| dma_di_eng | DMA de-interleaving engine.<br>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 the description of the VSPA_DMA_XFR_CTRL register.<br>0b - DMA does not support de-interleaving commands<br>1b - DMA does support de-interleaving commands |
| 17<br>nco_present | nco_present<br>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.<br>0b - VCPU does not support NCO instructions<br>1b - VCPU does support NCO instructions |
| 16<br>cmm_present | cmm_present<br>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.<br>0b - VCPU does not support CMM modes<br>1b - VCPU supports CMM modes |
| 15<br>srt_present | srt_present<br>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.<br>0b - VCPU does not support srt instructions<br>1b - VCPU supports srt instructions |
| 14<br>rrt_present | rrt_present<br>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.<br>0b - VCPU does not support rrt instructions<br>1b - VCPU supports rrt instructions |
| 13<br>rcp_present | rcp_present<br>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.<br>0b - VCPU does not support rcp instructions |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 1b - VCPU supports rcp instructions |
| 12<br>atan_present | atan_present<br>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.<br>0b - VCPU does not support atan instructions<br>1b - VCPU supports atan instructions |
| 11<br>cgu_present | cgu_present<br>This status bit indicates whether the VSPA VCPU supports the CGU instructions. If a CGU instruction or a CGU S1 mode is issued when the VCPU does not support CGU functionality, the instruction will be treated as a NOP, and will not be trapped as an illegal opcode.<br>0b - VCPU does not support CGU instructions<br>1b - VCPU supports CGU instructions |
| 10-8<br>unalign | unalign<br>The unalign bit field is: 6<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>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<br>st_llr8_qam_enable | st_llr8_qam_enable<br>Indicates if the Store QAM llr8 instructions are enabled<br>0b - VCPU does not support st.uline.llr8 instructions<br>1b - VCPU supports st.uline.llr8 instructions |
| 6-0 | atid_value |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| atid_value | ATID value<br>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. |

## 13.2.13 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 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 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 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

### Fields

| Field | Function |
|---|---|
| 31<br>rsse | rsse<br>RSSE module present<br>Indicates the presence of the RSSE sub-module selected by the VSPA block configuration. The values in this register are valid immediately after reset.<br>0b - RSSE module does not exist in this configuration<br>1b - RSSE module exists in this configuration |
| 30-28<br>axi_data_width | axi_data_width<br>AXI data width |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 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<br>000b - 32-bit AXI data width<br>001b - 64-bit AXI data width<br>010b - 128-bit AXI data width<br>011b - 256-bit AXI data width<br>100b - 512-bit AXI data width<br>101b - 1024-bit AXI data width |
| 27-24<br>axi_id_width | axi_id_width<br>AXI ID width<br>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<br>dma_cnt | dma_cnt<br>DMA channel count<br>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<br>gp_out | gp_out<br>GP_OUT Register Count<br>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<br>gp_in | gp_in<br>GP_IN Register Count<br>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 |

### 13.2.14 VSPA Parameters 2 (PARAM2)

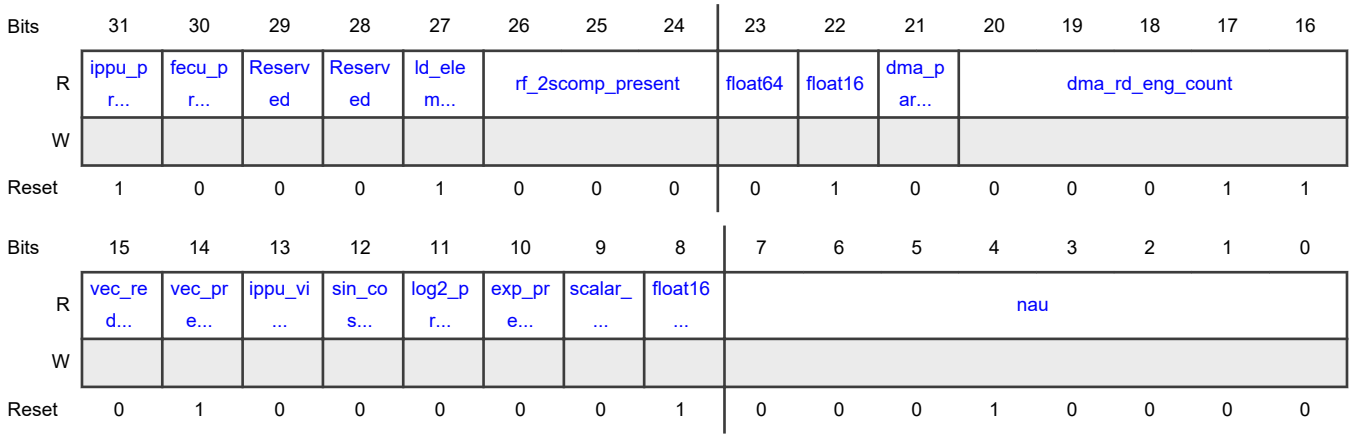
**Offset**

| 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<br>ippu_present | ippu_present<br>IPPU present<br>This bit indicates whether an IPPU exists in this instance of VSPA<br>0b - IPPU does not exist in this instance of VSPA<br>1b - IPPU exists in this instance of VSPA |
| 30<br>fecu_present | fecu_present<br>FECU present<br>This bit indicates whether a FECU exists in this instance of VSPA.<br>0b - FECU does not exist in this instance of VSPA<br>1b - FECU exists in this instance of VSPA |
| 29<br>— | -<br>Reserved |
| 28<br>— | -<br>Reserved |
| 27<br>ld_elem_reorder | ld_elem_reorder<br>If this parameter bit is set, the element reverse instructions are available, otherwise, they are not.<br>The associated VCPU instruction is ld.el_rev. In the IPPU, the affected instruction is ld. The two ld modes er(Element-reversal) and brmr(BRMR - Bit-reversal-multiple-registers) are available only when this parameter is set. |
| 26-24<br>rf_2scomp_present | rf_2scomp_present<br>000b - ld.2scomp instruction is not supported. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 001b - ld.2scomp instruction supports conversion of 16-bit 2's complement values only.<br>010b - ld.2scomp instruction supports conversion of 8, 10, 12 and 16-bit 2's complement values. |
| 23<br>float64 | float64 enable<br>Float 64 support<br>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.<br>0b - 64-bit floating point AU operations are not supported<br>1b - 64-bit floating point AU operations are supported |
| 22<br>float16 | float16 enable<br>The value of this bit indicates whether this instance of VCPU supports 16-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.<br>0b - 16-bit floating point AU operations are not supported<br>1b - 16-floating point AU operations are supported |
| 21<br>dma_parity_present | DMA parity present<br>This bit indicates whether the DMA on this VSPA supports parity.<br>0b - DMA parity logic is not supported<br>1b - DMA parity logic is supported |
| 20-16<br>dma_rd_eng_count | DMA read engine count<br>Indicates the number of DMA read engines selected by the VSPA configuration parameter DMA_RD_ENG_COUNT. The values in this register are valid immediately after reset. |
| 15<br>vec_red_present | Vector reduction present<br>This bit indicates whether the vector reduction feature is supported in this instance of VSPA<br>0b - Vector reduction feature is not supported<br>1b - Vector reduction feature is supported |
| 14<br>vec_pred_present | Vector predicate present<br>This bit indicates whether the vector predicate logic exists in this instance of VSPA.<br>0b - Vector predicate logic is not supported<br>1b - Vector predicate logic is supported |
| 13<br>ippu_vinx_present | ippu_vinx_present<br>The value of this bit indicates whether this instance of VSPA includes IPPU enhanced vector indexing logic and instructions (ld.vinx.mem_type insruction). |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 0b - Enhanced vector indexing (ld.vinx.mem_type insruction) is not supported<br>1b - Enhanced vector indexing (ld.vinx.mem_type insruction) is supported |
| 12<br>sin_cos_present | sin_cos_present<br>This status bit indicates whether the VSPA VCPU supports the sin, cos, asin, acos and expj instructions. If a VCPU sin/cos/asin/acos/expj instruction is issued when the VCPU does not support sin/cos/asin/acos/expj functionality, the instruction will be treated as a NOP, and will not be trapped as an illegal opcode.<br>0b - VCPU does not support sin/cos/asin/acos/expj instructions<br>1b - VCPU supports sin/cos/asin/acos/expj instructions |
| 11<br>log2_present | log2_present<br>This status bit indicates whether the VSPA VCPU supports the log2 instruction. If a VCPU log2 instruction is issued when the VCPU does not support log2 functionality, the instruction will be treated as a NOP, and will not be trapped as an illegal opcode.<br>0b - VCPU does not support the log2 instruction<br>1b - VCPU supports the log2 instruction |
| 10<br>exp_present | exp_present<br>This status bit indicates whether the VSPA VCPU supports the exp and expj instructions. If a VCPU exp/expj instruction is issued when the VCPU does not support exp/expj functionality, the instruction will be treated as a NOP, and will not be trapped as an illegal opcode.<br>0b - VCPU does not support exp/expj instructions<br>1b - VCPU supports exp/expj instructions |
| 9<br>scalar_fp_present | scalar_fp_present<br>This status bit indicates whether the VSPA VCPU supports the scalar Floating Point instructions. If a VCPU scalar FP instruction is issued when the VCPU does not support scalar FP functionality, the instruction will be treated as a NOP, and will not be trapped as an illegal opcode.<br>0b - VCPU does not support scalar floating point instructions<br>1b - VCPU supports scalar floating point instructions |
| 8<br>float16_nco_padd | float16_nco_padd<br>This status bit indicates whether the VSPA VCPU supports 16-bit floating point nco and padd operations.<br>0b - VCPU does not support 16-bit floating point nco and padd operations<br>1b - VCPU supports 16-bit floating point nco and padd operations |
| 7-0<br>nau | nau<br>Number of arithmetic units (AU)<br>Indicates the number of arithmetic units implemented in this VCPU configuration. The values in this register are valid immediately after reset. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 00010000b - 16AU |

## 13.2.15 VCPU DMEM Size (DMEM0_SIZE)

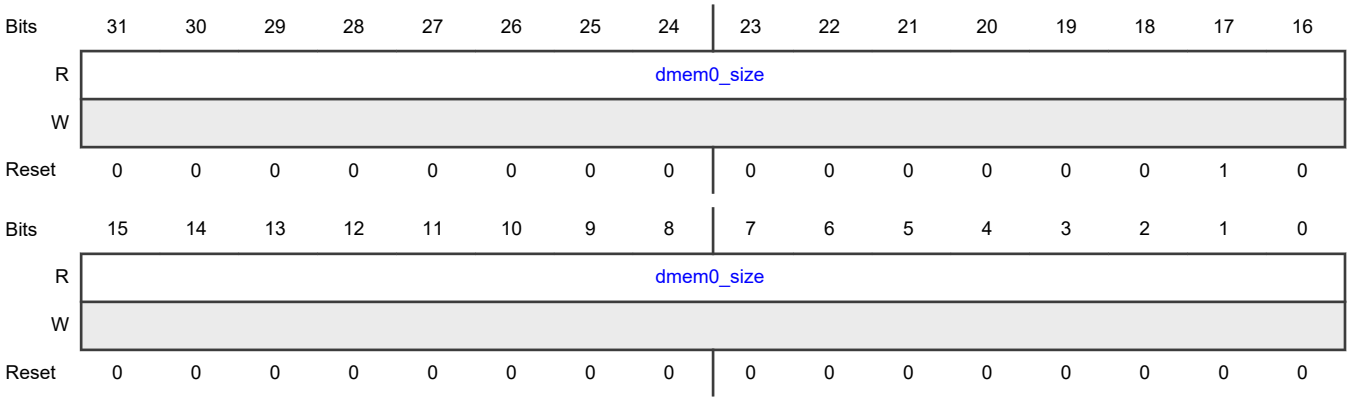
### Offset

| Register | Offset |
|---|---|
| DMEM0_SIZE | 4Ch |

### Function

VCPU DMEM Size

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-0<br>dmem0_size | VSPA VCPU DMEM size in bytes.<br>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. |

## 13.2.16 AXI Slave flags register a (AXISLV_FLAGS0 - AXISLV_FLAGS1)

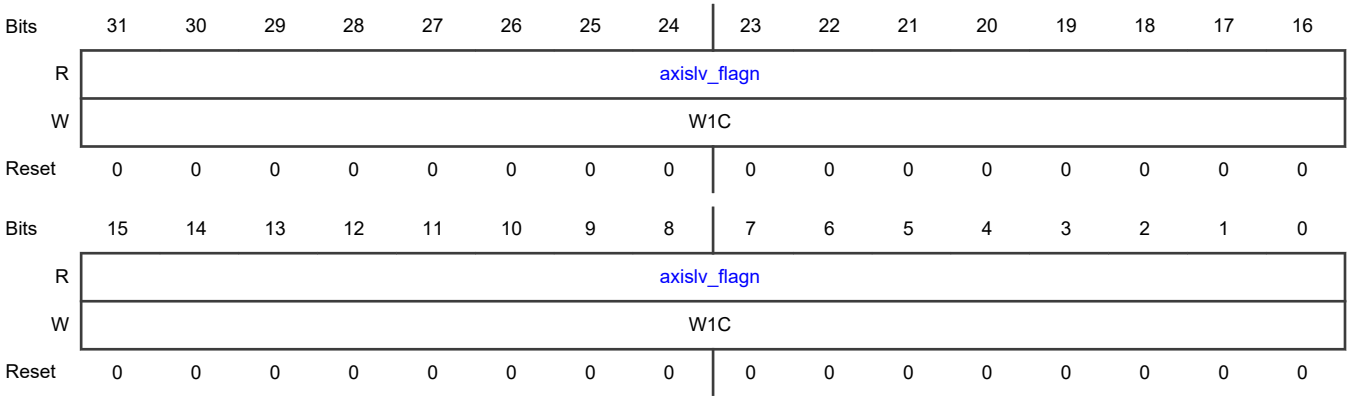
### Offset

| 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<br>axislv_flagn | axislv_flagn (n is 31-0 for flags0, and 63-32 for flags1)<br>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. |

## 13.2.17 AXI Slave Go Enable register a (AXISLV_GOEN0 - AXISLV_GOEN1)

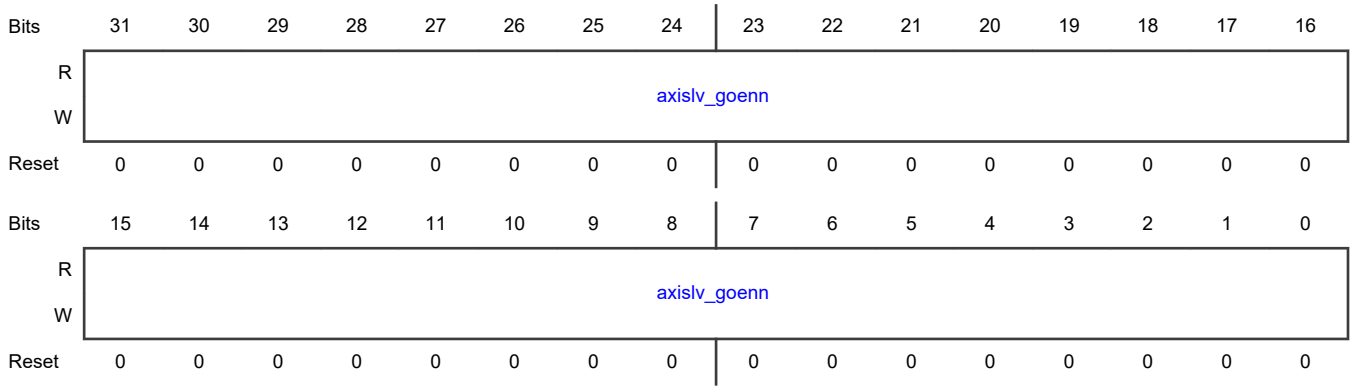
### Offset

| 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<br>axislv_goenn | axislv_goenn (n is 31-0 for goen0, and 63-32 for goen1)<br>Enable/disable a VCPU GO event upon detection of rising edges of the associated ext_VCPU_go inputs of the VSPA platform.<br>00000000000000000000000000000000b - Do not allow generation of a GO event.<br>00000000000000000000000000000001b - Enable generation of a GO event. |

## 13.2.18 Platform Input (PLAT_IN_0)

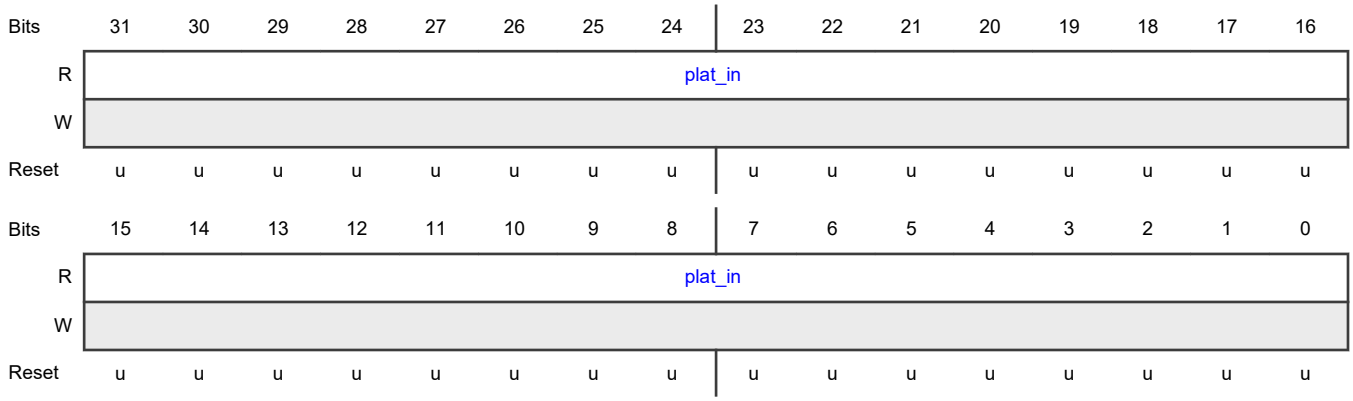
### Offset

| 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<br>plat_in | plat_in<br>Platform integration inputs.<br>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. |

## 13.2.19 Platform Output (PLAT_OUT_0)

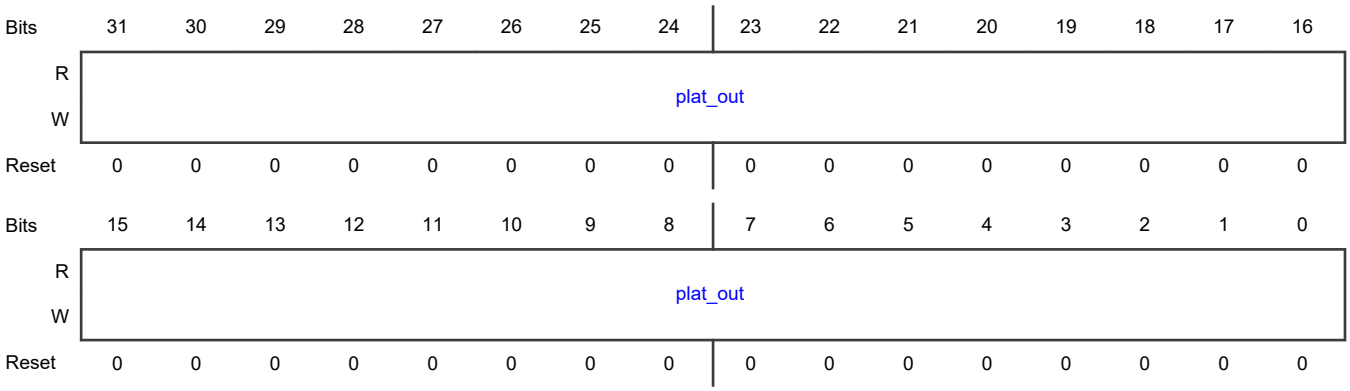
### Offset

| 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<br>plat_out | plat_out<br>Platform integration outputs.<br>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. |

### 13.2.20 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.

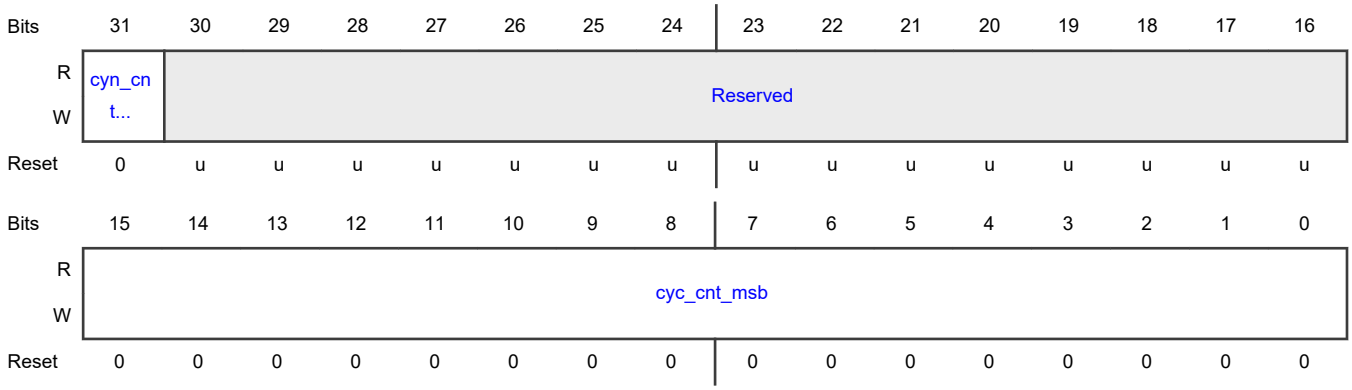
**NOTE**

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<br>cyn_cnt_en | cyn_cnt_en<br>Cycle count enable.<br>This bit controls if the Cycle Counter is enabled (counting) or disabled (stopped).<br>0b - Counter is disabled. The counter is not incremented.<br>1b - Counter is enabled. The counter is incremented by one VSPA clock. |
| 30-16<br>— | -<br>Reserved |
| 15-0<br>cyc_cnt_msb | cyc_cnt_msb<br>Cycle count MSB<br>This field contains the upper 16-bit, cyc_count[47:32], of the 48-bit cycle counter.<br>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. |

## 13.2.21 Cycle Counter LSB Register (CYC_COUNTER_LSB)

### Offset

| Register | Offset |
|---|---|
| CYC_COUNTER_LSB | 9Ch |

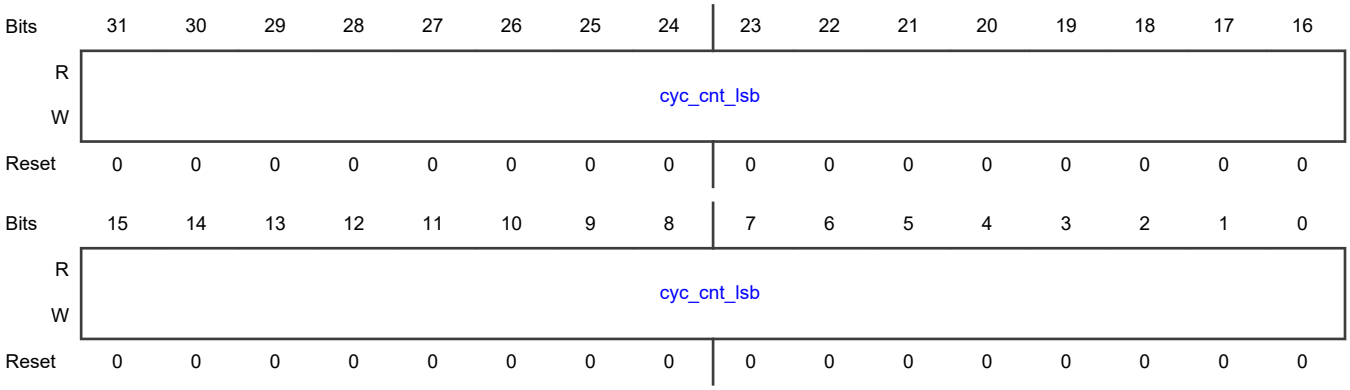
### Function

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<br>cyc_cnt_lsb | cyc_cnt_lsb<br>Cycle count LSB.<br>This field contains the lower 32-bit, cyc_count[31:0], of the 48-bit cycle counter.<br>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. |

## 13.2.22 DMEM/PRAM Address (DMA_DMEM_PRAM_ADDR)

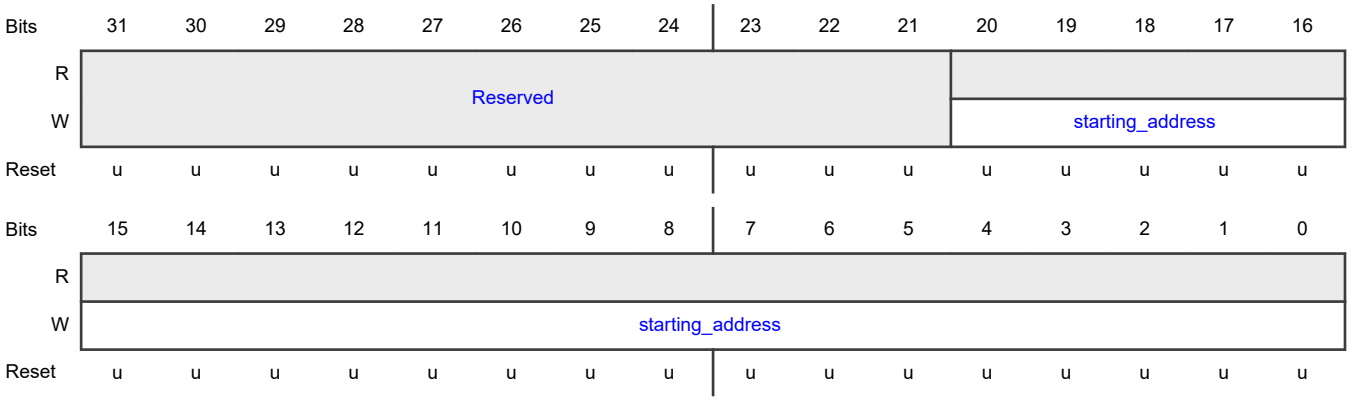
### Offset

| 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<br>— | -<br>Reserved |
| 20-0<br>starting_address | starting_address<br>Starting address<br>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.<br>• 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.<br>• 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.<br>• 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.<br>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. |

## 13.2.23 DMA AXI Address (DMA_AXI_ADDRESS)

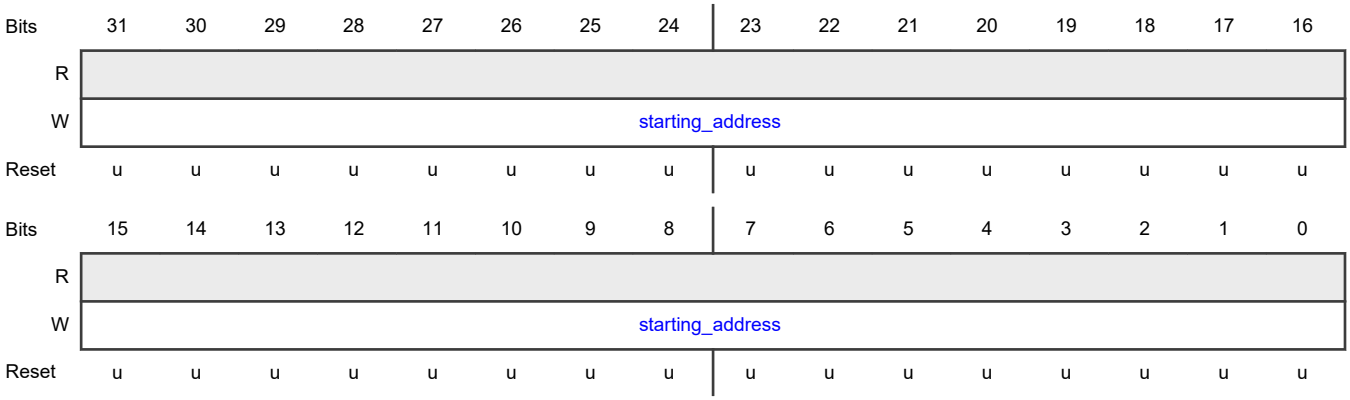
### Offset

| 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<br>starting_address | starting_address<br>Starting address<br>This field is a 32-bit field.<br>• 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.<br>• 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.<br>Settings:<br>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. |

## 13.2.24 AXI Byte Count register (DMA_AXI_BYTE_CNT)

### Offset

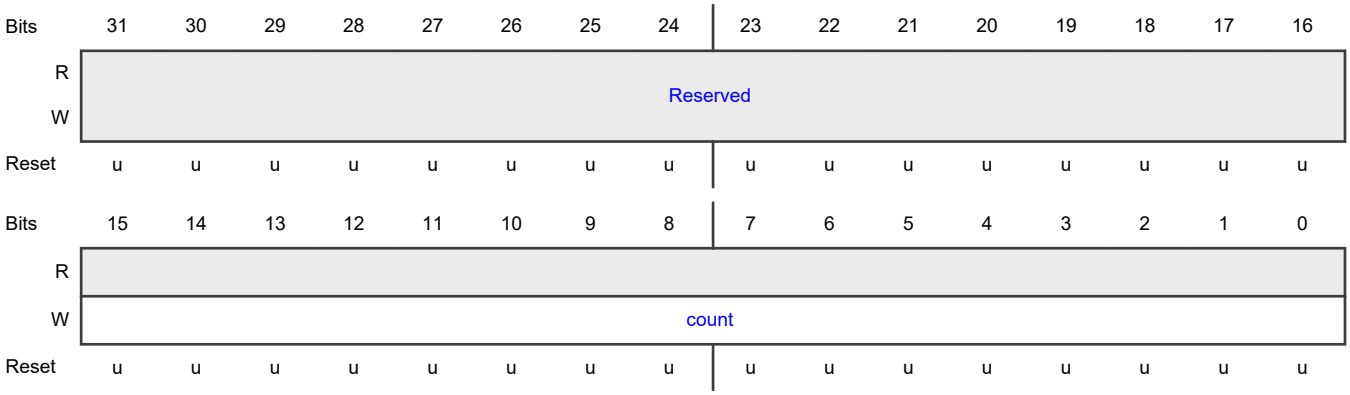
| Register | Offset |
|---|---|
| DMA_AXI_BYTE_CNT | B8h |

### 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<br>— | -<br>Reserved |
| 15-0<br>count | count<br>The number of bytes that get transferred.<br>Maximum transfer size is 65536 bytes.<br>• 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.<br>• 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.<br>• A configuration error will result when transferring to IPPU PRAM unless the total number of bytes transferred is an integer multiple of 4 bytes.<br>Settings:<br>0=65536 bytes, 1=1 byte, 2=2 bytes, and so on.<br>When the DMA is programmed for a DI storage operation, this register must be programmed to the number of data values to be stored. |

## 13.2.25 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.

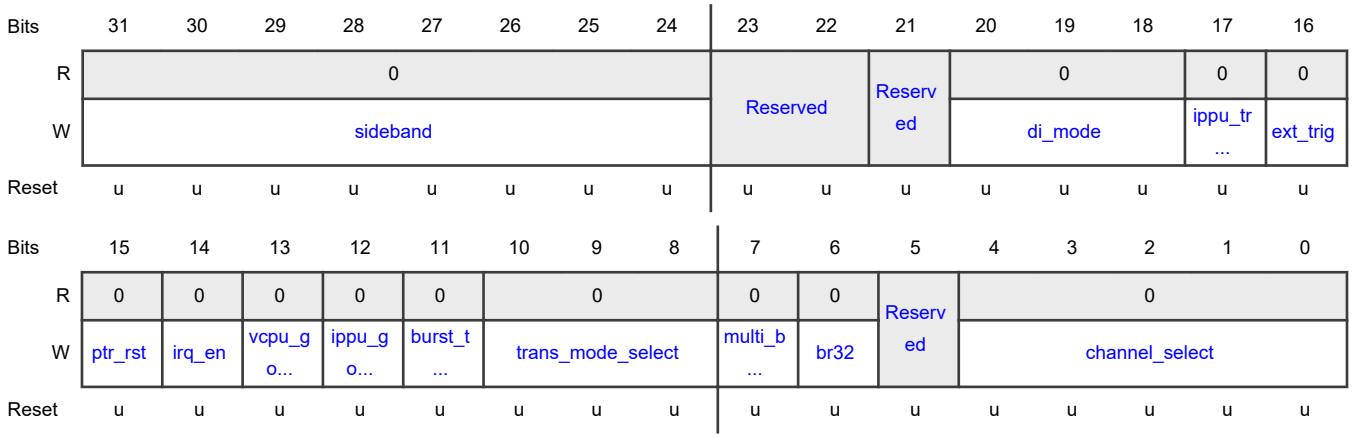
Since VCPU and a host processor can both use the DMA, there are separate sets of registers for each of them. Both sets of registers are located at the same addresses; the DMA detects whether writes come from VCPU or from the host, and updates the appropriate registers.

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<br>sideband | sideband<br>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 |
| 23-22<br>— | -<br>Reserved |
| 21<br>— | -<br>Reserved |
| 20-18<br>di_mode | Deinterleaving mode<br>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.<br>000b - reserved<br>001b - 8-bit deinterleave<br>010b - 4-bit deinterleave, AXI address MSB=0<br>011b - 4-bit deinterleave, AXI address MSB=1<br>100b - 16-bit deinterleave<br>101b - 32-bit deinterleave<br>110b - reserved<br>111b - reserved |
| 17<br>ippu_trig | IPPU trigger |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 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.<br>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.<br>0b - Ignore IPPU done events<br>1b - Wait for a single IPPU done event before enabling all transfers scheduled for this channel |
| 16<br>ext_trig | External trigger<br>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.<br>**NOTE**<br>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.<br>0b - Ignore dma_ext_trig input<br>1b - Wait for dma_ext_trig input before beginning each AXI burst |
| 15<br>ptr_rst | Pointer reset<br>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.<br>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.<br>0b - Ignored<br>1b - Upon completion of the scheduled transfers, assert ptr_rst_req |
| 14<br>irq_en | IRQ enable<br>Used to generate an IRQ to the host upon completion of the scheduled transfers.<br>0b - Ignored<br>1b - Generate IRQ upon completion of scheduled transfers |
| 13<br>vcpu_go_en | VCPU Go enable<br>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.<br>0b - Ignored<br>1b - Generate VCPU GO upon completion of scheduled transfers |
| 12<br>ippu_go_en | IPPU Go enable<br>Used to generate a IPPU GO upon completion of the scheduled transfers. |

*Table continues on the next page...*

*Table continued from the previous page...*

<table>
<tr><th>Field</th><th>Function</th></tr>
<tr><td></td><td>0b - Ignored<br>1b - Generate IPPU GO upon completion of scheduled transfers</td></tr>
<tr><td>11<br>burst_type</td><td>Burst type<br>Used to select between transferring to a memory or to a FIFO.<br>0b - Incrementing burst type<br>1b - Fixed burst type (all bursts go to the same AXI address)</td></tr>
<tr><td>10-8<br>trans_mode_select</td><td>Transfer mode select<br>Used to select characteristics of the desired DMA transfer, such as source and target memory and format conversion.<br>Table 121. Transfer Mode Select Encoding
<table>
<tr><th>Encoding</th><th>DMA from</th><th>DMA to</th><th>Notes</th></tr>
<tr><td>000</td><td>AXI</td><td>DMEM</td><td>No format conversion</td></tr>
<tr><td>001</td><td>AXI</td><td>DMEM</td><td>Format conversion from 16-bit 2's complement -> 16-bit sign magnitude</td></tr>
<tr><td>010</td><td>AXI</td><td>PRAM</td><td>Load VCPU PRAM</td></tr>
<tr><td>011</td><td>AXI</td><td>IPPU PRAM</td><td>Load IPPU PRAM</td></tr>
<tr><td>100</td><td>DMEM</td><td>AXI</td><td>Store and deinterleave data to AXI</td></tr>
<tr><td>101</td><td>u</td><td>u</td><td>Reserved</td></tr>
<tr><td>110</td><td>DMEM</td><td>AXI</td><td>No format conversion</td></tr>
<tr><td>111</td><td>DMEM</td><td>AXI</td><td>Format conversion from 16-bit sign magnitude -> 16-bit 2's complement</td></tr>
</table>
NOTE<br>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.</td></tr>
<tr><td>7<br>multi_burst</td><td>Multi-read burst enable<br>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.<br>Warnings:<br>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.</td></tr>
</table>

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 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.<br>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.<br>0b - Do only 1 AXI read burst per arbitration win<br>1b - Perform up to 4 AXI read bursts per arbitration win |
| 6<br>br32 | Bit reversal mode<br>Setting this bit when the transfer mode is 000 (AXI to DMEM, no format conversion) will cause the read data bits from AXI to be bit reversed within each 32-bit word. This bit is ignored for all AXI write commands.<br>This bit should not be set when the if the transfer mode is not 000. |
| 5<br>— | -<br>Reserved |
| 4-0<br>channel_select | Channel select<br>Selects the DMA channel to be used to execute the command.<br>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.<br>00000b - channel 0<br>00001b - channel 1<br>11111b - channel 31 |

### 13.2.26 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

| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R | dma_chan_n | | | | | | | | | | | | | | | |
| 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 | dma_chan_n | | | | | | | | | | | | | | | |
| W | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

### Fields

| Field | Function |
|---|---|
| 31-0<br>dma_chan_n | dma_chan_n<br>**NOTE**<br>Access to this field is non-standard and is described in detail below.<br>Read - DMA Channel n status for n=31 to 0<br>Each nth bit specifies if the nth DMA channel is enabled.<br>Write 1 - Disable/abort channel n and clear its FIFO.<br>Read:<br>0 Channel n Disabled.<br>1 Channel n Enabled. |

*Table continues on the next page...*

| Field | Function |
|---|---|
| | Write:<br>0 Do Nothing.<br>1 Disable/abort channel n, clear its FIFO entries. |

### 13.2.27 DMA IRQ Status (DMA_IRQ_STAT)

#### Offset

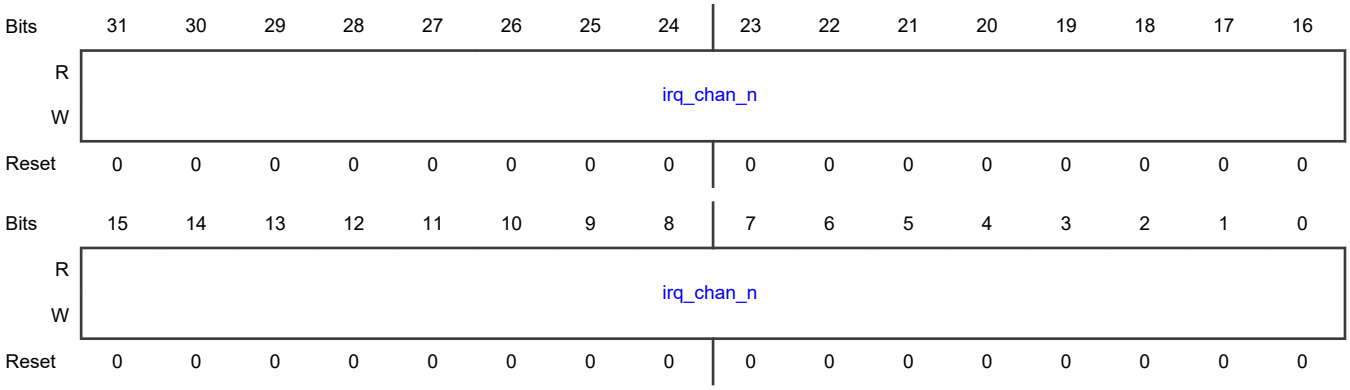
| Register | Offset |
|---|---|
| DMA_IRQ_STAT | C4h |

#### 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 |
|---|---|
| 31-0<br>irq_chan_n | irq_chan_n<br>**NOTE**<br>Access to this field is non-standard and is described in detail below.<br>IRQ channel n for n=31 to 0<br>Each nth bit specifies whether the nth DMA channel asserted an interrupt request upon completion of all of its scheduled transfers. |

*Table continues on the next page...*

| Field | Function |
|---|---|
| | These bits are write 1 to clear, and they can only be cleared by the external IPbus master, not by VCPU.<br>00000000000000000000000000000000b - Channel n interrupt not requested<br>00000000000000000000000000000001b - Channel n interrupt requested |

### 13.2.28 DMA Complete Status (DMA_COMP_STAT)

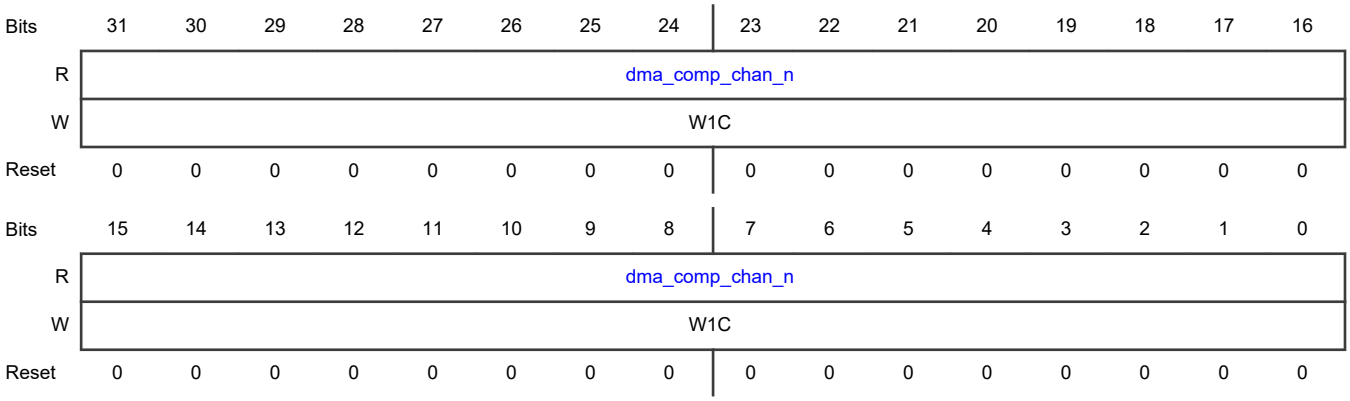
#### Offset

| 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 |
|---|---|
| 31-0<br>dma_comp_chan_n | dma_comp_chan_n<br>DMA completed channel n for n=31 to 0<br>Each nth bit specifies whether the nth DMA channel has completed all of its scheduled transfers.<br>00000000000000000000000000000000b - Channel n transfers have not completed.<br>00000000000000000000000000000001b - Channel n transfers have completed. |

## 13.2.29 DMA Transfer Error Status (DMA_XFRERR_STAT)

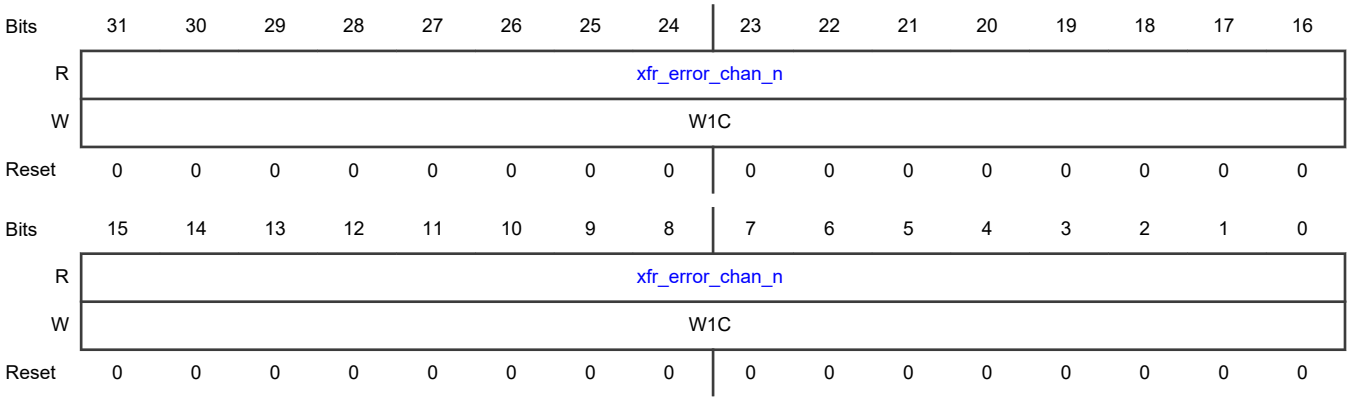
### Offset

| 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 |
|---|---|
| 31-0<br>xfr_error_chan_n | dma_comp_chan_n<br>DMA transfer error channel n for n=31 to 0<br>Each nth bit indicates whether the nth DMA channel has detected AXI transaction errors during a programmed transfer.<br>00000000000000000000000000000000b - Channel n AXI transfers have no errors, or have not completed.<br>00000000000000000000000000000001b - Channel n AXI transfers have errors. |

## 13.2.30 DMA Configuration Error Status (DMA_CFGERR_STAT)

### Offset

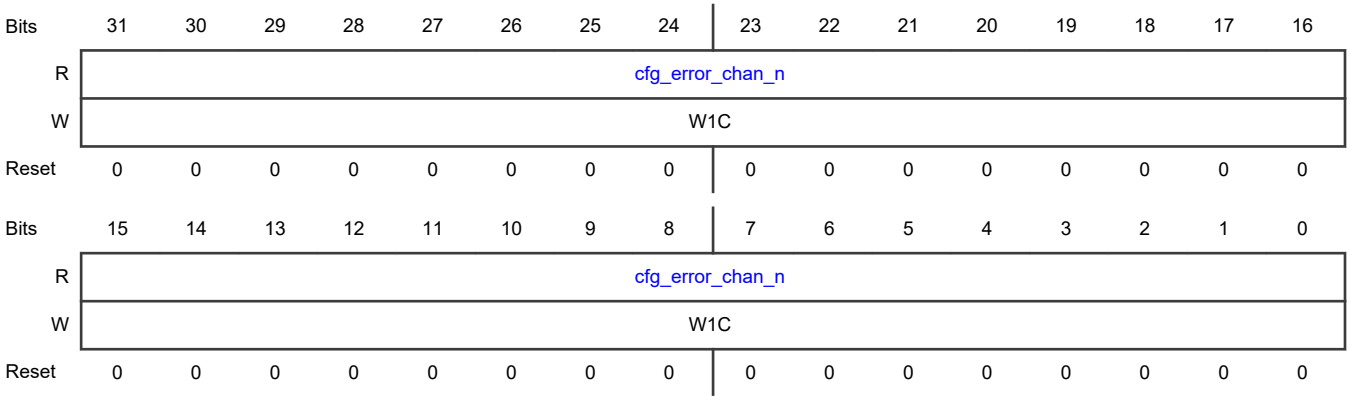
| Register | Offset |
|---|---|
| DMA_CFGERR_STAT | D0h |

### 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 |
|---|---|
| 31-0<br>cfg_error_chan_n | cfg_error_chan_n<br>DMA configuration error for channel n for n=31 to 0<br>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.<br>00000000000000000000000000000000b - No attempt has been made to activate channel n with an invalid configuration.<br>00000000000000000000000000000001b - An attempt was made to activate channel n with an invalid configuration specified by the associated channel configuration registers. |

## 13.2.31 DMA Transfer Running Status (DMA_XRUN_STAT)

### Offset

| Register | Offset |
|---|---|
| DMA_XRUN_STAT | D4h |

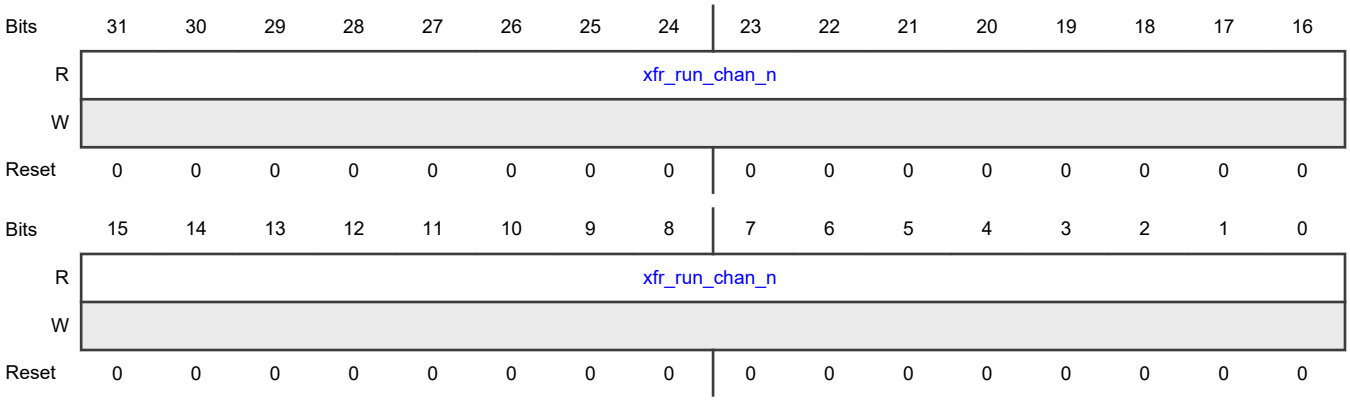
### Function

- 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 |
|---|---|
| 31-0<br>xfr_run_chan_n | xfr_run_chan_n<br>Transfer running for channel n for n=31 to 0<br>Each nth bit indicates whether the nth DMA channel is actively doing transfers.<br>00000000000000000000000000000000b - No transfer activity is presently happening for the associated channel.<br>00000000000000000000000000000001b - Transfer activity is presently happening for the associated channel. |

## 13.2.32 DMA Go Status (DMA_GO_STAT)

### Offset

| Register | Offset |
|---|---|
| DMA_GO_STAT | D8h |

### Function

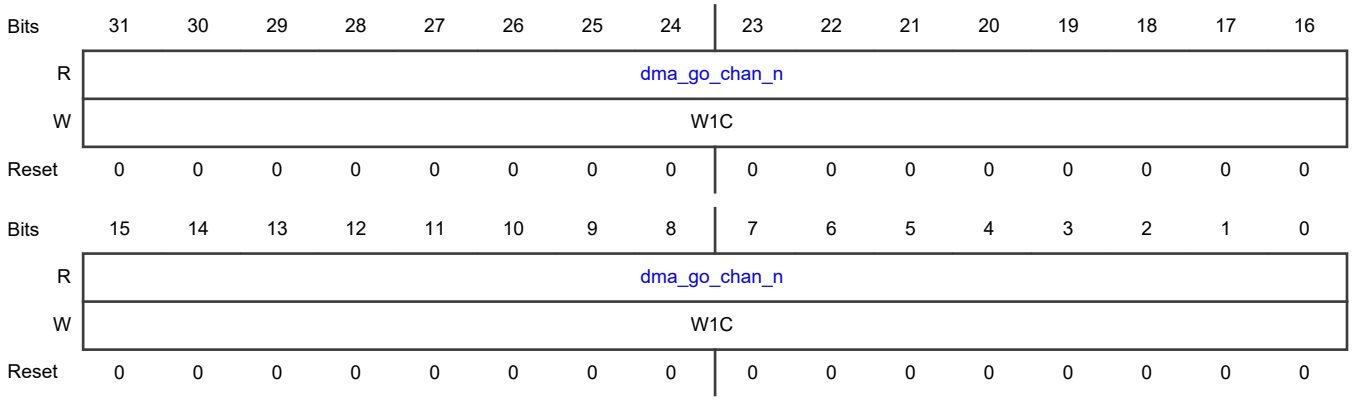
- The DMA_GO_STAT register bits are write 1 to clear status bits.

**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 |
|---|---|
| 31-0<br>dma_go_chan_n | dma_go_chan_n<br>**NOTE**<br>Access to this field is non-standard and is described in detail below.<br>VCPU go generated by channel n for n=31 to 0<br>Each nth bit indicates whether the nth DMA channel initiated a GO.<br>0 read - Channel did not initiate a GO<br>1 read - Channel initiated a GO<br>0 write - Ignored<br>1 write - Clear the bit (if set) |

## 13.2.33 DMA FIFO Availability Status (DMA_FIFO_STAT)

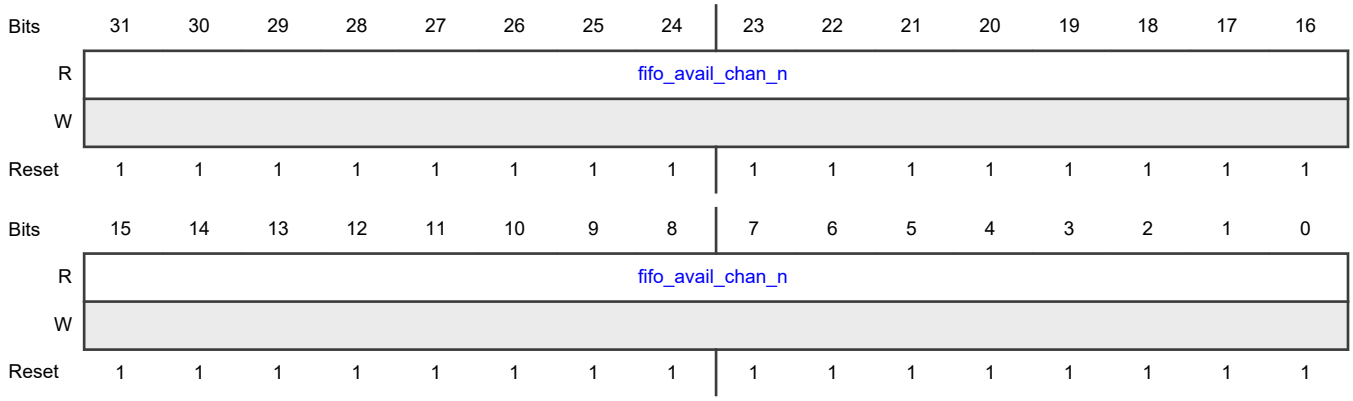
### Offset

| 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 |
|---|---|
| 31-0<br>fifo_avail_chan_n | fifo_avail_chan_n<br>Empty FIFO entry available for channel n for n=31 to 0<br>Each nth bit indicates whether the nth DMA channel has an empty FIFO entry, and is thus capable of accepting a new command.<br>00000000000000000000000000000000b - No FIFO entry is available for the associated channel.<br>00000000000000000000000000000001b - One or more FIFO entries are available for the associated channel. |

## 13.2.34 AUX DMA status register (AD_STATUS)

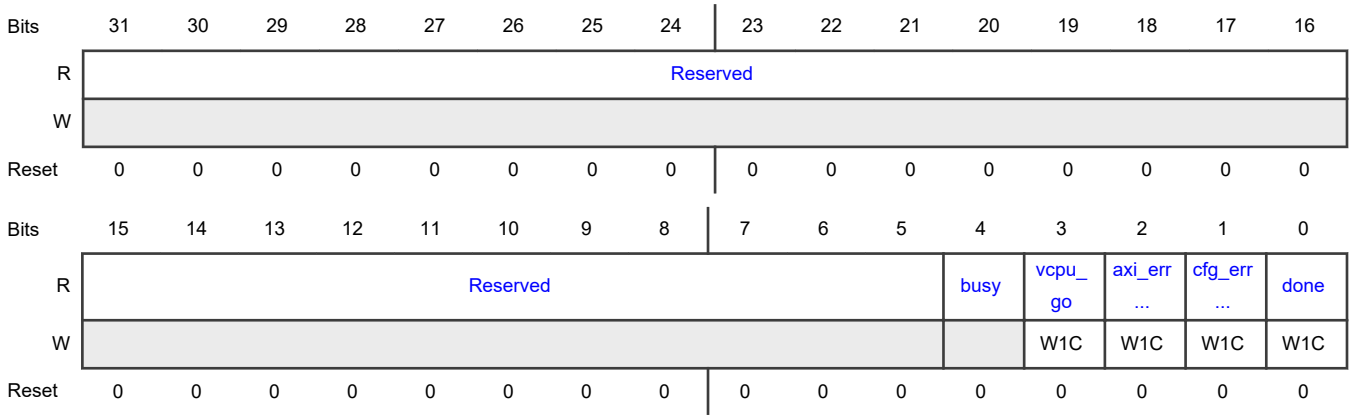
### Offset

| Register | Offset |
|---|---|
| AD_STATUS | E0h |

### Function

Indicates the overall status of the AUX DMA. Shows if the AUX DMA is busy, done, or encountered an AXI data error. Also indicates if a VCPU go was triggered, or if the AUX DMA was initialized incorrectly and ignored a command.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-5<br>— | -<br>Reserved |
| 4<br>busy | AUX DMA busy<br>Indicates whether the AUX DMA is busy executing a command.<br>1 - the AUX DMA is busy.<br>0 - the AUX DMA is not busy. |
| 3<br>vcpu_go | VCPU go request<br>Indicates whether a VCPU go is being requested .<br>This bit can be cleared by writing a 1 to it (W1C) from the VCPU. A host cannot clear this bit.<br>1 - a VCPU go is being requested due to the completion of an AUX DMA command for which the vcpu_go_en bit of the AD_CTRL register was set.<br>0 - no VCPU go is being requested. |
| 2<br>axi_error | AXI error<br>Indicates whether the AUX DMA encounter an AXI error during command execution.<br>This bit can be cleared by writing a 1 to it (W1C), from either the VCPU or a host.<br>1 - an AXI error was detected<br>0 - a command was not detected since this bit was cleared. |
| 1<br>cfg_error | configuration error<br>Indicates whether an AUX DMA command was ignored due to a configuration error.<br>This bit can be cleared by writing a 1 to it (W1C), from either the VCPU or a host.<br>1 - a command was ignored due to a configuration error.<br>0 - a command was not ignored due to a configuration error since this bit was cleared. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | A configuration error will result if the AD_CTRL register is written, if any of the following is true.<br>- The 6 LSBs of AD_PRAM_ADDR are non-zero.<br>- The 6 LSBs of AD_AXI_ADDR are non-zero.<br>- The 6 LSBs of AD_BYTE_COUNT are non-zero. |
| 0<br>done | done<br>Indicates whether an AUX DMA command was completed.<br>This bit can be cleared by writing a 1 to it (W1C), from either the VCPU or a host.<br>1 - a command was completed<br>0 - a command was not completed since this bit was cleared. |

### 13.2.35 Load Register File Control register (Slow read register) (LD_RF_CONTROL)

#### Offset

| Register | Offset |
|---|---|
| LD_RF_CONTROL | 100h |

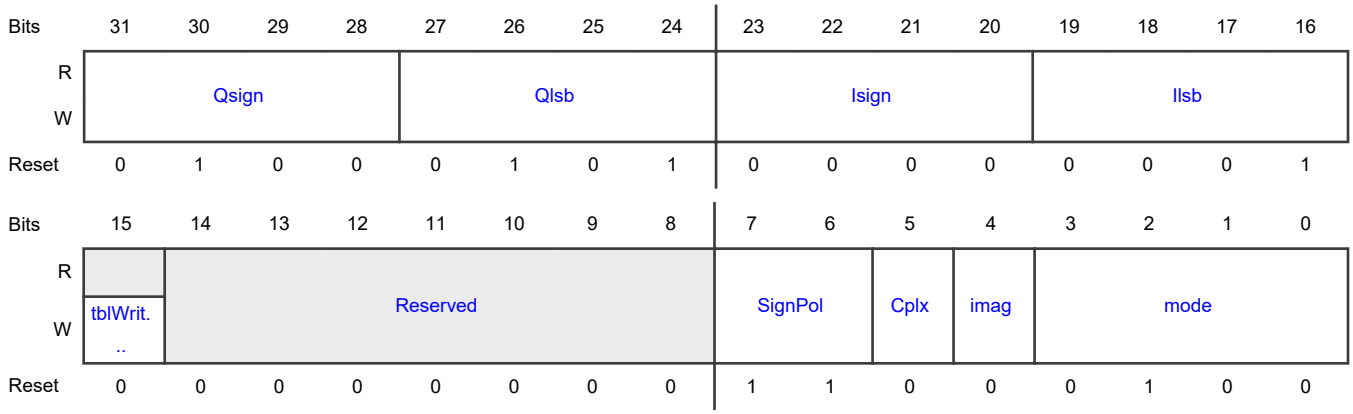
#### Function

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<br>Qsign | Qsign<br>Sign bit of imaginary component for QAM 256 and QAM 1024 |
| 27-24<br>Qlsb | Qlsb<br>Least significant bit of imaginary component for QAM 256 and QAM 1024 |
| 23-20<br>Isign | Isign<br>Sign bit of real component for QAM 256 and QAM 1024 |
| 19-16<br>Ilsb | Ilsb<br>Least significant bit of real component for QAM 256 and QAM 1024 |
| 15<br>tblWriteEn_b | QAM Table Write Enable<br>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. |
| 14-8<br>— | -<br>Reserved |
| 7-6<br>SignPol | SignPol<br>Polarity of sign bits<br>Bit 7 indicates the polarity of the sign bit of the imaginary (Q) component for QAM 256 and QAM 1024<br>Bit 6 indicates the polarity of the sign bit of the real (I) component for QAM 256 and QAM 1024<br>0 - non-inverted sign<br>1 - inverted sign |
| 5<br>Cplx | Complex<br>Generate complex values for the QAM data |
| 4<br>imag | imag<br>Imaginary Load Select<br>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.<br>0b - real<br>1b - imag |
| 3-0<br>mode | mode<br>Modulation Mode Select |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 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 802.15.4g 16QAM and 64QAM.<br>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.<br>1 = 2 coefficients (BPSK)<br>2 = 4 coefficients (4PAM)<br>4 = 16 coefficients (16Q/PAM)<br>6 = 64 coefficients (64Q/PAM)<br>8 = 256 coefficients (256Q/PAM)<br>10 = 1024 coefficients (1024Q/PAM)<br>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 default 802.11 16QAM values.<br>Writing 6 to the mode bits will automatically update these registers to default 802.15.4g 64QAM values.<br>Writing 8 to the mode bits will automatically update these registers to default 802.11 256QAM values.<br>Writing 10 to the mode bits will automatically update these registers to default 802.11 1024QAM values.<br>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. |

### 13.2.36 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

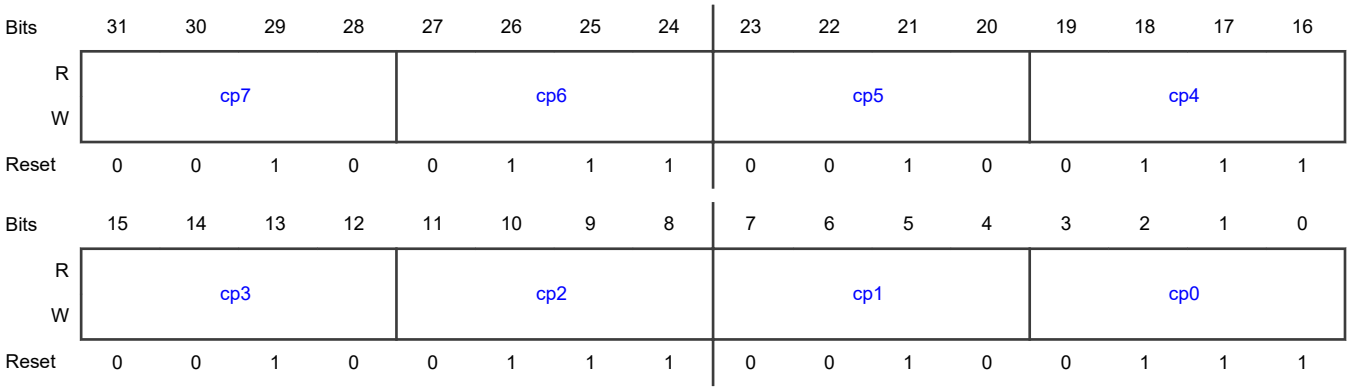
This register is a Slow read register.

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:<br>X=0-15 for mode=0100<br>X=0-7 for all other modes | 00 - 1<br>01 - 3<br>10 - -1<br>11 - -3 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5<br>0b011x - 7<br>0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5<br>0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5<br>0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-28<br>cp7 | cp7<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 27-24<br>cp6 | cp6<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp5 | cp5<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp4 | cp4<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 15-12<br>cp3 | cp3<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp2 | cp2<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp1 | cp1<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 110xb - -5<br>111xb - -7 |
| 3-0<br>cp0 | cp0<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.37 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.

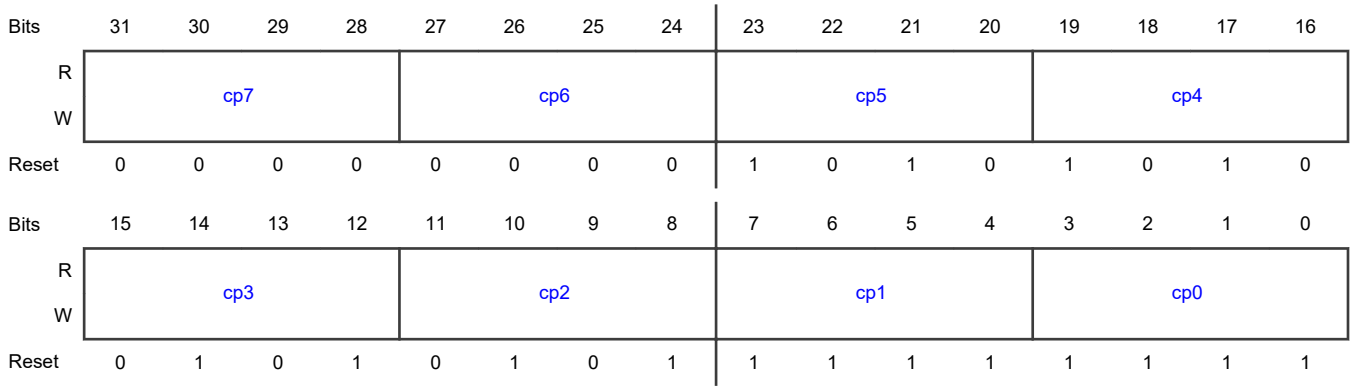
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

| Register bit-fields | Mode==0100 | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|---|
| cpX:<br>X=0-15 for mode=0100<br>X=0-7 for all other modes | 00 - 1<br>01 - 3<br>10 - -1<br>11 - -3 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5<br>0b011x - 7 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5<br>0bx011 - 7 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5<br>0b0011 - 7 |

*Table continues on the next page...*

| Register bit-fields | Mode==0100 | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|---|
| | | 0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-28<br>cp7 | cp7<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 110xb - -5<br>111xb - -7 |
| 27-24<br>cp6 | cp6<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp5 | cp5<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp4 | cp4<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15-12<br>cp3 | cp3<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp2 | cp2<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp1 | cp1<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp0 | cp0<br>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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.38 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:<br>X=0-7 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5 |

*Table continues on the next page...*

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| | 0b011x - 7<br>0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram

| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R<br>W | cp15 | | | | cp14 | | | | cp13 | | | | cp12 | | | |
| Reset | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 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<br>W | cp11 | | | | cp10 | | | | cp9 | | | | cp8 | | | |
| Reset | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

### Fields

| Field | Function |
|---|---|
| 31-28<br>cp15 | cp15<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 101xb - -3<br>110xb - -5<br>111xb - -7 |
| 27-24<br>cp14 | cp14<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp13 | cp13<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp12 | cp12<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15-12<br>cp11 | cp11<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp10 | cp10<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp9 | cp9<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp8 | cp8<br>Real 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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.39 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.

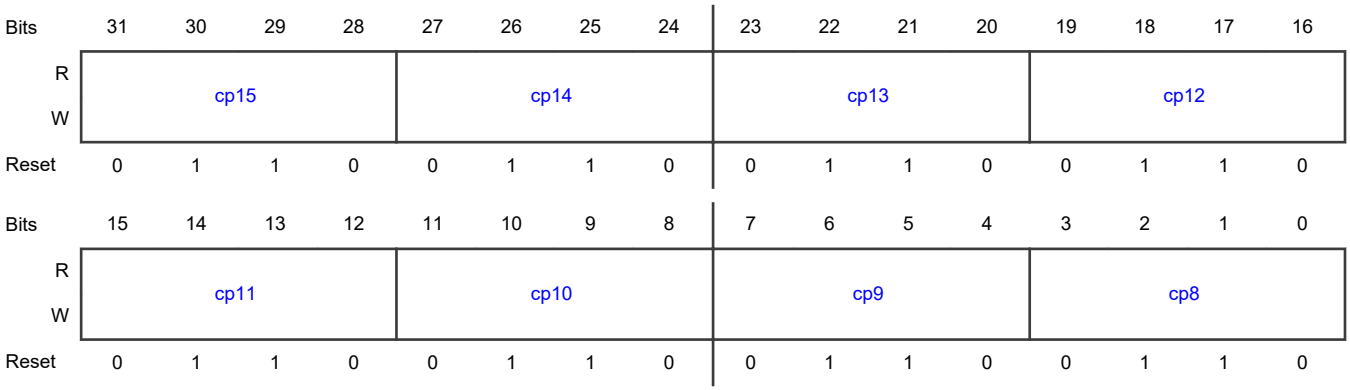
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| cpX:<br>X=0-7 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5 |

*Table continues on the next page...*

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| | 0b011x - 7<br>0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-28<br>cp15 | cp15<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 101xb - -3<br>110xb - -5<br>111xb - -7 |
| 27-24<br>cp14 | cp14<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp13 | cp13<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp12 | cp12<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15-12<br>cp11 | cp11<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp10 | cp10<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp9 | cp9<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp8 | cp8<br>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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.40 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.

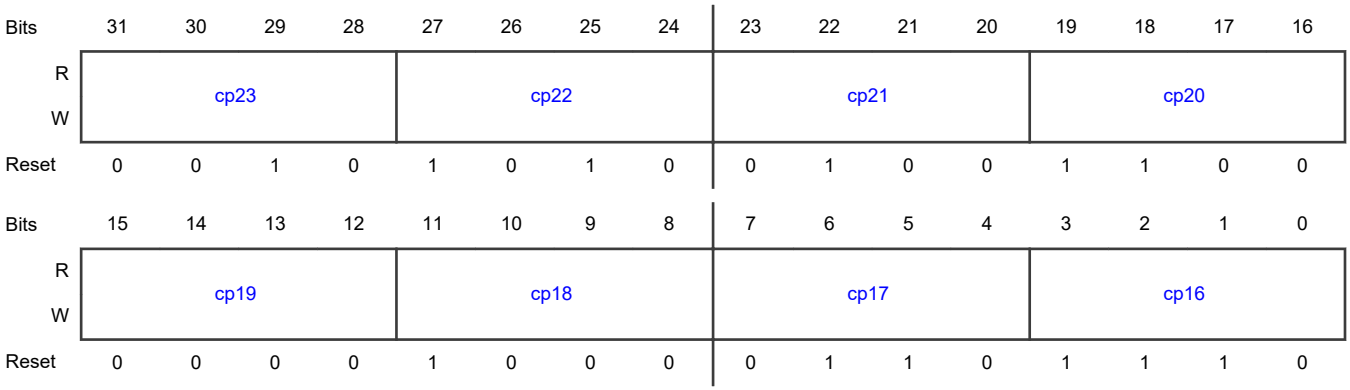
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| cpX:<br>X=0-7 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5 |

*Table continues on the next page...*

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| | 0b011x - 7<br>0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-28<br>cp23 | cp23<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 101xb - -3<br>110xb - -5<br>111xb - -7 |
| 27-24<br>cp22 | cp22<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp21 | cp21<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp20 | cp20<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15-12<br>cp19 | cp19<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp18 | cp18<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp17 | cp17<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp16 | cp16<br>Real 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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.41 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.

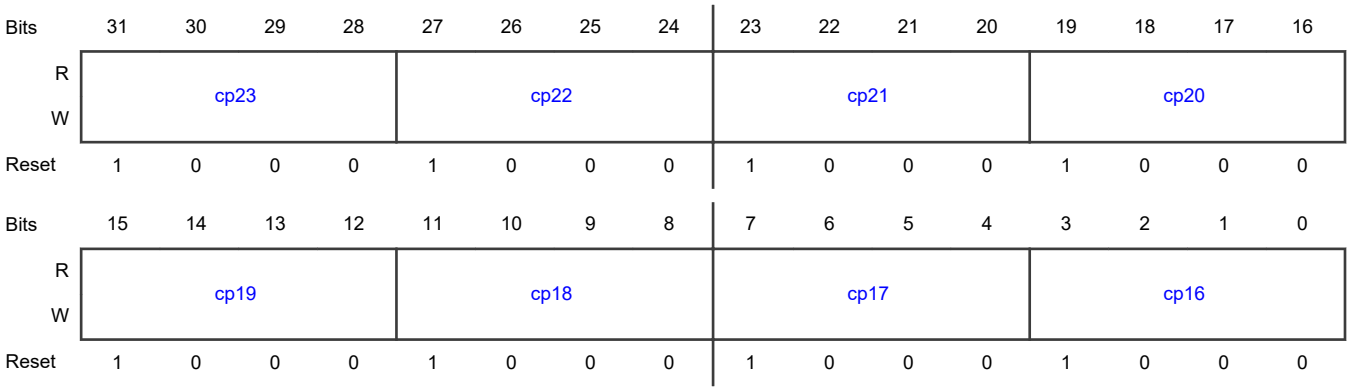
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| cpX:<br>X=0-7 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5 |

*Table continues on the next page...*

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| | 0b011x - 7<br>0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-28<br>cp23 | cp23<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 101xb - -3<br>110xb - -5<br>111xb - -7 |
| 27-24<br>cp22 | cp22<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp21 | cp21<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp20 | cp20<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15-12<br>cp19 | cp19<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp18 | cp18<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp17 | cp17<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp16 | cp16<br>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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.42 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:<br>X=0-7 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5 |

*Table continues on the next page...*

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| | 0b011x - 7<br>0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram

| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R<br>W | cp31 | | | | cp30 | | | | cp29 | | | | cp28 | | | |
| Reset | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 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<br>W | cp27 | | | | cp26 | | | | cp25 | | | | cp24 | | | |
| Reset | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

### Fields

| Field | Function |
|---|---|
| 31-28<br>cp31 | cp31<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 101xb - -3<br>110xb - -5<br>111xb - -7 |
| 27-24<br>cp30 | cp30<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp29 | cp29<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp28 | cp28<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15-12<br>cp27 | cp27<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp26 | cp26<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp25 | cp25<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp24 | cp24<br>Real 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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.43 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.

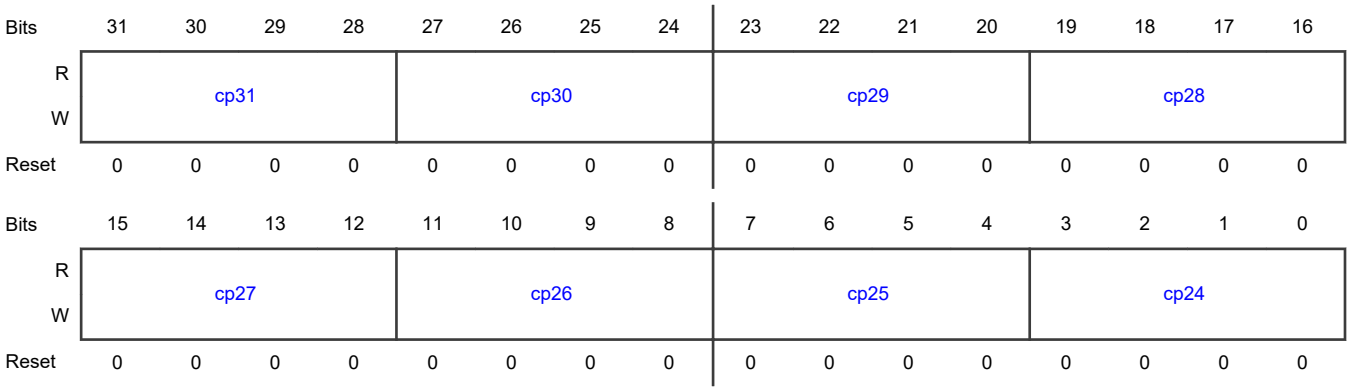
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| cpX:<br>X=0-7 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5 |

*Table continues on the next page...*

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| | 0b011x - 7<br>0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-28<br>cp31 | cp31<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 101xb - -3<br>110xb - -5<br>111xb - -7 |
| 27-24<br>cp30 | cp30<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp29 | cp29<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp28 | cp28<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15-12<br>cp27 | cp27<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp26 | cp26<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp25 | cp25<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp24 | cp24<br>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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.44 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:<br>X=0-7 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5 |

*Table continues on the next page...*

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| | 0b011x - 7<br>0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram

| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R<br>W | cp39 | | | | cp38 | | | | cp37 | | | | cp36 | | | |
| Reset | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 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<br>W | cp35 | | | | cp34 | | | | cp33 | | | | cp32 | | | |
| Reset | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

### Fields

| Field | Function |
|---|---|
| 31-28<br>cp39 | cp39<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 101xb - -3<br>110xb - -5<br>111xb - -7 |
| 27-24<br>cp38 | cp38<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp37 | cp37<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp36 | cp36<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15-12<br>cp35 | cp35<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp34 | cp34<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp33 | cp33<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp32 | cp32<br>Real 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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.45 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.

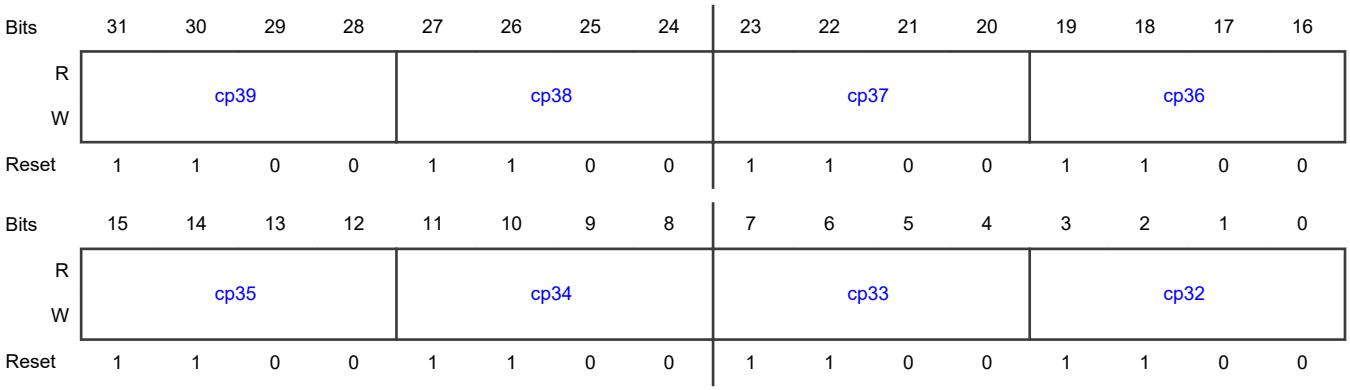
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| cpX:<br>X=0-7 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5 |

*Table continues on the next page...*

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| | 0b011x - 7<br>0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-28<br>cp39 | cp39<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 101xb - -3<br>110xb - -5<br>111xb - -7 |
| 27-24<br>cp38 | cp38<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp37 | cp37<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp36 | cp36<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15-12<br>cp35 | cp35<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp34 | cp34<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp33 | cp33<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp32 | cp32<br>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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.46 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.

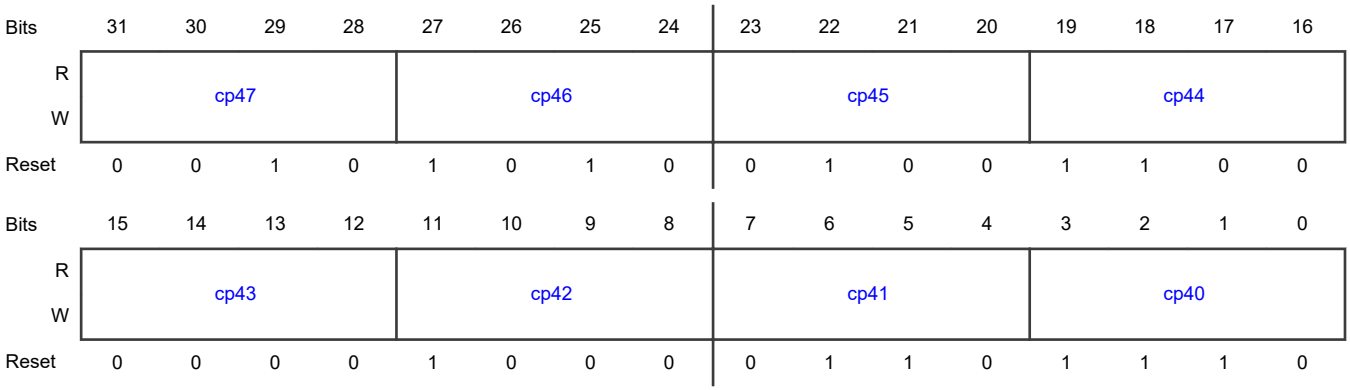
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| cpX:<br>X=0-7 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5 |

*Table continues on the next page...*

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| | 0b011x - 7<br>0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-28<br>cp47 | cp47<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 101xb - -3<br>110xb - -5<br>111xb - -7 |
| 27-24<br>cp46 | cp46<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp45 | cp45<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp44 | cp44<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15-12<br>cp43 | cp43<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp42 | cp42<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp41 | cp41<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp40 | cp40<br>Real 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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.47 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.

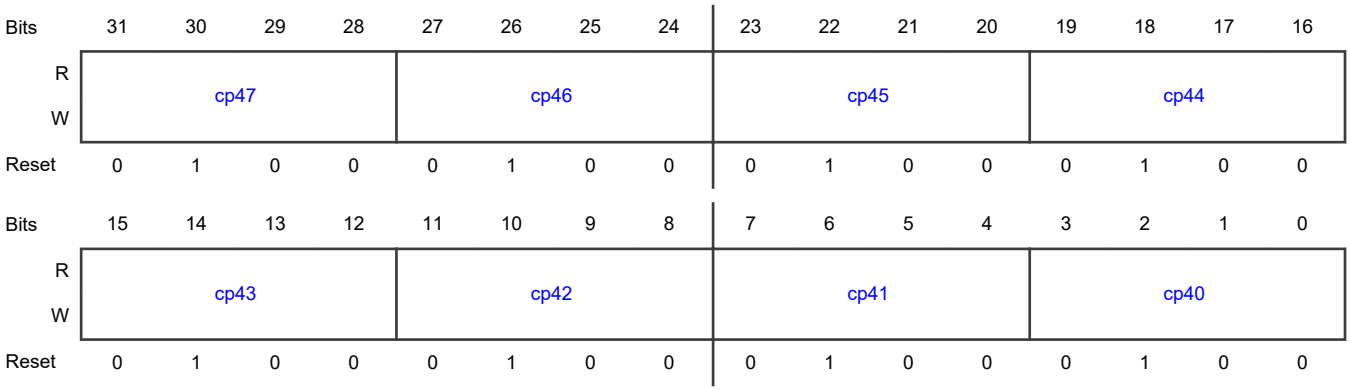
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| cpX:<br>X=0-7 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5 |

*Table continues on the next page...*

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| | 0b011x - 7<br>0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-28<br>cp47 | cp47<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 101xb - -3<br>110xb - -5<br>111xb - -7 |
| 27-24<br>cp46 | cp46<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp45 | cp45<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp44 | cp44<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15-12<br>cp43 | cp43<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp42 | cp42<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp41 | cp41<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp40 | cp40<br>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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.48 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.

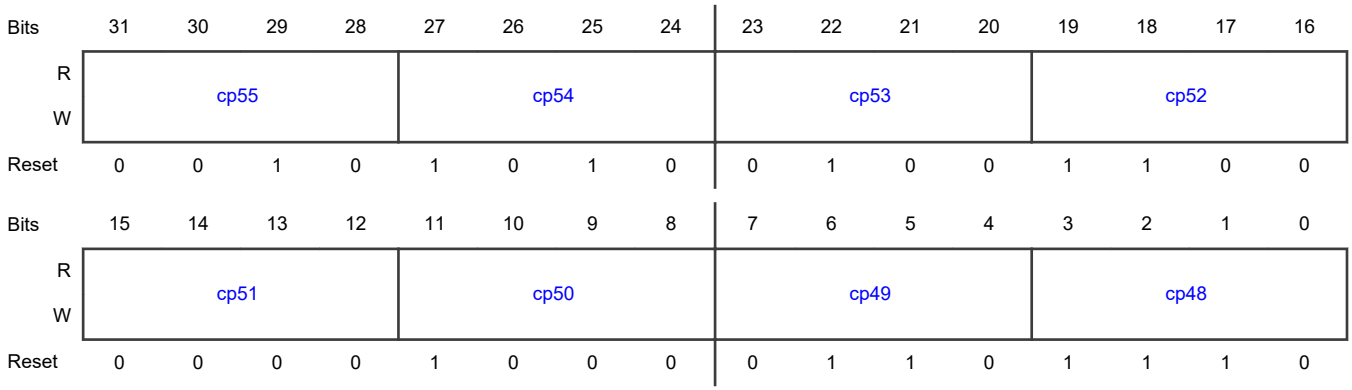
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| cpX:<br>X=0-7 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5 |

*Table continues on the next page...*

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| | 0b011x - 7<br>0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-28<br>cp55 | cp55<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 101xb - -3<br>110xb - -5<br>111xb - -7 |
| 27-24<br>cp54 | cp54<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp53 | cp53<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp52 | cp52<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15-12<br>cp51 | cp51<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp50 | cp50<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp49 | cp49<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp48 | cp48<br>Real 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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.49 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.

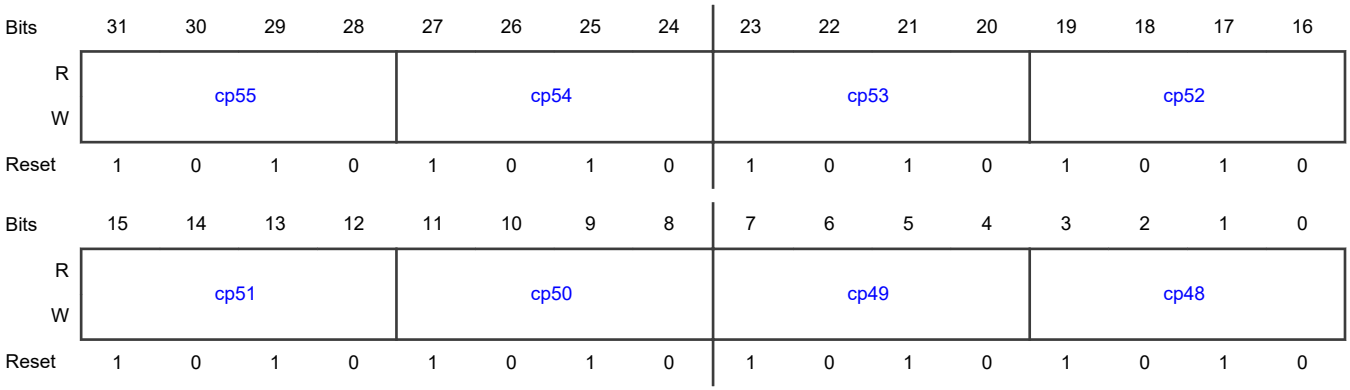
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| cpX:<br>X=0-7 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5 |

*Table continues on the next page...*

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| | 0b011x - 7<br>0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-28<br>cp55 | cp55<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 101xb - -3<br>110xb - -5<br>111xb - -7 |
| 27-24<br>cp54 | cp54<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp53 | cp53<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp52 | cp52<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15-12<br>cp51 | cp51<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp50 | cp50<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp49 | cp49<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp48 | cp48<br>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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.50 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.

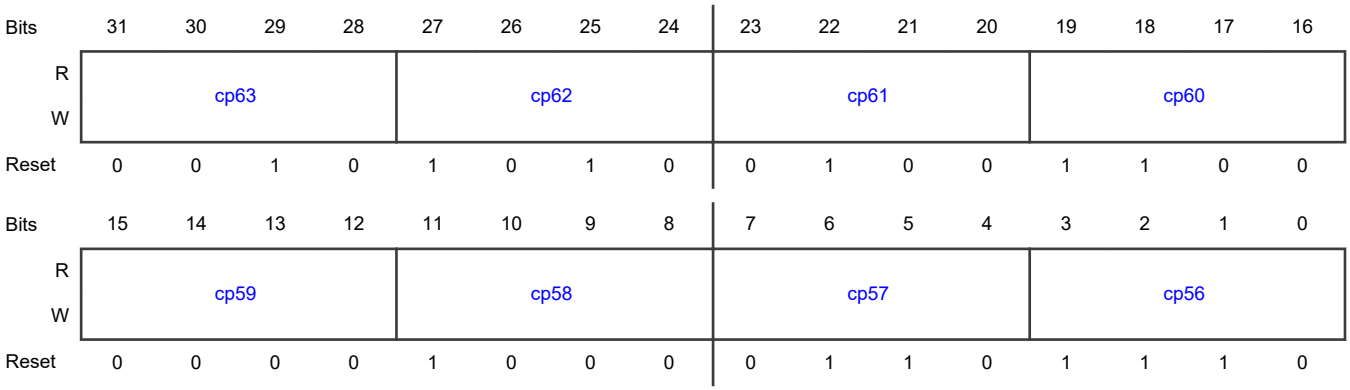
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| cpX:<br>X=0-7 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5 |

*Table continues on the next page...*

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| | 0b011x - 7<br>0b100x - -1<br>0b101x - -3<br>0b110x - -5<br>0b111x - -7 | 0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11<br>0bx110 - 13<br>0bx111 - 15 | 0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11<br>0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-28<br>cp63 | cp63<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 101xb - -3<br>110xb - -5<br>111xb - -7 |
| 27-24<br>cp62 | cp62<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp61 | cp61<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp60 | cp60<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15-12<br>cp59 | cp59<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp58 | cp58<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp57 | cp57<br>Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp56 | cp56<br>Real 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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.51 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.

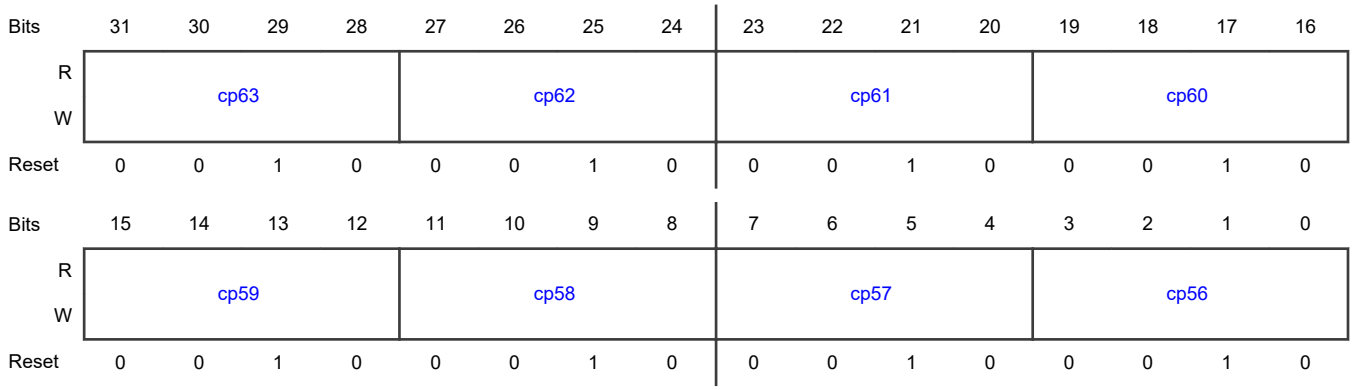
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| cpX:<br>X=0-7 | 0b000x - 1<br>0b001x - 3<br>0b010x - 5<br>0b011x - 7<br>0b100x - -1<br>0b101x - -3 | 0bx000 - 1<br>0bx001 - 3<br>0bx010 - 5<br>0bx011 - 7<br>0bx100 - 9<br>0bx101 - 11 | 0b0000 - 1<br>0b0001 - 3<br>0b0010 - 5<br>0b0011 - 7<br>0b0100 - 9<br>0b0101 - 11 |

*Table continues on the next page...*

| Register bit-fields | Mode==0110 | Mode==1000 | Mode==1010 |
|---|---|---|---|
| | 0b110x - -5<br>0b111x - -7 | 0bx110 - 13<br>0bx111 - 15 | 0b0110 - 13<br>0b0111 - 15<br>0b1000 - 17<br>0b1001 - 19<br>0b1010 - 21<br>0b1011 - 23<br>0b1100 - 25<br>0b1101 - 27<br>0b1110 - 29<br>0b1111 - 31 |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-28<br>cp63 | cp63<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 27-24<br>cp62 | cp62<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 23-20<br>cp61 | cp61<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 19-16<br>cp60 | cp60<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 15-12<br>cp59 | cp59<br>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<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 11-8<br>cp58 | cp58<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 7-4<br>cp57 | cp57<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3<br>010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |
| 3-0<br>cp56 | cp56<br>Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.<br>000xb - 1<br>001xb - 3 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 010xb - 5<br>011xb - 7<br>100xb - -1<br>101xb - -3<br>110xb - -5<br>111xb - -7 |

### 13.2.52 VCPU Go Address (VCPU_GO_ADDR)

#### Offset

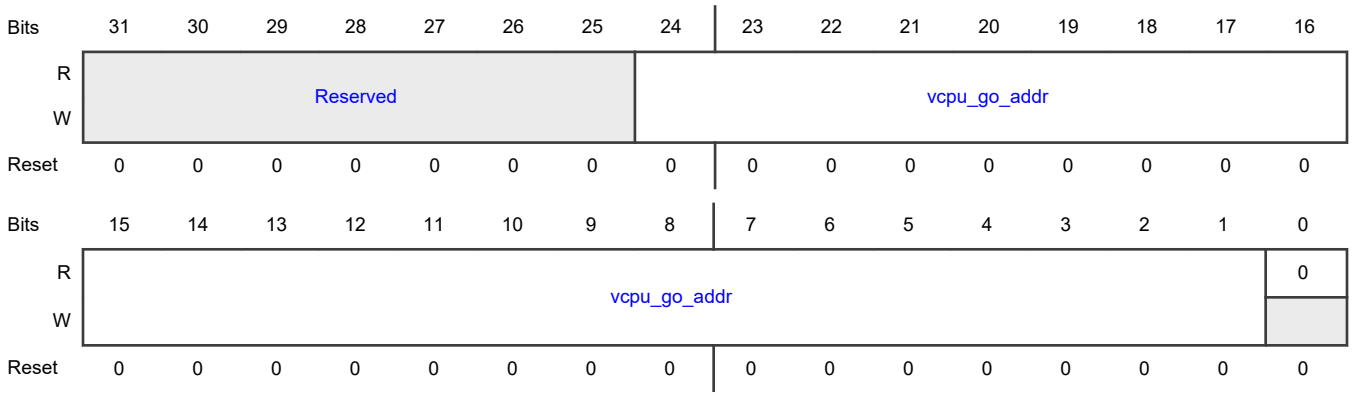
| Register | Offset |
|---|---|
| VCPU_GO_ADDR | 180h |

#### Function

This register can be written by either the VCPU or a host processor.

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<br>— | -<br>Reserved |
| 24-1 | vcpu_go_addr |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 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.<br>**NOTE**<br>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<br>— | -<br>Reserved |

### 13.2.53 VCPU Go Stack (VCPU_GO_STACK)

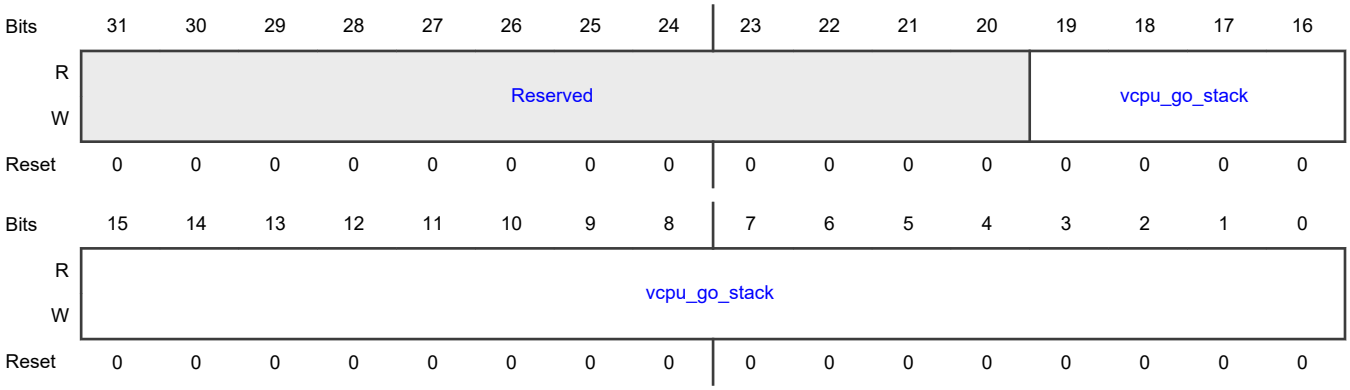
#### Offset

| 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<br>— | -<br>Reserved |
| 19-0<br>vcpu_go_stack | vcpu_go_stack |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | The bit field width is 20 bits, and it points to a byte (8-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. |

### 13.2.54 AUX DMA program memory address register (AD_PRAM_ADDR)

#### Offset

| Register | Offset |
|---|---|
| AD_PRAM_ADDR | 188h |

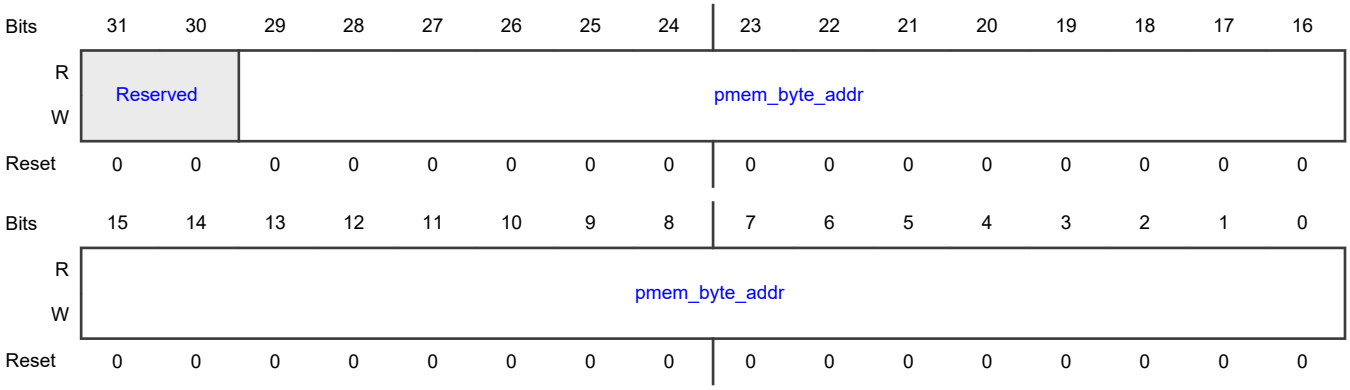
#### Function

This register configures the byte address of the first program memory address to be written by the AUX DMA. It can be written by either the VCPU or a host.

The value written must be aligned to 64 bytes, so the least significant 6 bits must be written to 0. Otherwise, when the control register is written, the command will be ignored and the cfg_error bit in AD_STATUS will be set.

This is a slow-read register.

#### Diagram




#### Fields

| Field | Function |
|---|---|
| 31-30<br>— | -<br>Reserved |
| 29-0<br>pmem_byte_addr | program memory byte address<br>Configures the byte address of the first program memory location to be written by the AUX DMA. |

## 13.2.55 AUX DMA AXI address register (AD_AXI_ADDR)

### Offset

| Register | Offset |
|---|---|
| AD_AXI_ADDR | 18Ch |

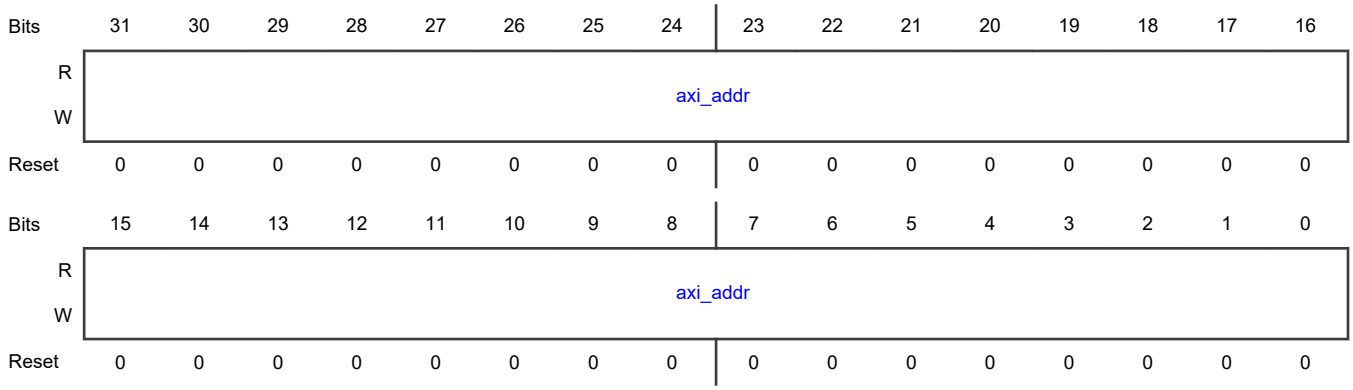
### Function

This register configures the AXI address to be read first by the AUX DMA. It can be written by either the VCPU or a host.

The value written must be aligned to 64 bytes, so the least significant 6 bits must be written to 0. Otherwise, when the control register is written, the command will be ignored and the cfg_error bit in AD_STATUS will be set.

This is a slow-read register.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-0<br>axi_addr | AXI address<br>Controls the initial AXI address to be read by the AUX DMA. |

## 13.2.56 AUX DMA byte count register (AD_BYTE_COUNT)

### Offset

| Register | Offset |
|---|---|
| AD_BYTE_COUNT | 190h |

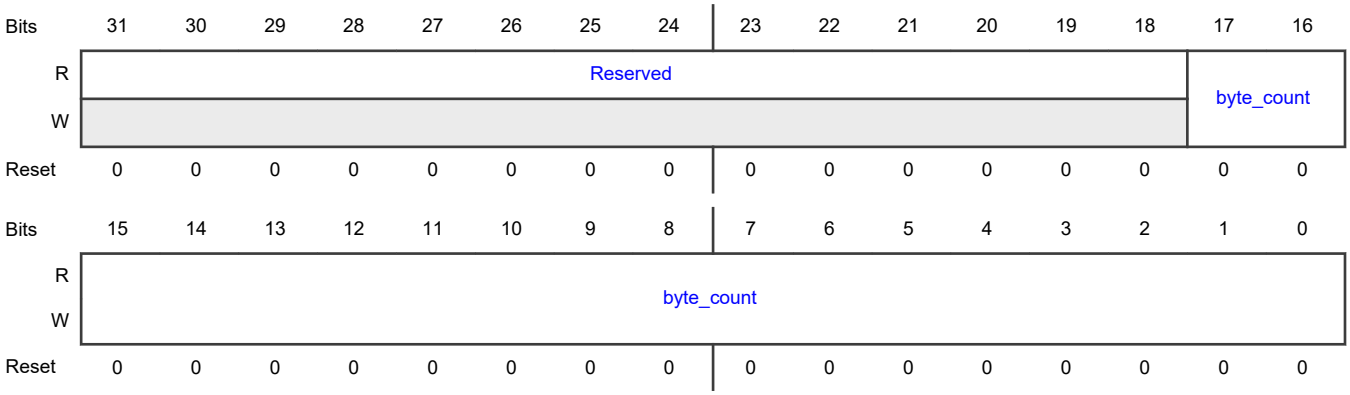
### Function

This register configures the number of bytes the AUX DMA will write to program memory. It can be written by either the VCPU or a host.

The value written must be an integer multiple of 64 bytes, so the least significant 6 bits must be written to 0. Otherwise, when the control register is written, the command will be ignored and the cfg_error bit in AD_STATUS will be set.

This is a slow-read register.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-18<br>— | -<br>Reserved |
| 17-0<br>byte_count | byte count to be written by the AUX DMA.<br>Controls the number of bytes to be written by the AUX DMA. |

## 13.2.57 AUX DMA control register (AD_CTRL)

### Offset

| Register | Offset |
|---|---|
| AD_CTRL | 194h |

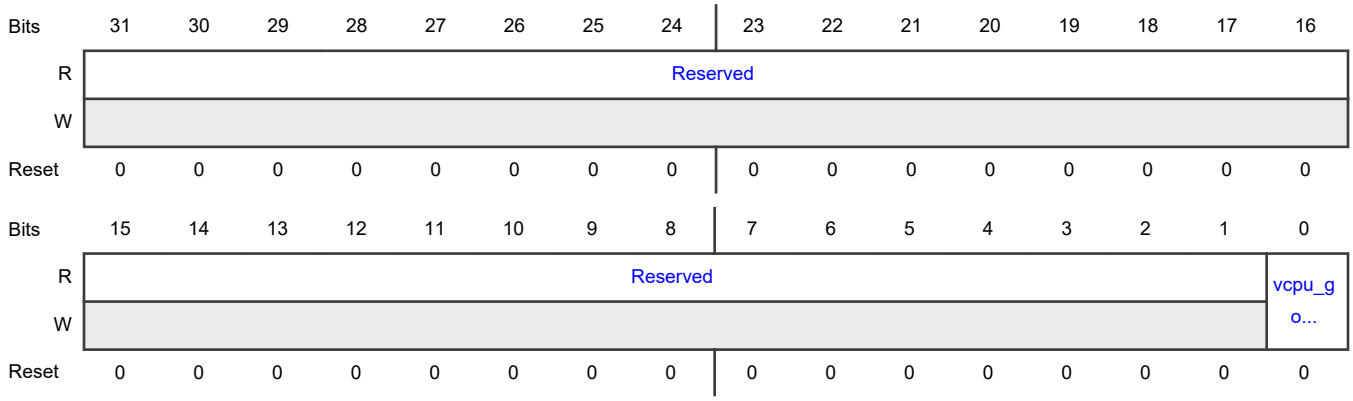
### Function

Writing to this register initiates an AUX DMA command. It can be written by either the VCPU or a host.

If the byte count or either address is incorrectly aligned, the AUX DMA will not execute the command, and the cfg_error bit in AD_STATUS will be set 2 cycles after the write to the control register.

This is a slow-read register.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-1<br>— | -<br>Reserved |
| 0<br>vcpu_go_en | VCPU go enable<br>- When set, causes the VCPU to go upon completion of the AUX DMA command. |

## 13.2.58 VCPU Mode 0 (VCPU_MODE0)

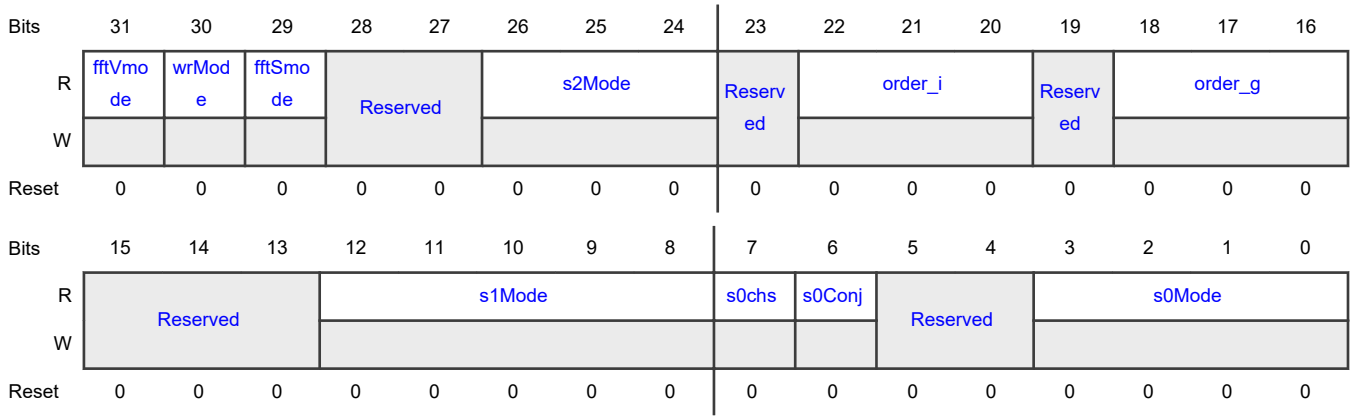
### Offset

| 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<br>fftVmode | fftVmode<br>FFT mode control bit for vector writeback |
| 30<br>wrMode | wrMode<br>Mode control bit for vector writeback |
| 29<br>fftSmode | fftSmode<br>FFT mode control bit for source mux |
| 28-27<br>— | -<br>Reserved |
| 26-24<br>s2Mode | s2Mode<br>Mode control for source mux 2 |
| 23<br>— | -<br>Reserved |
| 22-20<br>order_i | S1 order_i<br>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<br>— | -<br>Reserved |
| 18-16<br>order_g | S0 order_g<br>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). |
| 15-13<br>— | -<br>Reserved |
| 12-8<br>s1Mode | s1Mode<br>Mode control for source mux 1 |
| 7<br>s0chs | s0chs<br>Sign control for source mux 0 |
| 6<br>s0Conj | s0Conj<br>Conjugate control for source mux 0 |
| 5-4 | - |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| — | Reserved |
| 3-0<br>s0Mode | s0Mode<br>Mode control for source mux 0 |

## 13.2.59 VCPU Mode 1 (VCPU_MODE1)

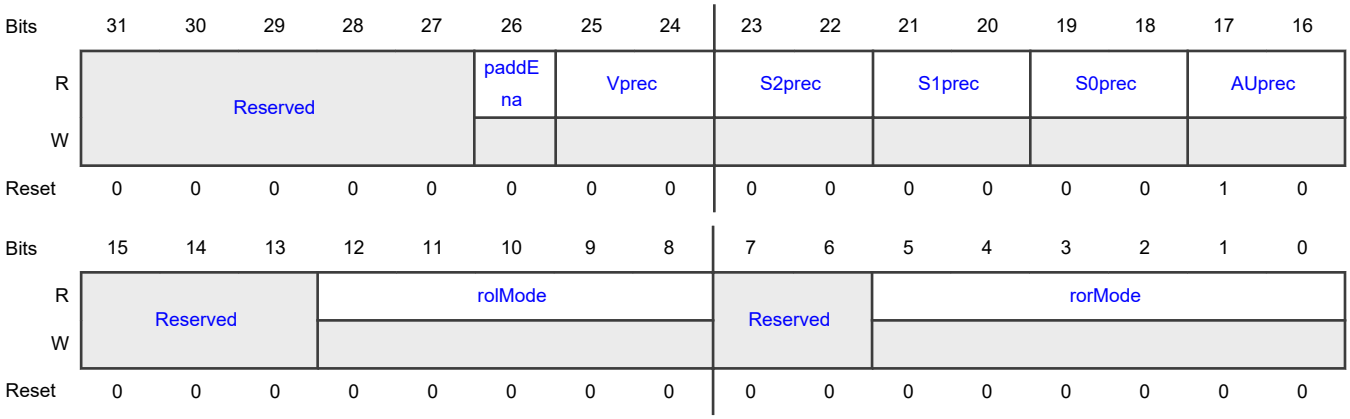
### Offset

| 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 |
|---|---|
| 31-27<br>— | -<br>Reserved |
| 26<br>paddEna | paddEna<br>In conjunction with AUprec enables paddF24 mode |
| 25-24<br>Vprec | Vprec<br>VRA write back precision mode |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 23-22<br>S2prec | S2prec<br>Source mux 2 precision mode |
| 21-20<br>S1prec | S1prec<br>Source mux 1 precision mode |
| 19-18<br>S0prec | S0prec<br>Source mux 0 precision mode |
| 17-16<br>AUprec | AUprec<br>Arithmetic unit precision mode |
| 15-13<br>— | -<br>Reserved |
| 12-8<br>rolMode | rolMode<br>Left rotation mode |
| 7-6<br>— | -<br>Reserved |
| 5-0<br>rorMode | rorMode<br>Right rotation mode |

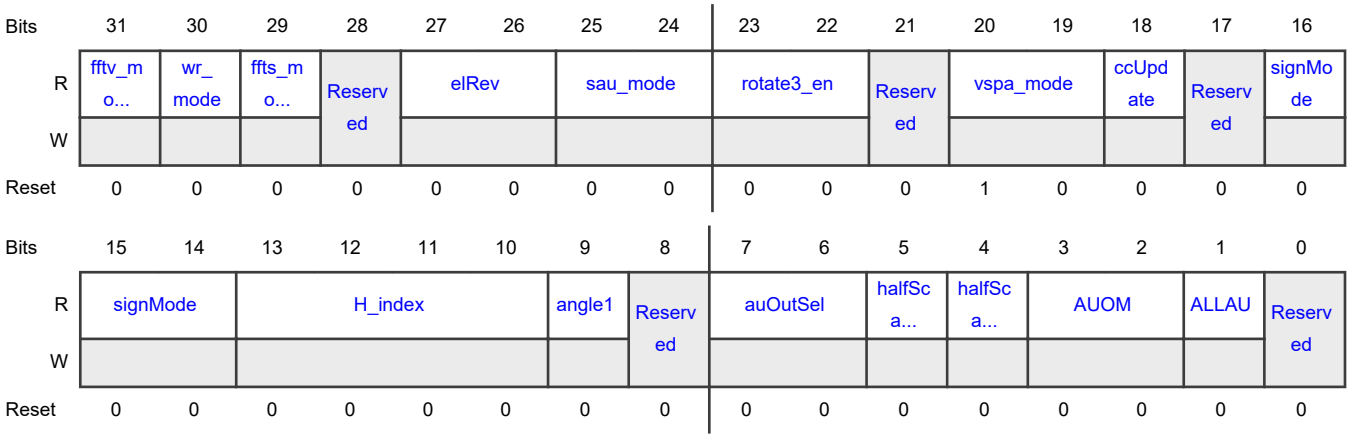
### 13.2.60 VCPU CREG 0 (VCPU_CREG0)

**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<br>fftv_mode | FFTV_mode<br>FFT Writeback mode select |
| 30<br>wr_mode | WR_mode<br>Writeback mode select |
| 29<br>ffts_mode | FFTS_mode<br>FFT Source mode select |
| 28<br>— | -<br>Reserved |
| 27-26<br>elRev | Element Reverse<br>Selects element size for VCPU instruction ld.el_rev and IPPU ld_mode er (Element Reverse) & brmr (Bit Reverse Multiple Registers) |
| 25-24<br>sau_mode | sau_mode<br>Selects alternate SAU instructions. |

| CREG0[27:26] | Load vector register with element reversal |
|---|---|
| 0b00 | Reverses elements of 16 bits |
| 0b01 | Reverses elements of 32 bits |
| 0b10 | Reverses elements of 64 bits |
| 0b11 | Undefined |

*Table continues on the next page...*

*Table continued from the previous page...*

<table>
<tr><th>Field</th><th>Function</th></tr>
<tr><td></td><td>
<table>
<tr><th>CREG0[25:24]</th><th>available SAU instructions</th></tr>
<tr><td>0b00</td><td>srt, rrt, rcp, nco, padd</td></tr>
<tr><td>0b01</td><td>Reserved</td></tr>
<tr><td>0b10</td><td>Reserved</td></tr>
<tr><td>0b11</td><td>Reserved</td></tr>
</table>
There must be 1 instruction between changing CREG0[25:24] and an SAU instruction using the new mode.
</td></tr>
<tr><td>23-22<br>rotate3_en</td><td>rotate3_en<br>Enable three-register rotate modes in place of two-register rotate ones.
<table>
<tr><th>CREG0[23:22]</th><th>Three register rotate right</th><th>Three register rotate left</th></tr>
<tr><td>0b00</td><td>Not available</td><td>Not available</td></tr>
<tr><td>0b01</td><td>Available</td><td>Not available</td></tr>
<tr><td>0b10</td><td>Not available</td><td>Available</td></tr>
<tr><td>0b11</td><td>Available</td><td>Available</td></tr>
</table>
</td></tr>
<tr><td>21<br>—</td><td>-<br>Reserved</td></tr>
<tr><td>20-19<br>vspa_mode</td><td>vspa_mode<br>VSPA Operating mode
<table>
<tr><th>CREG0[20:19]</th><th>VSPA Operating mode</th></tr>
<tr><td>0b00</td><td>VSPA2 mode, supports half word (16-bit) memory access resolution</td></tr>
<tr><td>0b01</td><td>Reserved, not supported</td></tr>
<tr><td>0b10</td><td>VPSA3 mode, supports byte (8-bit) memory access resolution</td></tr>
<tr><td>0b11</td><td>Reserved, not supported</td></tr>
</table>
</td></tr>
<tr><td>18<br>ccUpdate</td><td>ccUpdate<br>Condition Code update switch</td></tr>
<tr><td>17<br>—</td><td>-<br>Reserved</td></tr>
</table>

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 16-14<br>signMode | signMode<br>H register control |
| 13-10<br>H_index | H_index<br>H register sub-address. Read only. |
| 9<br>angle1 | angle<br>ANGLE_MODULO_WRAP |
| 8<br>— | -<br>Reserved |
| 7-6<br>auOutSel | auOutSel<br>SP VAU output lane switch |
| 5<br>halfScale1 | halfScale<br>S1prec control |
| 4<br>halfScale | halfScale<br>Vprec control |
| 3-2<br>AUOM | AUOM<br>VAU output mode control |
| 1<br>ALLAU | ALLAU<br>SP VAU output width switch |
| 0<br>— | -<br>Reserved |

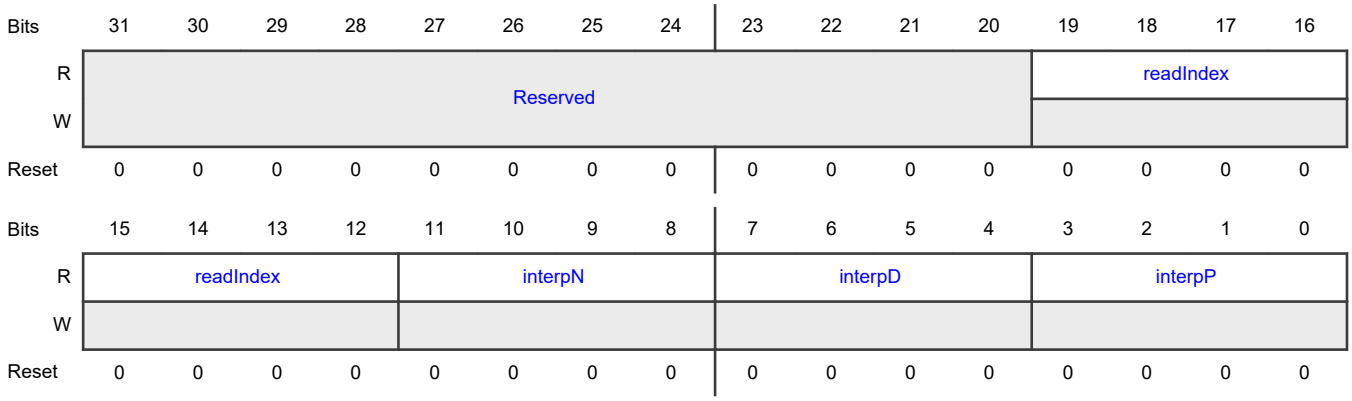
### 13.2.61 VCPU CREG 1 (VCPU_CREG1)

**Offset**

| Register | Offset |
|---|---|
| VCPU_CREG1 | 40Ch |

**Function**
This register may only be written by the VCPU, it is read only by the host.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-20<br>— | -<br>Reserved |
| 19-12<br>readIndex | readIndex<br>CReg read address |
| 11-8<br>interpN | interpN<br>Fractional interpolator numerator constant |
| 7-4<br>interpD | interpD<br>Fractional interpolator denominator constant. For more information, refer to Table 23 |
| 3-0<br>interpP | interpP<br>Fractional interpolator phase |

## 13.2.62 Store Unalign Vector Length (ST_UL_VEC_LEN)

### Offset

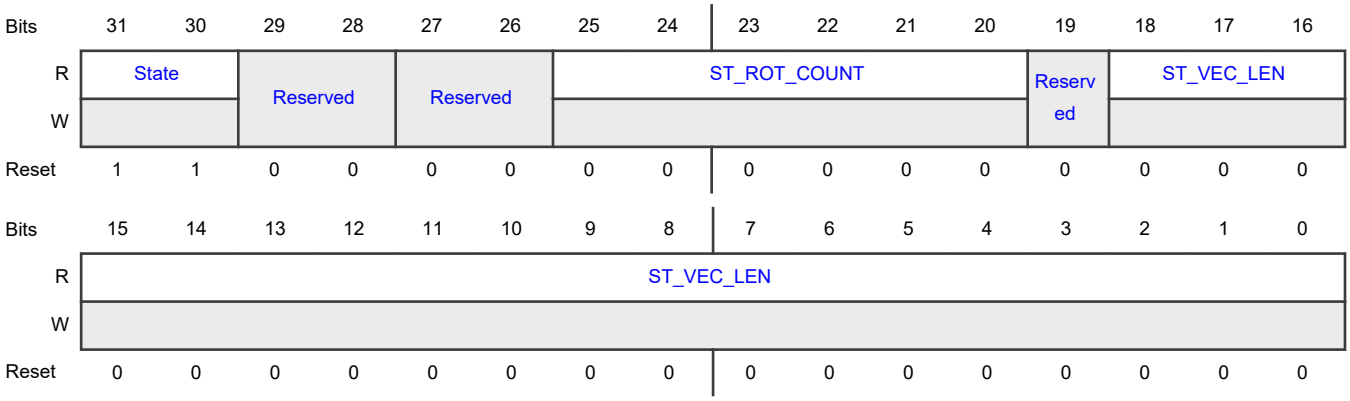
| Register | Offset |
|---|---|
| ST_UL_VEC_LEN | 410h |

### 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<br>State | State<br>Current state of mis-aligned store FSM<br>**NOTE**<br>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.<br>00b - active wrap (store crossing line boundary)<br>01b - invalid<br>10b - active no wrap (full line store)<br>11b - idle |
| 29-28<br>— | -<br>Reserved |
| 27-26<br>— | -<br>Reserved |
| 25-20<br>ST_ROT_COUNT | ST_ROT_COUNT<br>Rotation offset into dmem line. Maximum width of field is 8 bits, actual width is dependent on AU count. Actual width = log2(NAU*4)<br>**NOTE**<br>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<br>— | -<br>Reserved |
| 18-0 | ST_VEC_LEN |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 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.<br>This register may only be written by the VCPU, it is read only by the host. |

## 13.2.63 Vector Predicate Control (VP_CONTROL)

### Offset

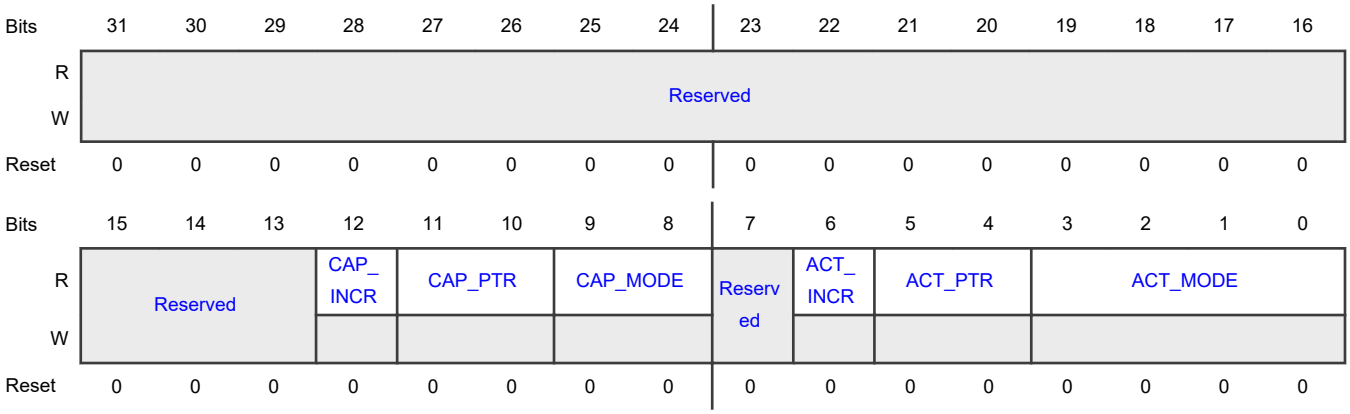
| Register | Offset |
|---|---|
| VP_CONTROL | 414h |

### Function

This register may only be written by the VCPU, it is read only by the host.

| Host/VCPU | Register Access |
|---|---|
| Host | RO |
| VCPU | RW |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-13<br>— | -<br>Reserved |
| 12 | CAP_INCR |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| CAP_INCR | VP capture pointer increment control<br>These are circular pointers, which are intended to wrap when incremented or decremented.<br>0b - no increment<br>1b - increment |
| 11-10<br>CAP_PTR | CAP_PTR<br>Value of VP capture pointer |
| 9-8<br>CAP_MODE | CAP_MODE<br>VP capture mode<br>00b - no capture<br>01b - capture VN (VAU vector is all negative)<br>10b - capture VZ(zero bits)<br>11b - capture VN and VZ (into either VP[3:2] or VP[1:0], destination controlled by the capture pointer's MSB) |
| 7<br>— | -<br>Reserved |
| 6<br>ACT_INCR | ACT_INCR<br>VP action pointer increment control<br>These are circular pointers, which are intended to wrap when incremented or decremented.<br>0b - no increment<br>1b - increment |
| 5-4<br>ACT_PTR | ACT_PTR<br>Value of VP action pointer |
| 3-0<br>ACT_MODE | ACT_MODE<br>VP alternate action control<br>**NOTE**<br>These alternate operations are what comes out of the computational unit of the AU, not necessarily out of the AU's accumulator - the output of each AU is the accumulator. If an instruction does not use accumulation, the accumulator output will be replaced with the values shown in the table above. If the instruction uses accumulation, the S2 operand into the AU is disabled and replaced with the previous state of the accumulator.<br>0000b - AUout <- 0 // force 0<br>0001b - AUout <- S0*S1 + 0 // multiply only<br>0010b - AUout <- S0*1 + S2 // add only |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 0011b - AUout <- S0*S1 + S2 // normal instruction operation (ignore predicate)<br>0100b - AUout <- S0*1 + 0 // forward S0<br>0101b - Reserved<br>0110b - AUout <- S0*0 + S2 // forward S2(not available for rmac and rmac.sau instructions)<br>0111b - AUout <- AUout // hold state(only available for rmac and rmac.sau instructions)<br>1000b - AUout <- +1 // force +1<br>1001b - Reserved<br>1010b - S0*1 - S2 // subtract only<br>1011b - Reserved<br>1100b - AUout <- S0*-1 + 0 // forward -S0<br>1101b - Reserved<br>1110b - AUout <- S0*0 + S2 // forward -S2(not available for rmac and rmac.sau instructions)<br>1111b - AUout <- -1 // force -1 |

## 13.2.64 Extended Condition Control (XC_CONTROL)

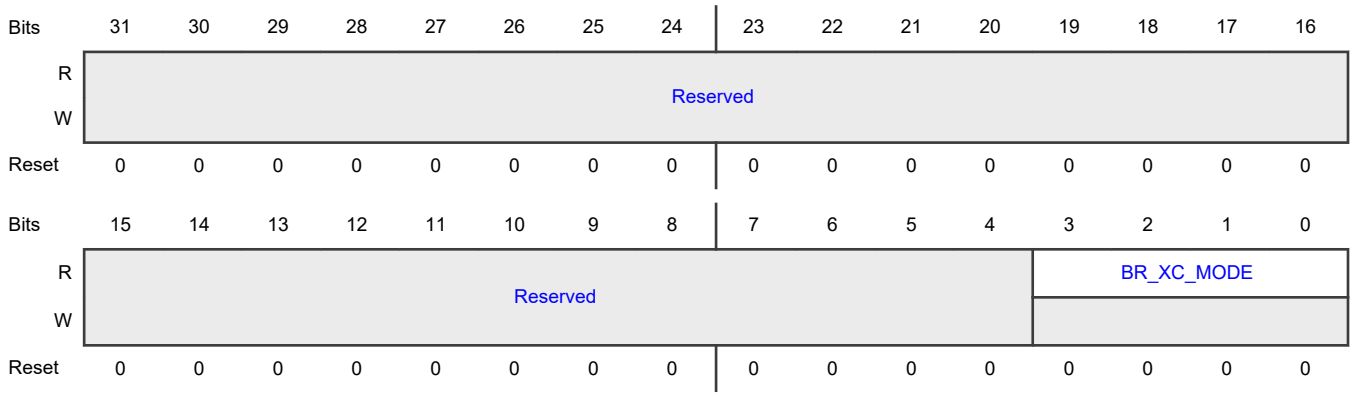
### Offset

| Register | Offset |
|---|---|
| XC_CONTROL | 418h |

### Function

This register may only be written by the VCPU, it is read only by the host.

### Diagram



**Fields**

| Field | Function |
|---|---|
| 31-4<br>— | -<br>Reserved |
| 3-0<br>BR_XC_MODE | BR_XC_MODE<br>Select special branch condition<br>0000b - nv (NeVer)<br>0001b - az (All Zero)<br>0010b - Reserved<br>0011b - naz (Not All Zero)<br>0100b - ap (All Positive)<br>0101b - an (All Negative)<br>0110b - nap (Not All Positive)<br>0111b - nan (Not All Negative)<br>1000b - Reserved<br>1001b - Reserved<br>1010b - Reserved<br>1011b - Reserved<br>1100b - Reserved<br>1101b - Reserved<br>1110b - Reserved<br>1111b - Reserved |

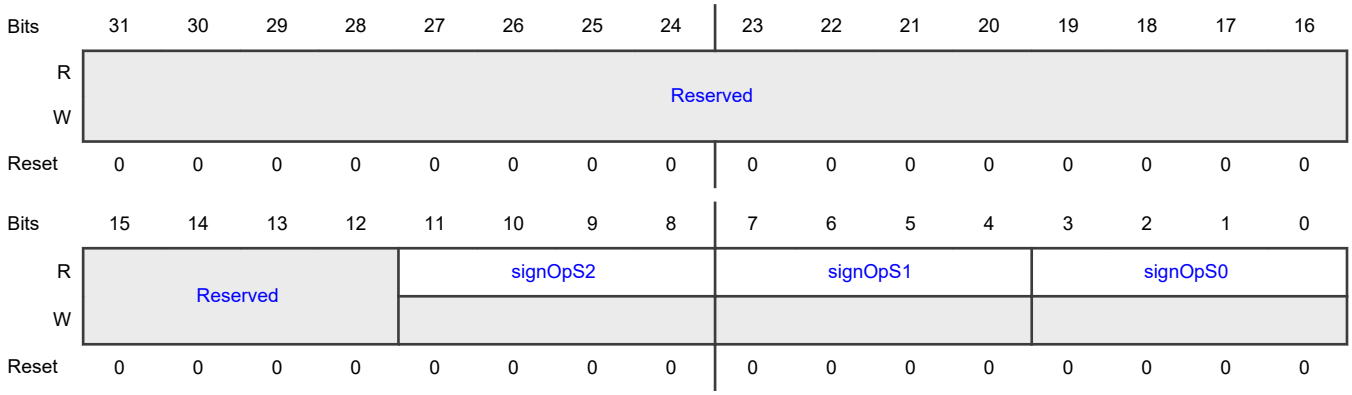
## 13.2.65 VCPU CREG 2 (VCPU_CREG2)

**Offset**

| Register | Offset |
|---|---|
| VCPU_CREG2 | 41Ch |

**Function**
This register may only be written by the VCPU, it is read only by the host.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-12<br>— | -<br>Reserved |
| 11-8<br>signOpS2 | signOpS2<br>S2 sign operation<br>signOpS2[3:2]<br>• 0b00 No sign operation<br>• 0b01 Sign operation on Real element<br>• 0b10 Sign operation on Imag element<br>• 0b11 Sign operation on Both element<br>signOpS2[1:0]<br>• 0b00 Absolute value<br>• 0b01 Negative value<br>• 0b10 Change sign<br>• 0b11 -1 if element < 0, else +1 |
| 7-4<br>signOpS1 | signOpS1<br>S1 sign operation<br>signOpS1[3:2]<br>• 0b00 No sign operation<br>• 0b01 Sign operation on Real element<br>• 0b10 Sign operation on Imag element<br>• 0b11 Sign operation on Both element<br>signOpS1[1:0] |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | • 0b00 Absolute value<br>• 0b01 Negative value<br>• 0b10 Change sign<br>• 0b11 -1 if element < 0, else +1 |
| 3-0<br>signOpS0 | signOpS0<br>S0 sign operation<br>signOpS0[3:2]<br>• 0b00 No sign operation<br>• 0b01 Sign operation on Real element<br>• 0b10 Sign operation on Imag element<br>• 0b11 Sign operation on Both element<br>signOpS0[1:0]<br>• 0b00 Absolute value<br>• 0b01 Negative value<br>• 0b10 Change sign<br>• 0b11 -1 if element < 0, else +1 |

### 13.2.66 RAG set rS0 & rS1 registers (RAG_SET_S0S1_0 - RAG_SET_S0S1_7)

**Offset**

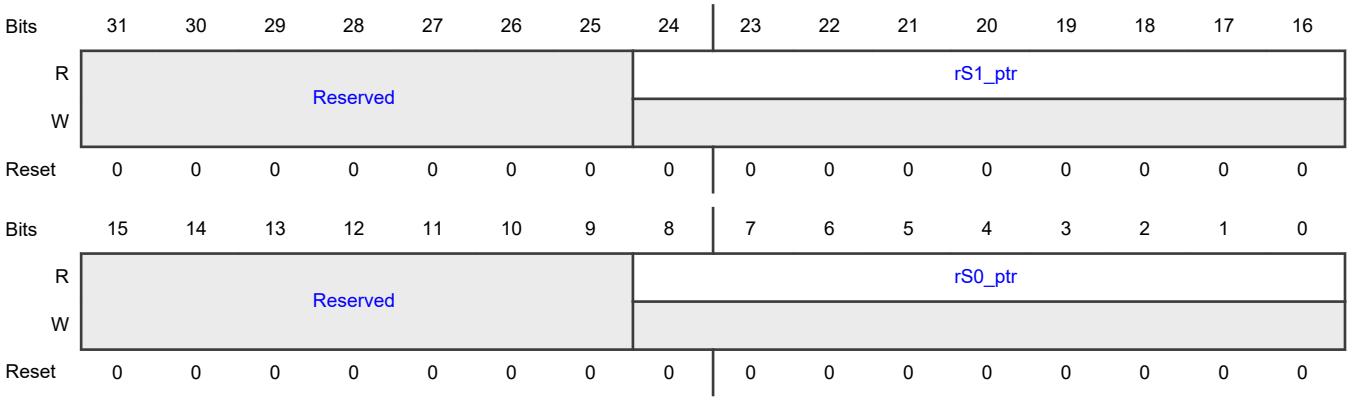
For i = 0 to 7:

| Register | Offset |
|---|---|
| RAG_SET_S0S1_(((i * 1) + 0)) | 480h + (i × 8h) |

**Function**

The RAG set registers are used by the set.VRAptrIP instructuction to load the rS0 and rS1 pointers.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-25<br>— | -<br>Reserved |
| 24-16<br>rS1_ptr | rS1_ptr<br>This value is loaded into the rS1 pointer when bit 1 of Iu5 is set in the set.VRAptrIP *Iu5*, Iu4; instruction. |
| 15-9<br>— | -<br>Reserved |
| 8-0<br>rS0_ptr | rS0_ptr<br>This value is loaded into the rS0 pointer when bit 0 of Iu5 is set in the set.VRAptrIP *Iu5*, Iu4; instruction. |

## 13.2.67 RAG set rS2, rV & rSt registers (RAG_SET_S2VSt_0 - RAG_SET_S2VSt_7)

### Offset

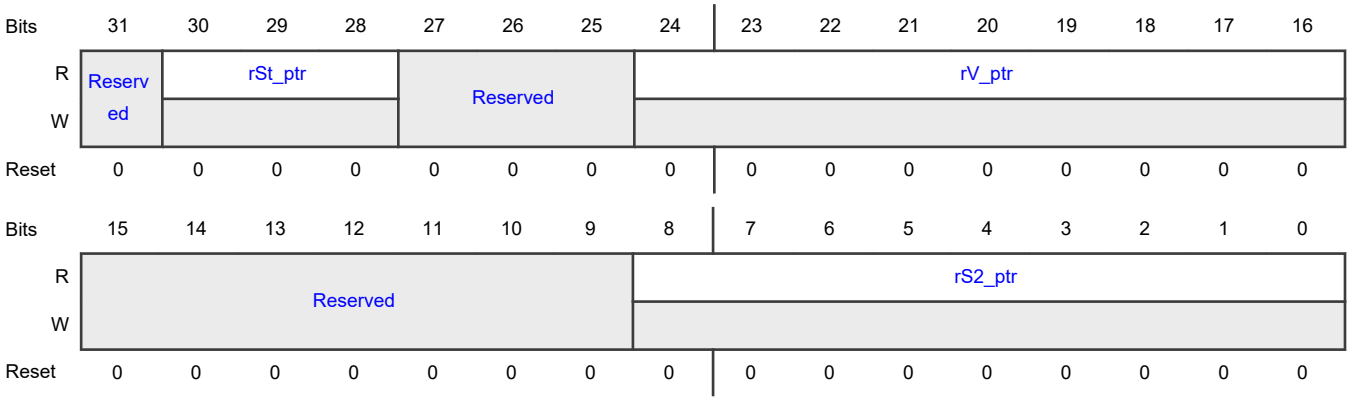
For i = 0 to 7:

| Register | Offset |
|---|---|
| RAG_SET_S2VSt_(((i * 1) + 0)) | 484h + (i × 8h) |

### Function

The RAG set registers are used by the set.VRAptrIP instructuction to load the rS2, rV and rSt pointers.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31<br>— | -<br>Reserved |
| 30-28<br>rSt_ptr | rSt_ptr<br>This value is loaded into the rSt pointer when bit 4 of Iu5 is set in the set.VRAptrIP *Iu5*, Iu4; instruction. |
| 27-25<br>— | -<br>Reserved |
| 24-16<br>rV_ptr | rV_ptr<br>This value is loaded into the rV pointer when bit 3 of Iu5 is set in the set.VRAptrIP *Iu5*, Iu4; instruction. |
| 15-9<br>— | -<br>Reserved |
| 8-0<br>rS2_ptr | rS2_ptr<br>This value is loaded into the rS2 pointer when bit 2 of Iu5 is set in the set.VRAptrIP *Iu5*, Iu4; instruction. |

## 13.2.68 General Purpose Input registers [23 registers] (GP_IN0 - GP_IN22)

### Offset

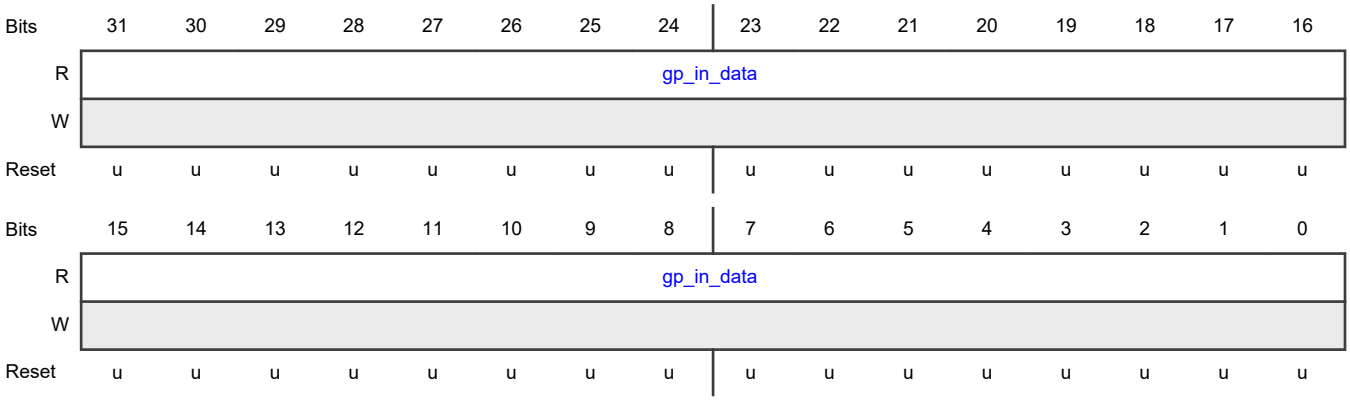
For a = 0 to 22:

| Register | Offset |
|---|---|
| GP_INa | 500h + (a × 4h) |

### Function

- GP_IN registers are read only - writes have no meaning or effect.
- 23 General Purpose Input registers are available.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-0<br>gp_in_data | gp_in_data<br>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. |

## 13.2.69 General Purpose Output registers [21 registers] (GP_OUT0 - GP_OUT20)

### Offset

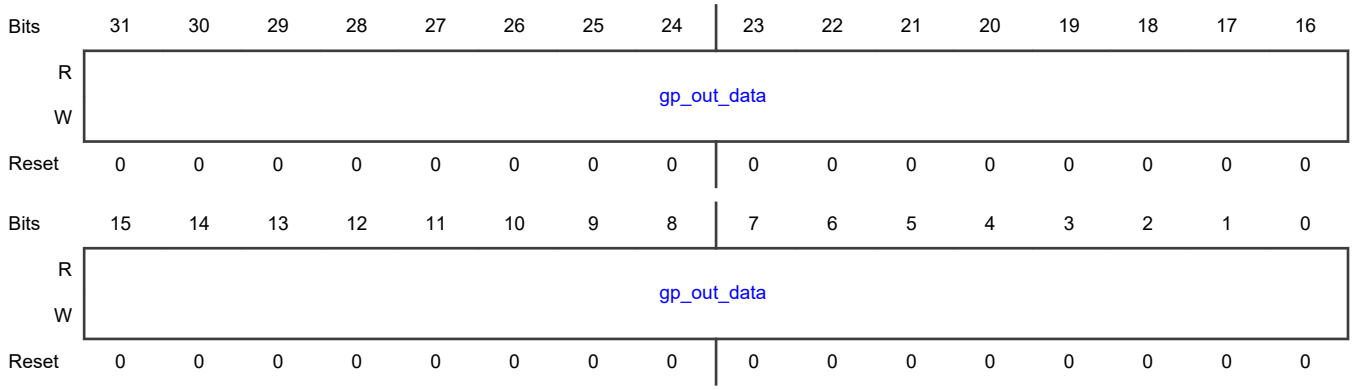
For a = 0 to 20:

| 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.
- 21 General Purpose Output registers are available.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-0<br>gp_out_data | gp_out_data<br>gp_out_data<br>• Writes control the state of the associated output port.<br>• Reads return the state of the associated output port.<br>00000000000000000000000000000000b - output port low.<br>00000000000000000000000000000001b - output port high. |

## 13.2.70 VCPU to DQM Trace Small Outbox register (DQM_SMALL)

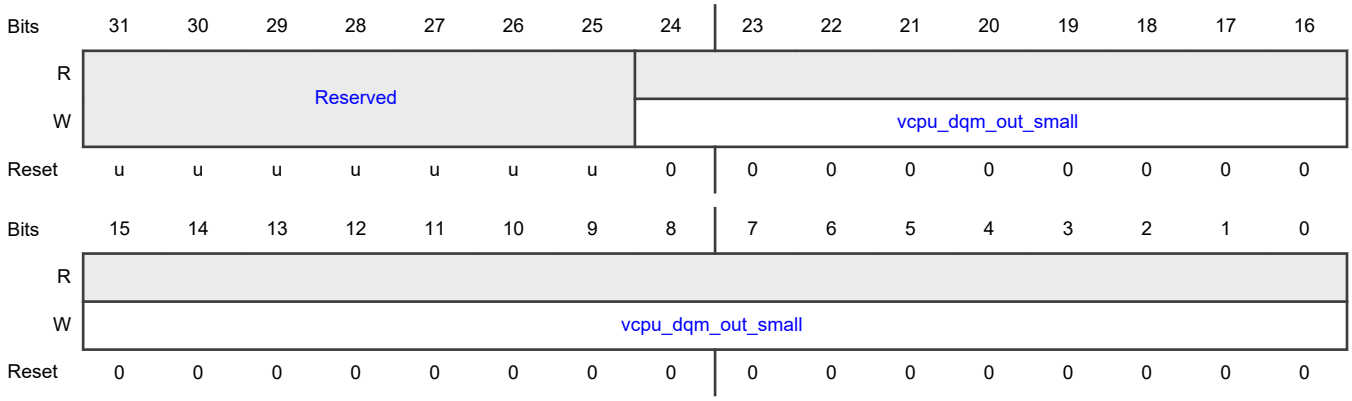
### Offset

| 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<br>— | -<br>Reserved |
| 24-0<br>vcpu_dqm_out_small | vcpu_dqm_out_small<br>DQM output data (small)<br>• Writes cause the value written to be transmitted as a DQM message.<br>• This register is write only - reads always return 0's.<br>Settings: Data value to be transmitted as a DQM via the VSPA trace port. |

## 13.2.71 VCPU to DQM Trace Large MSB Outbox register (DQM_LARGE_MSB)

### Offset

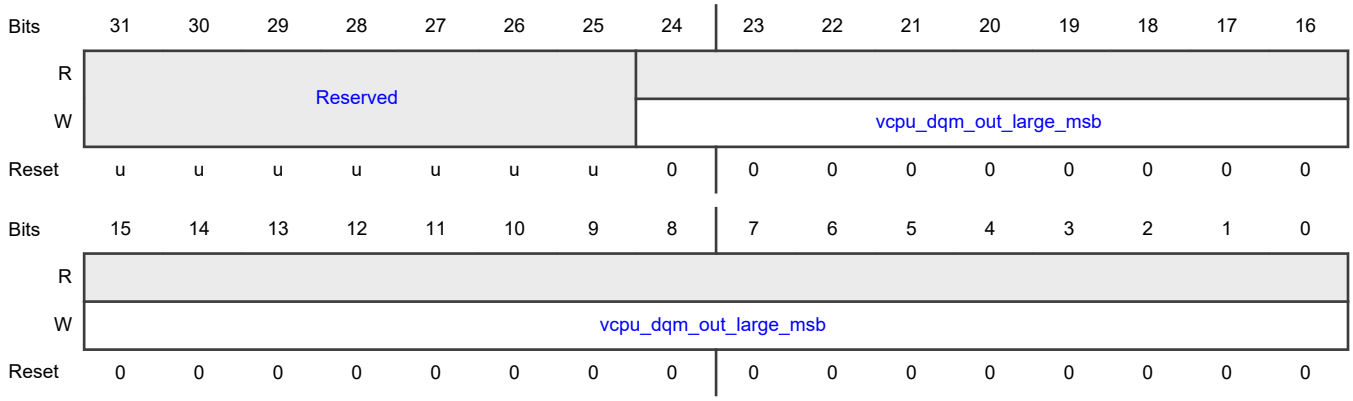
| Register | Offset |
|---|---|
| DQM_LARGE_MSB | 604h |

### Function

This register is part of the interface used by VCPU (or a host) to generate a "large" (57-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.

Note that writing this register is not required to initiate the DQM message. The large DQM message will be sent following a write to the DQM_LARGE_LSB register, and the message payload will contain 25 MSBs from this register plus all 32 bits from the DQM_LARGE_LSB register.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-25<br>— | -<br>Reserved |
| 24-0<br>vcpu_dqm_out_large_msb | vcpu_dqm_out_large_msb<br>DQM output data (large, MSB)<br>• Writes update the 25 MSBs of a pending 57-bit DQM.<br>• The message will be sent in its entirety only after a write to the DQM_LARGE_LSB register.<br>• Note that this register does not have to be updated for each 57-bit DQM - if it is not updated, the last value written will be reused when the DQM is sent.<br>• This register is write only - reads always return 0's.<br>Settings: MSB 25-bit data value to be transmitted as part of a 57-bit DQM via the VSPA trace port. |

## 13.2.72 VCPU to DQM Trace Large LSB Outbox register (DQM_LARGE_LSB)

### Offset

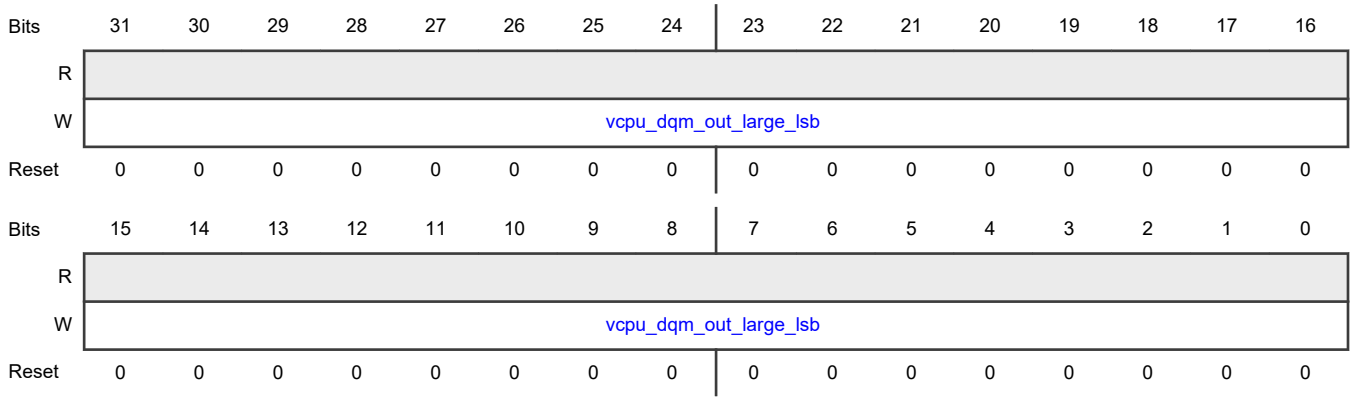
| Register | Offset |
|---|---|
| DQM_LARGE_LSB | 608h |

### Function

This register is part of the interface used by VCPU (or a host) to generate a "large" (57-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.

Note that writing to this register alone will initiate the large DQM message. The DQM message payload will contain 25 MSBs from the DQM_LARGE_MSB register plus all 32 bits from this register.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-0<br>vcpu_dqm_out_large_lsb | vcpu_dqm_out_large_lsb<br>DQM output data (large, LSB)<br>• Writes cause the immediate transmission of a 57-bit DQM, whose payload is comprised of 25 bits from the DQM_LARGE_MSB register and the 32 bits written to this register.<br>• This register is write only - reads always return 0's.<br>Settings: LSB 32-bit data value to be transmitted as part of a 57-bit DQM via the VSPA trace port. |

## 13.2.73 VCPU to Debugger 32-bit Outbox register (VCPU_DBG_OUT_32)

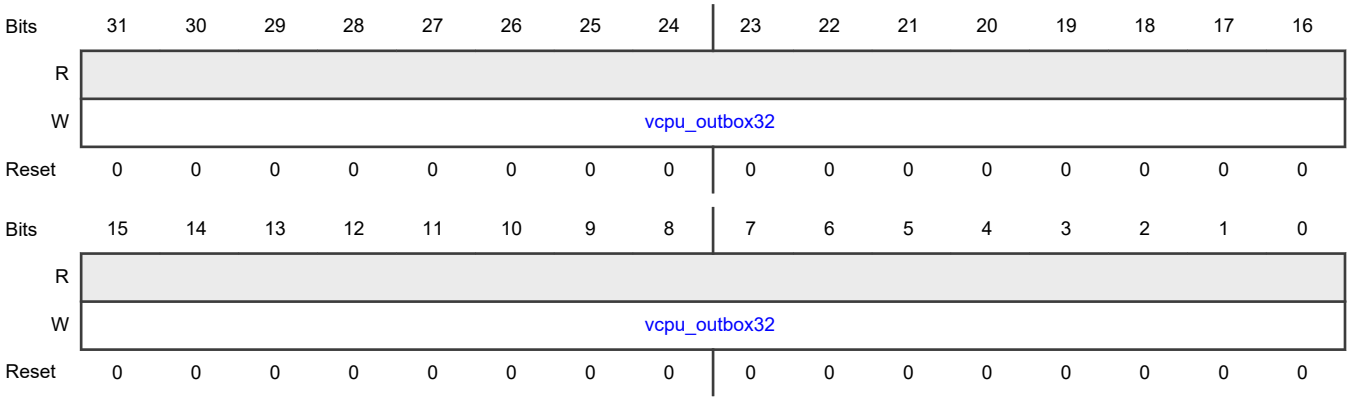
### Offset

| 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<br>vcpu_outbox32 | vcpu_outbox32<br>VCPU to Debugger 32-bit outbox<br>• Writes cause the value written to be sent to the debugger.<br>• This register is write only - reads always return 0's.<br>Settings: Data value to be sent to the debugger inbox. |

## 13.2.74 VCPU to Debugger 64-bit MSB Outbox register (VCPU_DBG_OUT_64_MSB)

### Offset

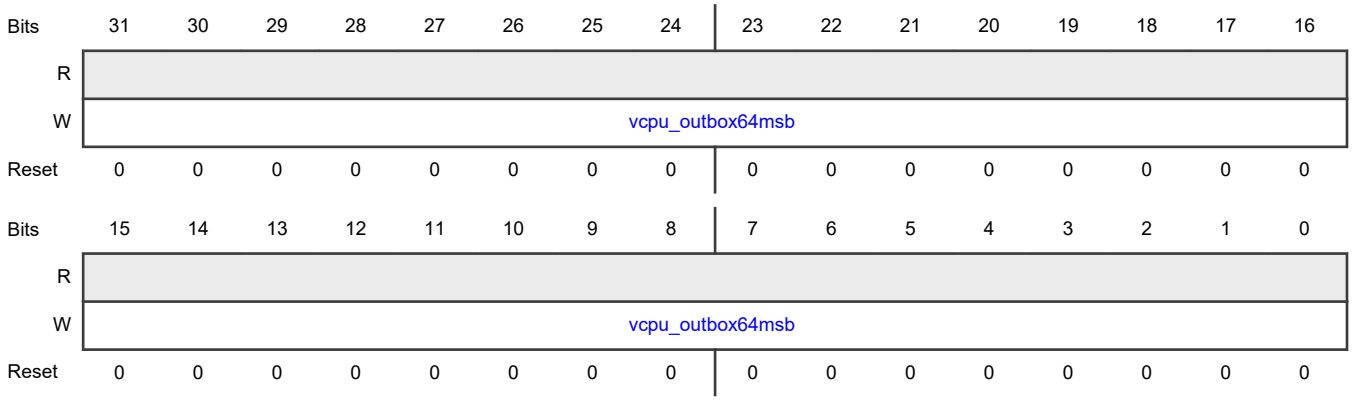
| Register | Offset |
|---|---|
| VCPU_DBG_OUT_64_MSB | 624h |

### 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<br>vcpu_outbox64msb | vcpu_outbox64msb<br>VCPU to Debugger 64-bit output data (MSB)<br>• Writes update the 32 MSBs of a pending 64-bit message that will be sent to the debugger's 64-bit inbox.<br>• The mail message will be sent in its entirety only after a write to the VCPU_DBG_OUT_64_LSB register.<br>• 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.<br>• This register is write only - reads always return 0's.<br>Settings: MSB 32-bit data value to be sent to the debugger's 64-bit inbox. |

## 13.2.75 VCPU to Debugger 64-bit LSB Outbox register (VCPU_DBG_OUT_64_LSB)

### Offset

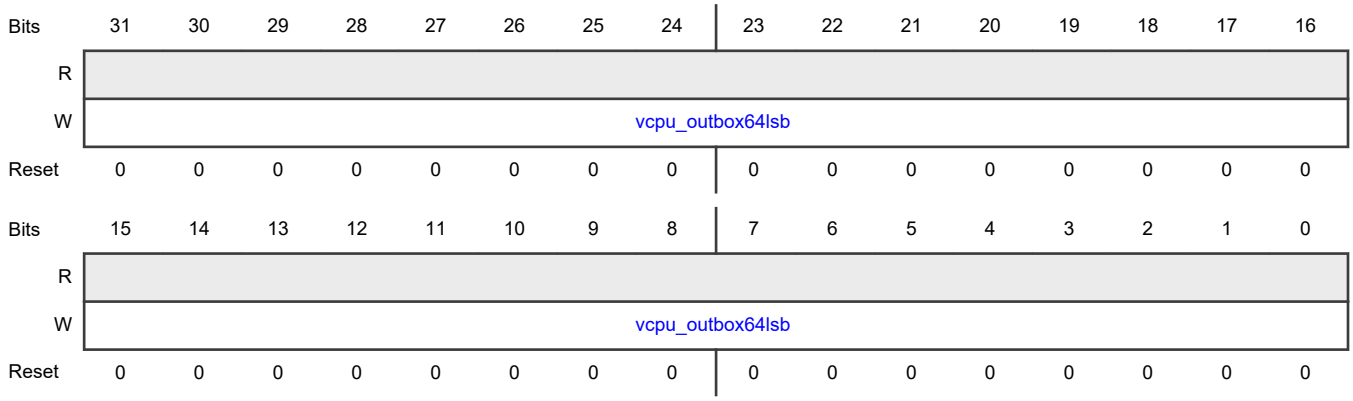
| Register | Offset |
|---|---|
| VCPU_DBG_OUT_64_LSB | 628h |

### 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<br>vcpu_outbox64lsb | vcpu_outbox64lsb<br>VCPU to Debugger 64-bit output data (LSB)<br>• 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.<br>• This register is write only - reads always return 0's.<br>Settings: LSB 32-bit data value to be sent to the debugger's 64-bit inbox. |

## 13.2.76 Debugger to VCPU 32-bit Inbox register (VCPU_DBG_IN_32)

### Offset

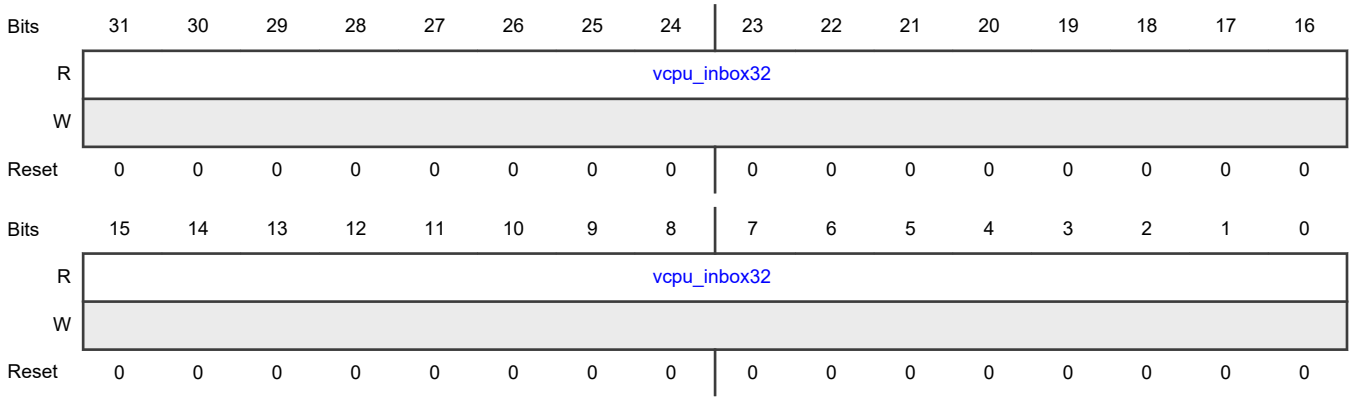
| Register | Offset |
|---|---|
| VCPU_DBG_IN_32 | 62Ch |

### 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<br>vcpu_inbox32 | vcpu_inbox32<br>Debugger to VCPU 32-bit inbox<br>• Reads the 32-bit mail message sent by the debugger.<br>• This register is read only - writes have no effect.<br>Settings: Reads 32-bit data value sent by the debugger outbox. |

## 13.2.77 Debugger to VCPU 64-bit MSB Inbox register (VCPU_DBG_IN_64_MSB)

### Offset

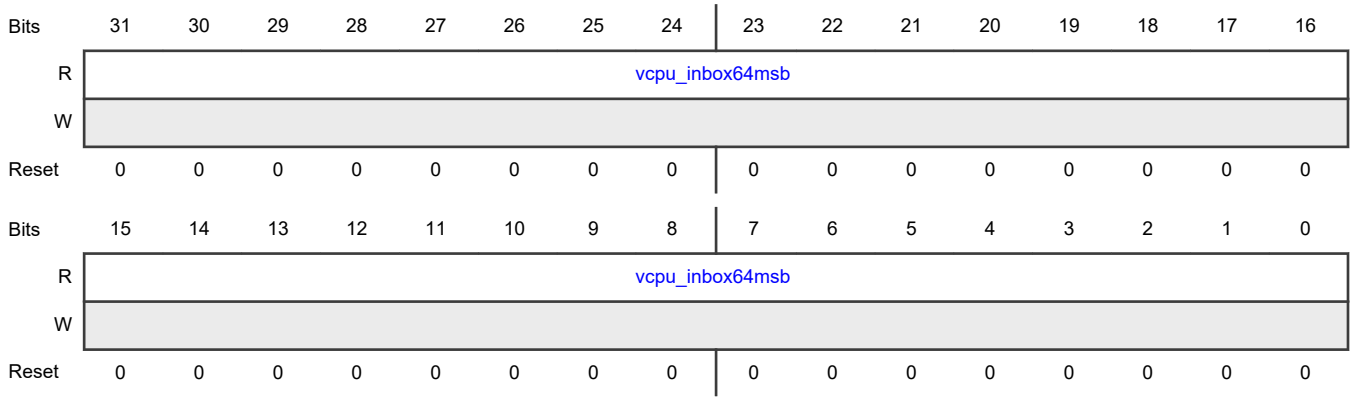
| Register | Offset |
|---|---|
| VCPU_DBG_IN_64_MSB | 630h |

### 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<br>vcpu_inbox64msb | vcpu_inbox64msb<br>Debugger to VCPU 64-bit input data (MSB)<br>• Reads the 32 MSBs of the 64-bit mail message sent by the debugger.<br>• This register is read only - writes have no effect.<br>Settings:<br>Reads return 32 MSBs of the 64-bit data value sent via the debugger's 64-bit outbox. |

## 13.2.78 Debugger to VCPU 64-bit LSB Inbox register (VCPU_DBG_IN_64_LSB)

### Offset

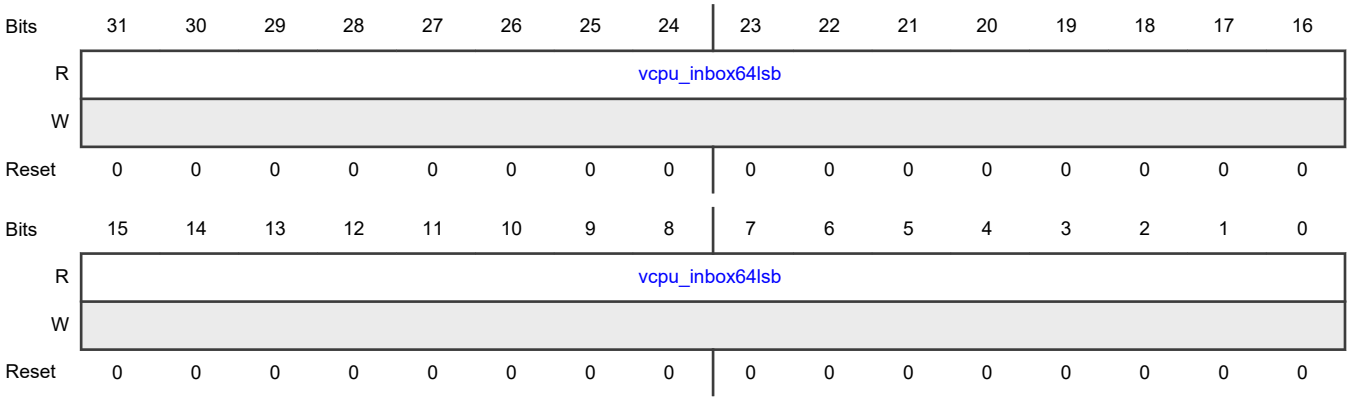
| Register | Offset |
|---|---|
| VCPU_DBG_IN_64_LSB | 634h |

### 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<br>vcpu_inbox64lsb | vcpu_inbox64lsb<br>Debugger to VCPU 64-bit input data (LSB)<br>• Reads the 32 LSBs of the 64-bit mail message sent by the debugger.<br>• This register is read only - writes have no effect.<br>Settings: Reads return 32 LSBs of the 64-bit data value sent via the debugger's 64-bit outbox. |

## 13.2.79 VCPU to Debugger Mailbox Status register (VCPU_DBG_MBOX_STATUS)

### Offset

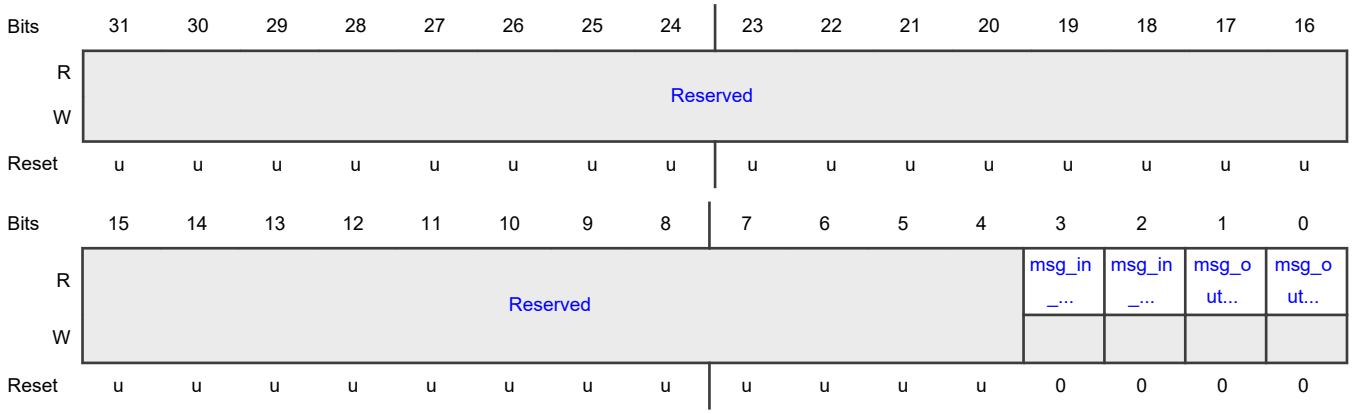
| Register | Offset |
|---|---|
| VCPU_DBG_MBOX_STATUS | 638h |

### 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<br>— | -<br>Reserved |
| 3<br>msg_in_valid_64bit | msg_in_valid_64bit<br>64-bit message inbox valid<br>• 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).<br>0b - Data in the 64-bit inbox registers is NOT valid.<br>1b - Data in the 64-bit inbox registers is valid. |
| 2<br>msg_in_valid_32bit | msg_in_valid_32bit<br>32-bit message inbox valid<br>• Shows the validity/invalidity of the data in the 32-bit message inbox register (Debugger to VCPU 32-bit Inbox register).<br>0b - Data in the 32-bit inbox register is NOT valid.<br>1b - Data in the 32-bit inbox register is valid. |
| 1<br>msg_out_valid_64bit | msg_out_valid_64bit<br>64-bit message outbox valid<br>• 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).<br>0b - No unread data is in the 64-bit outbox registers.<br>1b - Data in the 64-bit outbox registers has not yet been read by the debugger. |
| 0 | msg_out_valid_32bit<br>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).<br>0b - No unread data is in the 32-bit outbox register.<br>1b - Data in the 32-bit outbox register has not yet been read by the debugger. |

### 13.2.80 VCPU to host outbox message n MSB register (VCPU_OUT_0_MSB - VCPU_OUT_1_MSB)

#### Offset

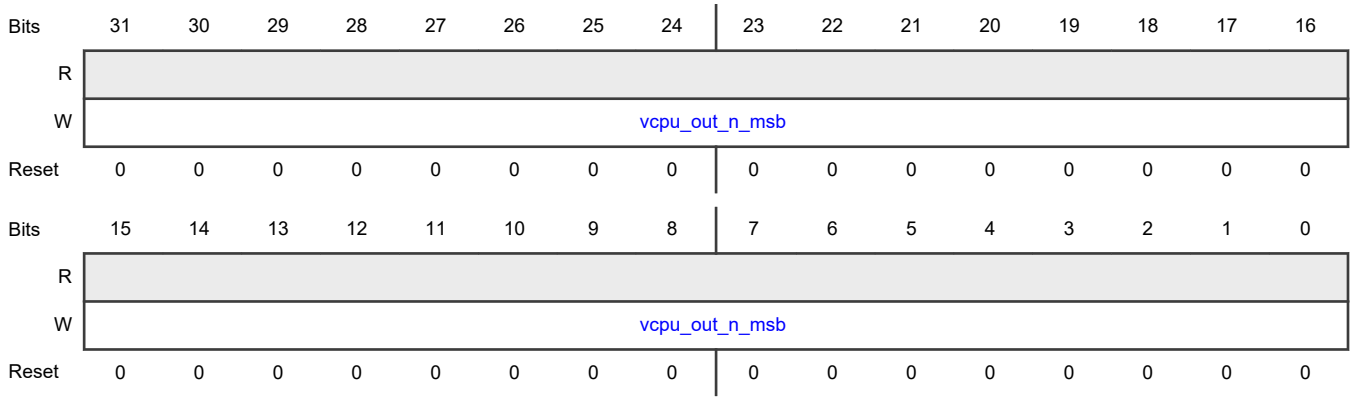
| Register | Offset |
|---|---|
| VCPU_OUT_0_MSB | 640h |
| VCPU_OUT_1_MSB | 648h |

#### 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<br>vcpu_out_n_msb | vcpu_out_n_msb<br>VCPU output message *n* data (MSB).<br>• 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.<br>• 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).<br>Settings: MSB 32-bit data value will be sent to the host's 64-bit message n inbox . |

### 13.2.81 VCPU to host outbox message n LSB register (VCPU_OUT_0_LSB - VCPU_OUT_1_LSB)

#### Offset

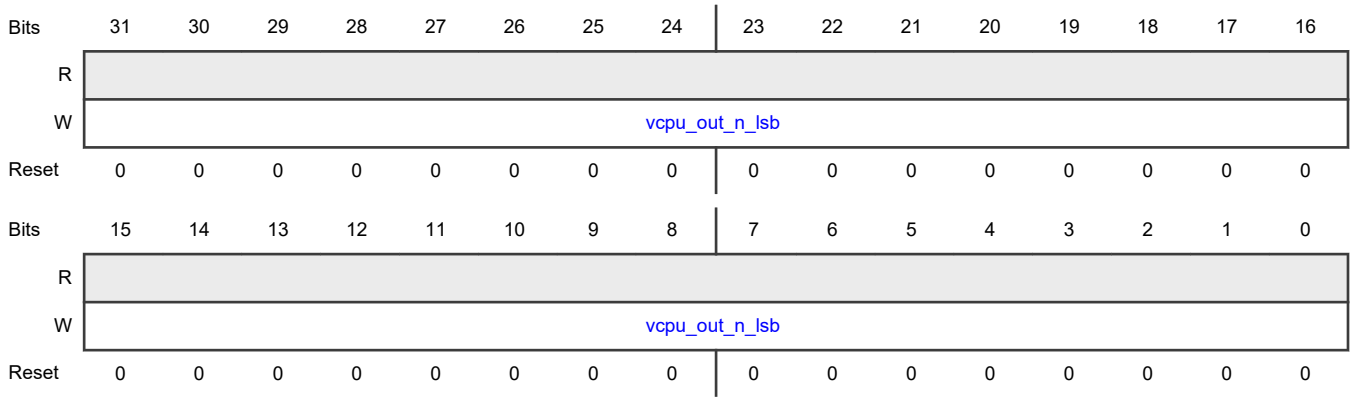
| Register | Offset |
|---|---|
| VCPU_OUT_0_LSB | 644h |
| VCPU_OUT_1_LSB | 64Ch |

#### 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<br>vcpu_out_n_lsb | vcpu_out_n_lsb<br>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.<br>Settings: LSB 32-bit data value will be sent to the host's 64-bit inbox. |

### 13.2.82 VCPU from Host Inbox Message n MSB (VCPU_IN_0_MSB - VCPU_IN_1_MSB)

#### Offset

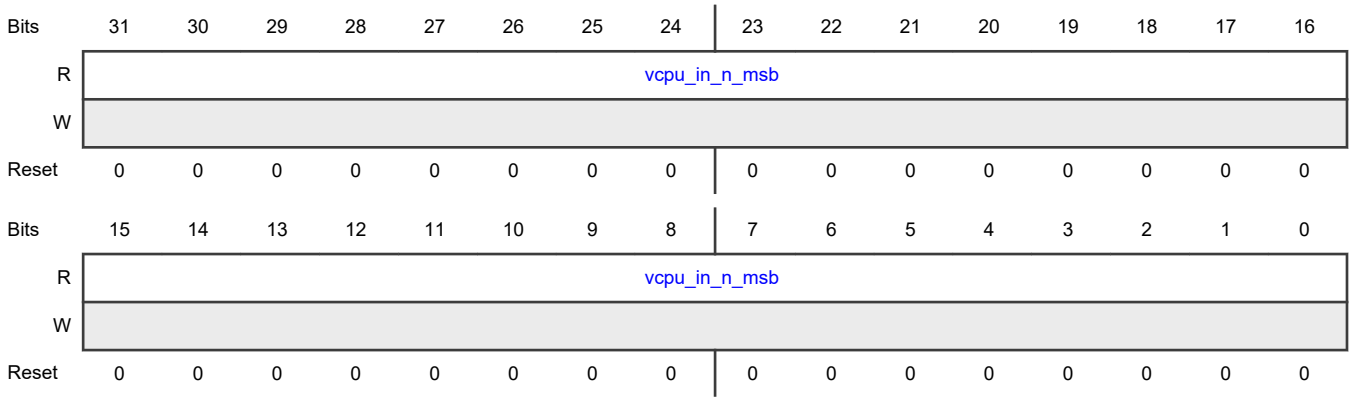
| Register | Offset |
|---|---|
| VCPU_IN_0_MSB | 650h |
| VCPU_IN_1_MSB | 658h |

#### 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-host mailbox status register (VSPA_VCPU_MBOX_STATUS) or the msg_out_*n*_valid flag in the host-VCPU mailbox status register (VSPA_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<br>vcpu_in_n_msb | vcpu_in_n_msb<br>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.<br>Settings: Reads return 32 MSBs of the message *n* 64-bit data value sent via the host outbox. |

## 13.2.83 VCPU from host inbox message n LSB register (VCPU_IN_0_LSB - VCPU_IN_1_LSB)

### Offset

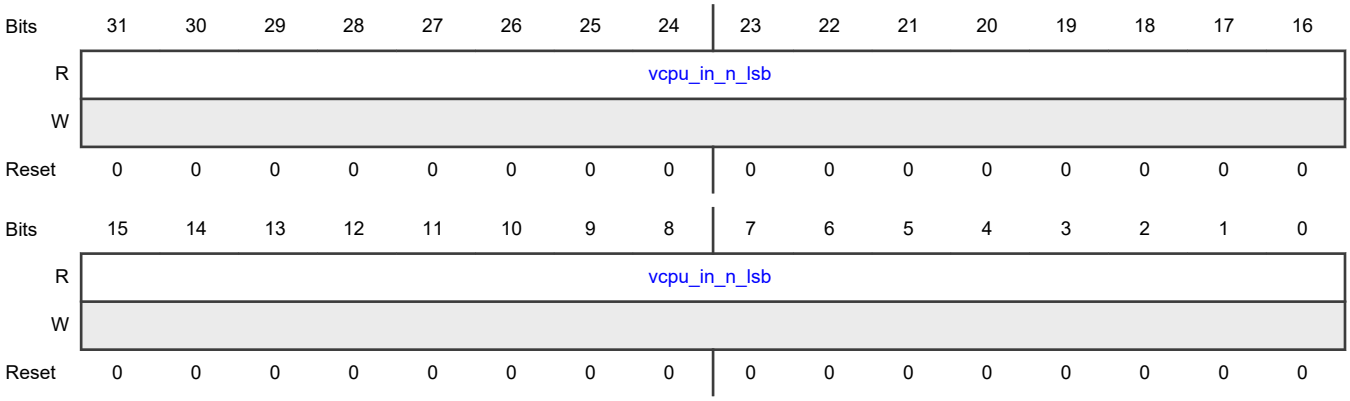
| Register | Offset |
|---|---|
| VCPU_IN_0_LSB | 654h |
| VCPU_IN_1_LSB | 65Ch |

### 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 (VSPA_VCPU_MBOX_STATUS) and the msg_out_*n*_valid flag in the host-VCPU mailbox status register (VSPA_HOST_MBOX_STATUS). This register is read only - writes have no effect.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-0<br>vcpu_in_n_lsb | vcpu_in_n_lsb<br>VCPU 64-bit message *n* input data (LSB).<br>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. |

## 13.2.84 VCPU to Host Mailbox Status register (VCPU_MBOX_STATUS)

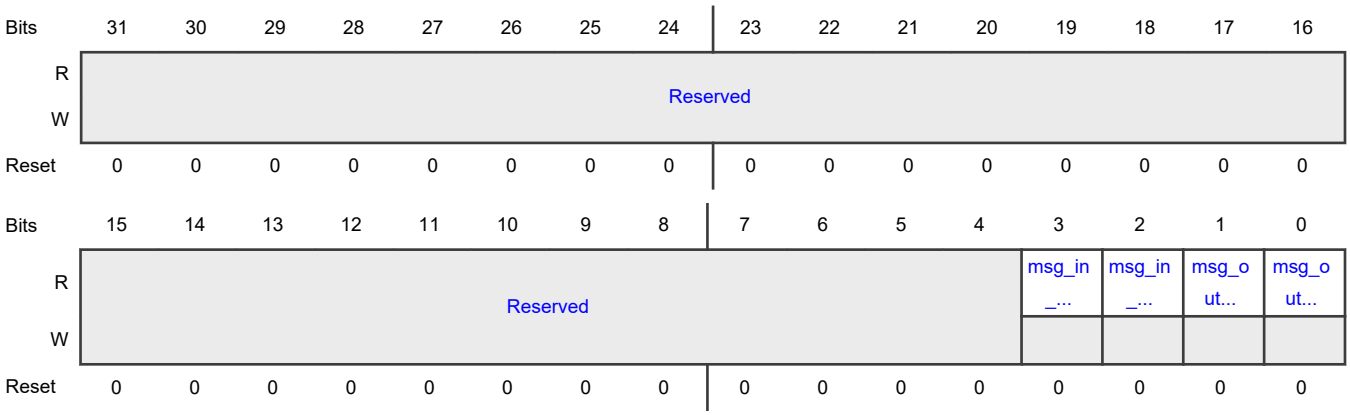
### Offset

| 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<br>— | -<br>Reserved |
| 3<br>msg_in_1_valid | msg_in_1_valid<br>64-bit inbox message 1 valid.<br>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).<br>0b - Data in the 64-bit inbox registers is not valid.<br>1b - Data in the 64-bit inbox registers is valid. |
| 2<br>msg_in_0_valid | msg_in_0_valid<br>64-bit inbox message 0 valid.<br>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).<br>0b - Data in the inbox registers VSPA_VCPU_IN_0_MSB and VSPA_VCPU_IN_0_LSB is not valid.<br>1b - Data in the inbox registers VSPA_VCPU_IN_0_MSB and VSPA_VCPU_IN_0_LSB is valid. |
| 1<br>msg_out_1_valid | msg_out_1_valid<br>64-bit outbox message 1 valid.<br>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.<br>1b - Data in the 64-bit message 1 outbox registers has not yet been read by the host. |
| 0<br>msg_out_0_valid | msg_out_0_valid<br>64-bit outbox message 0 valid.<br>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).<br>0b - No data is pending to be read in the 64-bit message 0 outbox registers.<br>1b - Data in the 64-bit message 0 outbox registers has not yet been read by the host. |

## 13.2.85 Host to VCPU Outbox Message n MSB register (HOST_OUT_0_MSB - HOST_OUT_1_MSB)

### Offset

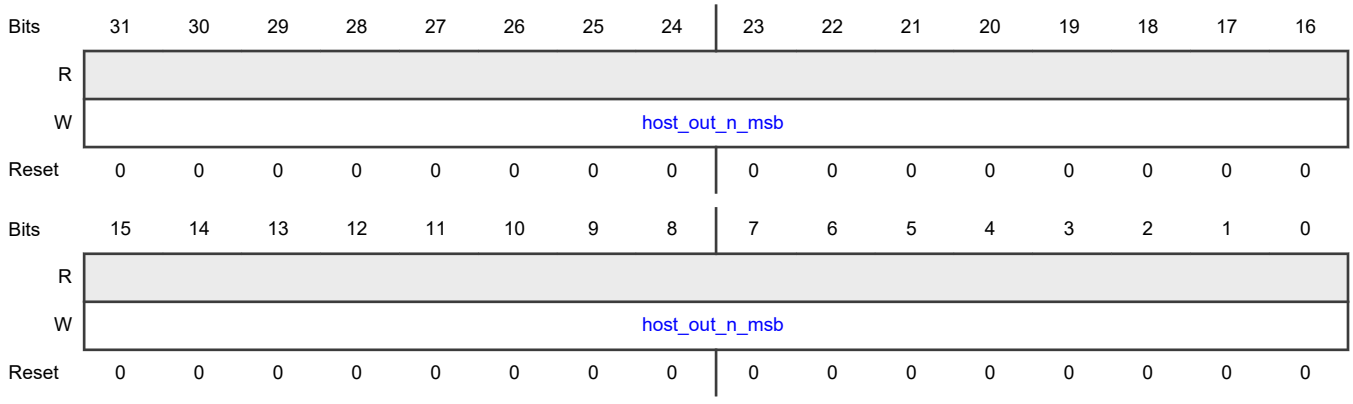
| Register | Offset |
|---|---|
| HOST_OUT_0_MSB | 680h |
| HOST_OUT_1_MSB | 688h |

### 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<br>host_out_n_msb | host_out_n_msb<br>Host output message *n* data (MSB)<br>• Writes update the 32 MSBs of a pending 64-bit message *n* that will be sent to VCPU's 64-bit inbox.<br>• The mail message will be sent in its entirety only after a write to the VSPA_HOST_OUT_*n*_LSB register.<br>• 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)<br>Settings: MSB 32-bit data value to be sent to VCPU's 64-bit message *n* inbox . |

## 13.2.86 Host to VCPU Outbox Message n LSB register (HOST_OUT_0_LSB - HOST_OUT_1_LSB)

### Offset

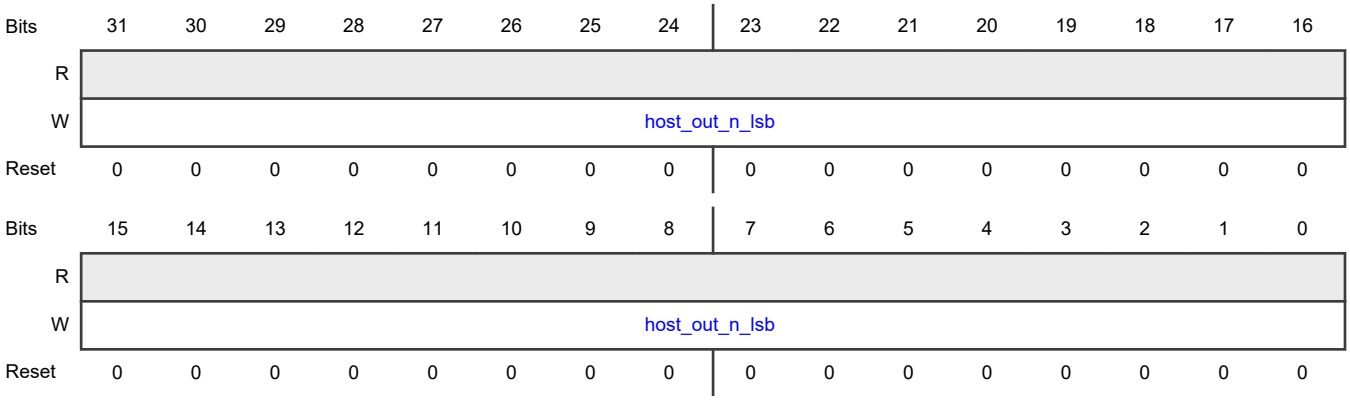
| Register | Offset |
|---|---|
| HOST_OUT_0_LSB | 684h |
| HOST_OUT_1_LSB | 68Ch |

### 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<br>host_out_n_lsb | host_out_n_lsb<br>Host output message *n* data (LSB)<br>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.<br>Settings: LSB 32-bit data value to be sent to VCPU's 64-bit inbox. |

## 13.2.87 Host from VCPU Inbox Message n MSB (HOST_IN_0_MSB - HOST_IN_1_MSB)

### Offset

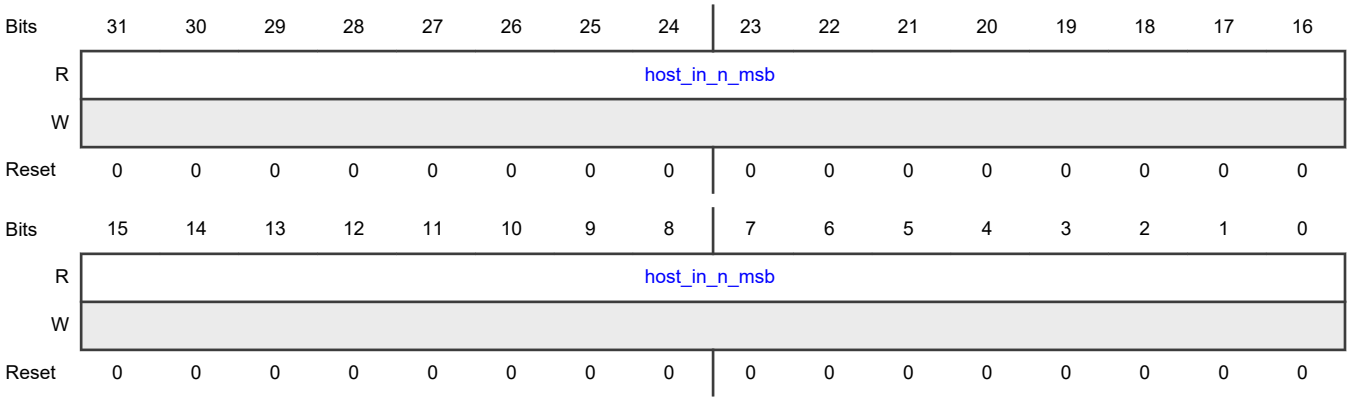
| Register | Offset |
|---|---|
| HOST_IN_0_MSB | 690h |
| HOST_IN_1_MSB | 698h |

### 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-VCPU mailbox status register (VSPA_HOST_MBOX_STATUS) or the msg_out_n_valid flag in VCPU-host mailbox status register (VSPA_HOST_MBOX_STATUS). These flags are cleared by reads of VCPU to the host 64-bit LSB inbox register.

### Diagram



### Fields

| Field | Function |
|---|---|
| 31-0<br>host_in_n_msb | host_in_n_msb<br>Host 64-bit message *n* input data (MSB). Reads the 32 MSBs of the 64-bit mail message sent by VCPU.<br>Settings: Reads return 32 MSBs of the message *n* 64-bit data value sent via VCPU outbox. |

## 13.2.88 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-VCPU mailbox status register (VSPA_HOST_MBOX_STATUS) and the msg_out_*n*_valid flag in VCPU-host mailbox status register (VSPA_HOST_MBOX_STATUS). This register is read only - writes have no effect.

### Diagram

| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R | host_in_n_lsb | | | | | | | | | | | | | | | |
| 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 | host_in_n_lsb | | | | | | | | | | | | | | | |
| W | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

### Fields

| Field | Function |
|---|---|
| 31-0<br>host_in_n_lsb | host_in_n_lsb<br>Host 64-bit message *n* input data (LSB). Reads the 32 LSBs of the 64-bit mail message sent by VCPU.<br>Settings: Reads return 32 LSBs of the message *n* 64-bit data value sent via VCPU outbox. |

## 13.2.89 Host Mailbox Status Register (HOST_MBOX_STATUS)

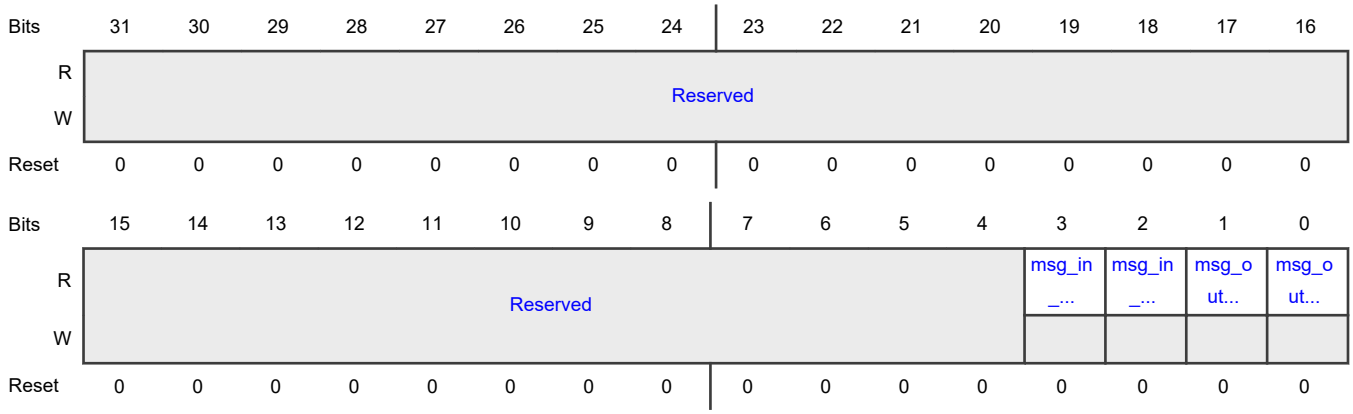
### Offset

| 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<br>— | -<br>Reserved |
| 3<br>msg_in_1_valid | msg_in_1_valid<br>64-bit inbox message 1 valid<br>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).<br>0b - Data in the 64-bit inbox registers is not valid.<br>1b - Data in the 64-bit inbox registers is valid. |
| 2<br>msg_in_0_valid | msg_in_0_valid<br>64-bit inbox message 0 valid<br>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.<br>1b - Data in the inbox registers VSPA_HOST_IN_0_MSB and VSPA_HOST_IN_0_LSB is valid |
| 1<br>msg_out_1_valid | msg_out_1_valid<br>64-bit outbox message 1 valid<br>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).<br>0b - No data is pending to be read in the 64-bit message 1 outbox registers.<br>1b - Data in the 64-bit message 1 outbox registers has not yet been read by VCPU. |
| 0<br>msg_out_0_valid | msg_out_0_valid<br>64-bit outbox message 0 valid<br>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).<br>0b - No data is pending to be read in the 64-bit message 0 outbox registers.<br>1b - Data in the 64-bit message 0 outbox registers has not yet been read by VCPU. |

### 13.2.90 IPPU Control register (IPPUCONTROL)

**Offset**

| Register | Offset |
|---|---|
| IPPUCONTROL | 700h |

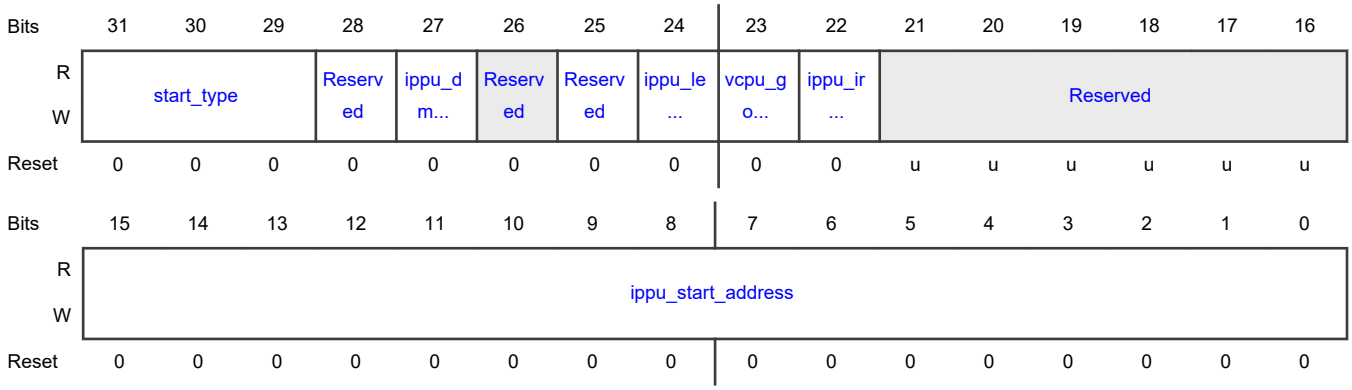
**Function**

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<br>start_type | Start type<br>Start type<br>000b - Reserved<br>001b - dma_ippu_go_en - DMA GO command Enable to IPPU[1]<br>010b - Reserved<br>011b - Reserved<br>100b - ippu_go_now[2]<br>101b - Reserved<br>110b - Reserved<br>111b - Reserved |
| 28<br>— | -<br>This bit is implemented, but has no effect |
| 27<br>ippu_dma_trigger | ippu_dma_trigger<br>Used to start a DMA transfer upon completion of IPPU code execution (ippu_done instruction is executed).<br>0b - Ignored<br>1b - Generate DMA start trigger upon completion of IPPU code execution (ippu_done instruction is executed). |
| 26<br>— | Reserved |
| 25<br>— | -<br>This bit is implemented, but has no effect |
| 24 | ippu_legacy_mem_addr |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| ippu_legacy_mem_addr | IPPU mode: legacy_mem_addr<br>0b - Turn on the mode. IPPU DMEM address pointers (aX) point to 32-bit elements.<br>1b - Turn off the mode. IPPU DMEM address pointers (aX) point to 16-bit elements. |
| 23<br>vcpu_go_en | IPPU vcpu_go_enable<br>Enables the VCPU GO upon executing the IPPU done instruction.<br>When set, it causes a VCPU_GO upon IPPU executing the done instruction by setting the the ippu_go status bit in the VSPA_CONTROL register bit 1.<br>Refer to VSPA_CONTROL for additional information.<br>0b - No Go request is generated when the IPPU executes a done instruction.<br>1b - Go request is generated when the IPPU executes a done instruction by setting the ippu_go status in the VSPA_CONTROL register. |
| 22<br>ippu_irq_en | IPPU done interrupt enable<br>Enables the IPPU interrupt upon executing the IPPU done instruction.<br>When set, IPPU executes a done instruction will set the irq_pend_ippu_done status flag in the VSPA_STATUS register bit 1.<br>This will assert the ippu_irq signal.<br>Refer to VSPA_STATUS register for additional information.<br>0b - No interrupt is generated upon executing the IPPU done instruction.<br>1b - An interrupt is generated upon executing the IPPU done instruction |
| 21-16<br>— | -<br>Reserved |
| 15-0<br>ippu_start_address | ippu_start_address<br>ippu_start_address- IPPU Start Address<br>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.<br>The ippu_start_address field is used by the IPPU only in conjunction with the start_type field.<br>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.<br>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 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. 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.

### 13.2.91 IPPU Status register (IPPUSTATUS)

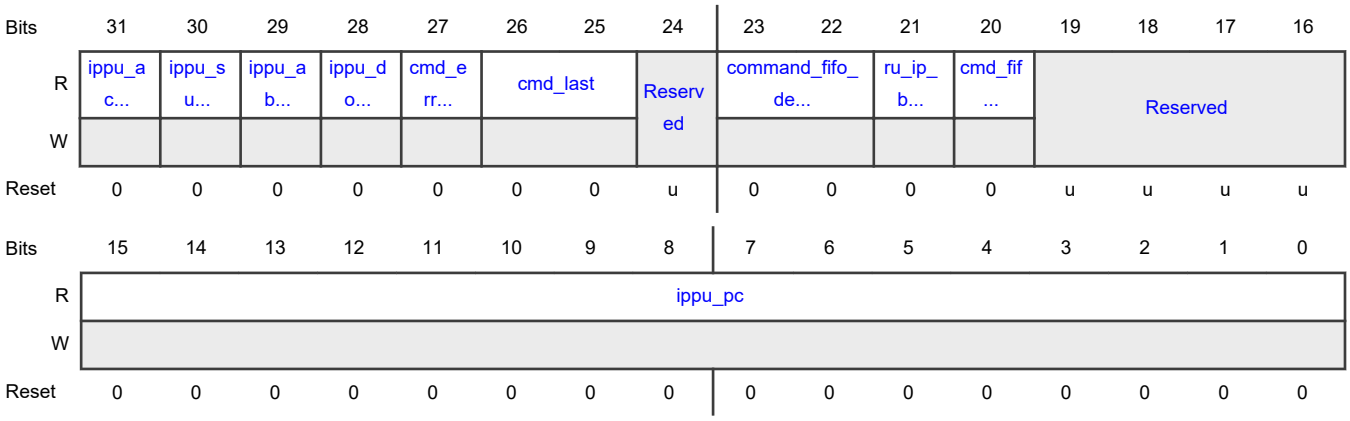
#### Offset

| Register | Offset |
|---|---|
| IPPUSTATUS | 704h |

#### Function

IPPU Status register

#### Diagram




#### Fields

| Field | Function |
|---|---|
| 31 | ippu_active (busy_or_pending) |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| ippu_active | 0b - Not busy and command fifo is empty<br>1b - Busy is set or there is at least one command pending in the FIFO. |
| 30<br>ippu_suspended | ippu_suspended<br>ippu_suspended - IPPU Suspended<br>ippu_suspended indicates whether the IPPU core is suspended or not.<br>This bit is set when the IPPU enters the Suspended state. It is cleared when the IPPU is not in the Suspended state.<br>ippu_suspended can be read by the host software or VCPU software at any time. Writing this bit has no effect.<br>0b - IPPU not in Suspended state.<br>1b - IPPU in Suspended state. |
| 29<br>ippu_aborted | ippu_aborted<br>ippu_aborted - IPPU Aborted<br>Indicates whether the IPPU core received the ippu_abort command while in the Running state.<br>ippu_aborted is set when the IPPU is in the Running state and the ippu_abort bit in the IPPURC register is set.<br>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, DMA done start).<br>ippu_aborted can be read by the host software or VCPU software at any time. Writing this bit has no effect.<br>0b - After Reset or IPPU in the Running state.<br>1b - ippu_abort bit was set while the IPPU was in the Running state. |
| 28<br>ippu_done | ippu_done<br>ippu_done - IPPU Done<br>Indicates whether the IPPU core executed the done instruction.<br>ippu_done is set when the IPPU executes the done instruction.<br>ippu_done is cleared after reset and when the IPPU enters the Running state.<br>ippu_done can be read by the host software or VCPU software at any time. Writing this bit has no effect.<br>0b - After Reset or IPPU busy<br>1b - IPPU executed the done instruction and in Idle sate. |
| 27<br>cmd_error | command error<br>command error - IPPU command fifo error<br>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. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 0b - After reset or no command error<br>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<br>cmd_last | command last source<br>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.<br>00b - After Reset or when ip_ippu_go_now is set.<br>01b - Reserved.<br>10b - The dma_ippu_go_en bit was set and the dma_ippu_go signal was asserted.<br>11b - Reserved |
| 24<br>— | -<br>Reserved |
| 23-22<br>command_fifo_depth | Command Fifo Depth<br>command_fifo_depth is the number of pending or inprogress operations in the command FIFO.<br>A maximum of 2 operations can be pending or in progress.<br>Further attempts to start an operation will set command_error.<br>00b - Empty, no entries in the fifo<br>01b - One entry in the fifo<br>10b - Full, two or more entries in the fifo<br>11b - Reserved |
| 21<br>ru_ip_busy | IPPU Busy<br>ippu_busy indicates whether the IPPU core is Idle or busy executing code.<br>ippu_busy is set when the IPPU enters the Running state.<br>ippu_busy is cleared after reset and when the IPPU executes the done instruction.<br>ippu_busy can be read by the host software or VCPU software at any time. Writing this bit has no effect.<br>0b - Idle state<br>1b - Running state |
| 20<br>cmd_fifo_full | Command Fifo Full<br>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.<br>0b - Command fifo is not full<br>1b - Command fifo is full; Command_depth equals 2. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 19-16<br>— | -<br>Reserved |
| 15-0<br>ippu_pc | ippu_pc<br>ippu_pc - IPPU Program Counter<br>Reading this value provides approximate value of the pc at the time the IPPU_STATUS register is read.<br>ippu_pc can be read by the host software or VCPU software at any time. Writing this field has no effect.<br>Settings: ippu_pc provides approximate value of the pc at the time this register is read. |

## 13.2.92 IPPU Run Control register (IPPURC)

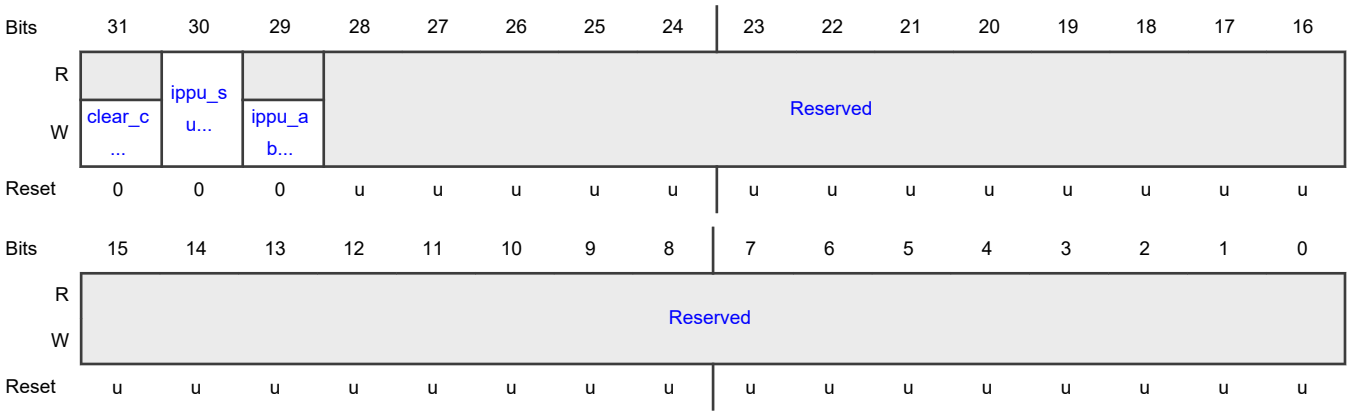
### Offset

| Register | Offset |
|---|---|
| IPPURC | 708h |

### Function

IPPU Run Control register

### Diagram




### Fields

| Field | Function |
|---|---|
| 31<br>clear_cmd_fifo_error | Clear command fifo error bit<br>Write only bit. Reading always return zero.<br>Writing one to this bit clears the IPPU command error bit in the status register. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 0b - Do not clear<br>1b - Writing 1 clears the IPPU command error bit in the status register |
| 30<br>ippu_suspend | IPPU suspend bit<br>The ippu_suspend bit acts as stop/continue control for the IPPU.<br>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.<br>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.<br>0b - IPPU runs normally<br>1b - While in Running state, the IPPU suspends instruction fetch and execution. No effect while in Idle state |
| 29<br>ippu_abort | IPPU Abort<br>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.<br>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.<br>When the abort bit is set, the IPPU clears all pending FIFO commands.<br>**NOTE**<br>The ippu_abort bit is reset to zero and is self clearing. Reading this bit will always return zero.<br>0b - Do nothing.<br>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<br>— | -<br>Reserved |

## 13.2.93 IPPU Arg Base Address register (IPPUARGBASEADDR)

**Offset**

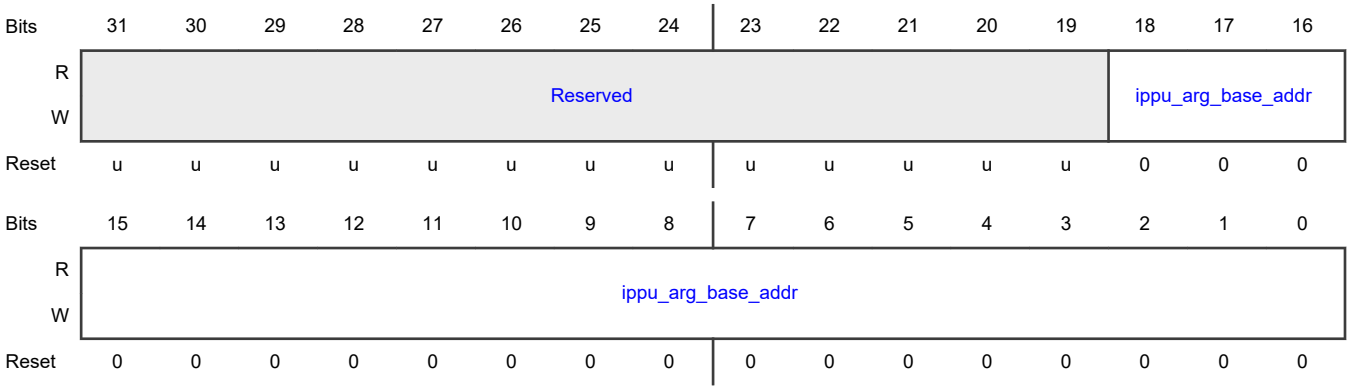
| Register | Offset |
|---|---|
| IPPUARGBASEADDR | 70Ch |

### 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




### Fields

| Field | Function |
|---|---|
| 31-19<br>— | -<br>Reserved |
| 18-0<br>ippu_arg_base_addr | ippu_arg_base_addr<br>ippu_arg_base_addr - IPPU Argument Base Address<br>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.<br>Settings: when the IPPU is accessing its arguments, this field is used as a start address for the argument memory space. |

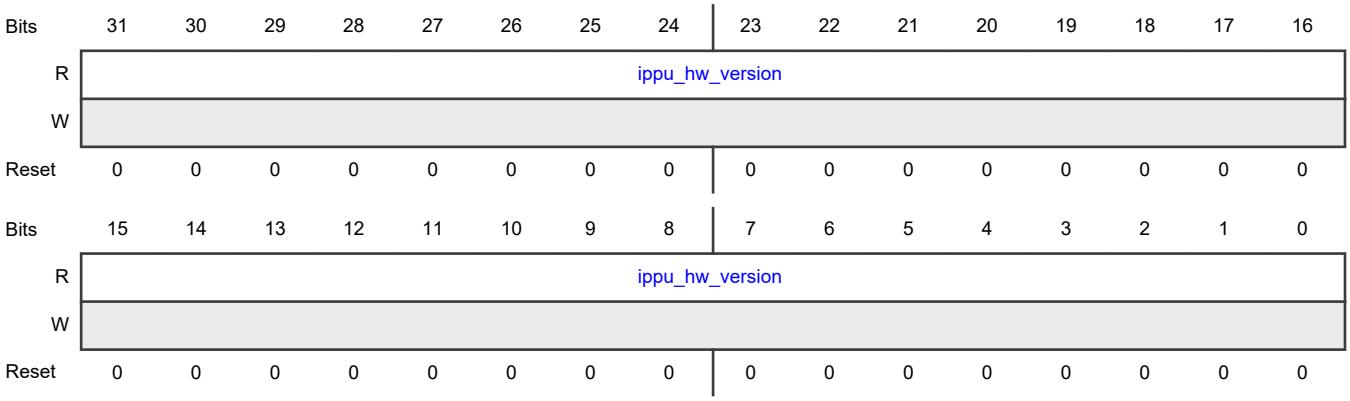
## 13.2.94 IPPU Hardware Version (IPPUHWVER)

### Offset

| Register | Offset |
|---|---|
| IPPUHWVER | 710h |

**Function**
IPPU Hardware Version

**Diagram**




**Fields**

| Field | Function |
|---|---|
| 31-0<br>ippu_hw_version | ippu_hw_version<br>IPPU Hardware Version<br>This field indicates the IPPU hardware version.<br>ippu_hw_version can be read by the host software or VCPU software at any time. Writing this field has no effect. |

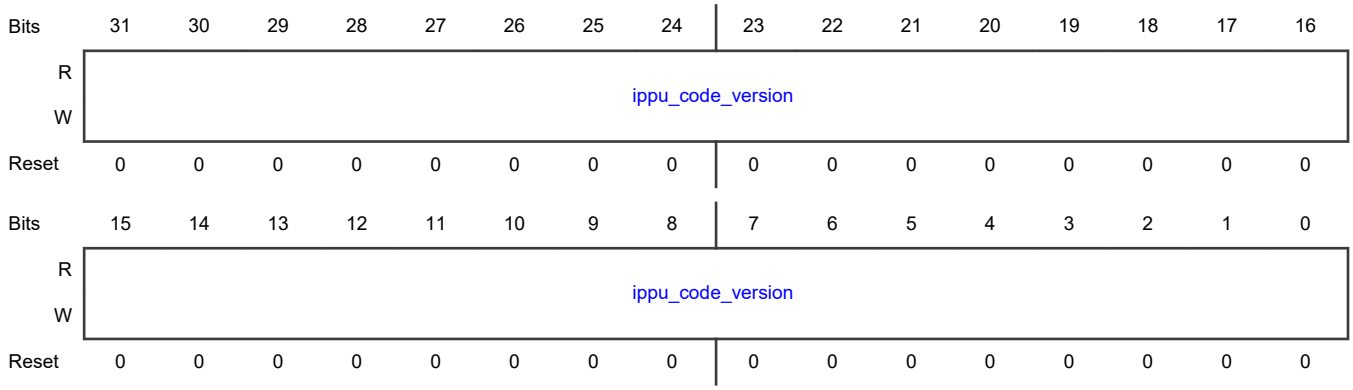
## 13.2.95 IPPU Software Version (IPPUSWVER)

**Offset**

| Register | Offset |
|---|---|
| IPPUSWVER | 714h |

**Function**
IPPU Software Version

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-0<br>ippu_code_version | ippu_code_version<br>IPPU software code version<br>This field can be used to indicate the version of the IPPU PRAM assembly code.<br>• IPPU software configures this register during global initialization after download of a new IPPUPRAM image.<br>• IPPUPRAM version numbers are non-zero.<br>• This register does not affect the HW operation on the IPPU. It provides a place holder for the programmer. |

# Chapter 14
# Debug registers

## 14.1 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.

### 14.1.1 VSPA_DBG memory map

VSPA0.VSPA_DBG0 base address: FC04_8000h

VSPA1.VSPA_DBG1 base address: FC04_A000h

VSPA2.VSPA_DBG2 base address: FC04_C000h

VSPA3.VSPA_DBG3 base address: FC04_E000h

VSPA4.VSPA_DBG4 base address: FC05_0000h

VSPA5.VSPA_DBG5 base address: FC05_2000h

VSPA6.VSPA_DBG6 base address: FC05_4000h

VSPA7.VSPA_DBG7 base address: FC05_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 |
| 80Ch | Trace Control register 1 (TC1) | 32 | RW | See description |
| 83Ch | Debug Halt Action Control register (HACR) | 32 | RW | See description |
| 840h | Debug Resume Action Control register (RACR) | 32 | RW | See description |

*Table continues on the next page...*

*Table continued from the previous page...*

| Offset | Register | Width (In bits) | Access | Reset value |
|---|---|---|---|---|
| 844h | Debug Program Trace Sync Action Control register (PTSACR) | 32 | RW | See description |
| 848h | Debug Program Trace Start Action Control register (PSTACR) | 32 | RW | See description |
| 84Ch | Debug Program Trace Stop Action Control register (PSPACR) | 32 | RW | See description |
| 850h | Debug Data Trace Start Action Control register (DSTACR) | 32 | RW | See description |
| 854h | Debug Data Trace Stop Action Control register (DSPACR) | 32 | RW | See description |
| 858h | Debug Data Trace Message Generation Action Control register (DTMACR) | 32 | RW | See description |
| 85Ch | Debug WPM Generation Action Control register (WPMACR) | 32 | RW | See description |
| 860h | Debug TSM Control register (TSMCR) | 32 | RW | See description |
| 864h | Debug VSPA Timestamp Counter (Upper) register (TSMCNT_U) | 32 | RO | See description |
| 868h | Debug VSPA Timestamp Counter (Lower) register (TSMCNT_L) | 32 | RO | 0000_0000h |
| 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 - 89Ch | Debug Cross Trigger Out a Action Control registers (CTO0ACR - CTO7ACR) | 32 | RW | See description |
| 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 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Offset | Register | Width (In bits) | Access | Reset value |
|---|---|---|---|---|
| 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 |
| 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 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Offset | Register | Width (In bits) | Access | Reset value |
|---|---|---|---|---|
| 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 |
| 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 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Offset | Register | Width (In bits) | Access | Reset value |
|---|---|---|---|---|
| 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 |

### 14.1.2 Global Debug Enable register (GDBEN)

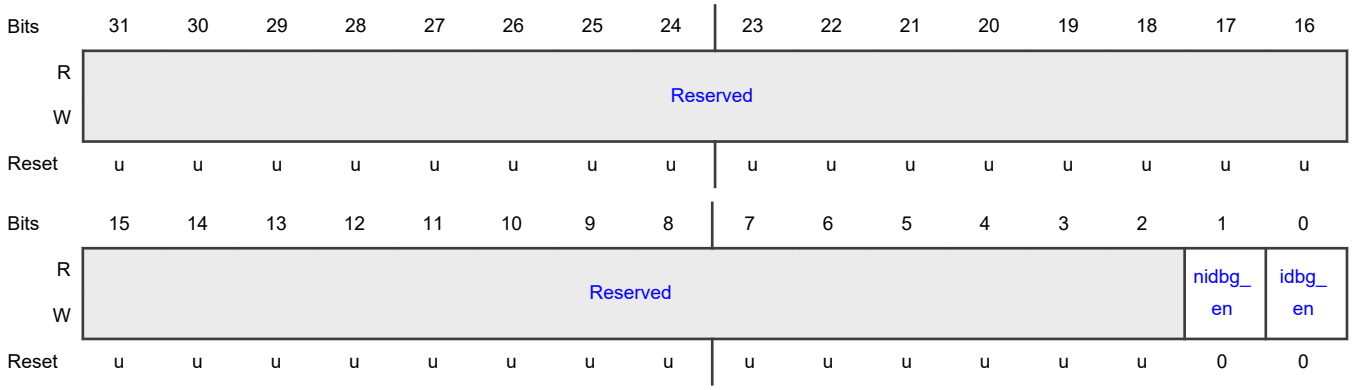
#### Offset

| 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<br>— | -<br>Reserved |
| 1<br>nidbg_en | nidbg_en<br>Non-invasive debug mode enable<br>This bit enables/disables non-invasive debug mode.<br>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".<br>0b - Non-Invasive debug mode disabled (no trace Messages will be produced).<br>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<br>idbg_en | idbg_en<br>Invasive debug mode enable<br>This is the enable for invasive (halting) debug mode.<br>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".<br>0b - Halting debug mode is disabled (VSP cannot be halted).<br>1b - Halting debug mode is enabled (VSP can be halted and enter debug mode) if the dbg_dbgen input is driven to a "1". |

### 14.1.3 Debug Run Control register (RCR)

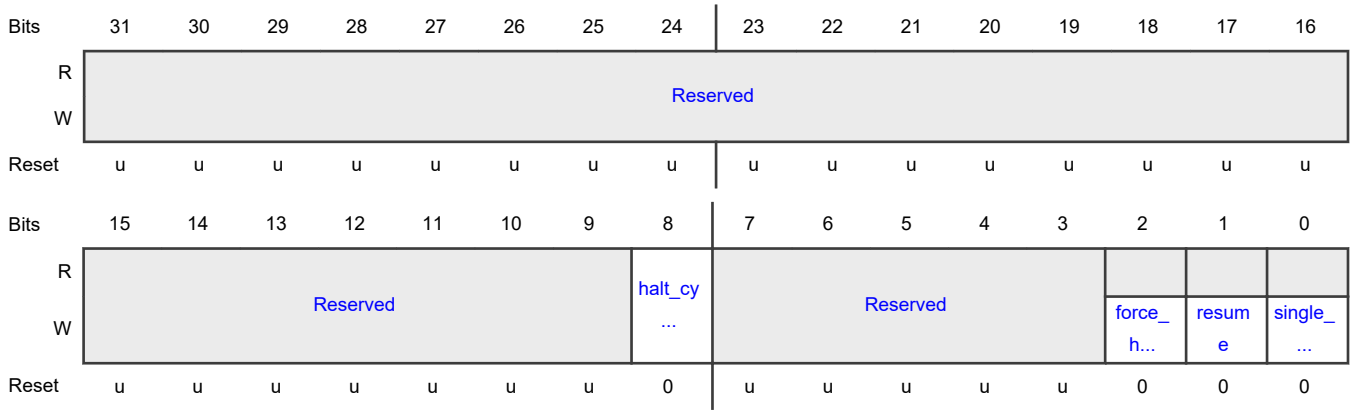
#### Offset

| 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<br>— | -<br>Reserved |
| 8<br>halt_cyc_counter | halt_cyc_counter<br>This bit controls halting the cycle counter, VSPA_CYC_COUNTER_MSB/VSPA_CYC_COUNTER_LSB registers, when in halted state.<br>0b - Cycle counter continues counting when VSPA is in the halted state<br>1b - Cycle counter halts counting when VSPA is in the halted state |
| 7-3<br>— | -<br>Reserved |
| 2<br>force_halt | force_halt<br>Force halt - Cause VSP to enter halting debug mode |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 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".<br>Settings: Write-only; Always reads as '0'. |
| 1<br>resume | resume<br>Resume normal operation - exit halting debug mode<br>When this bit is written to '1', VSP exits halting debug mode and resumes execution where it left off prior to the halt.<br>This command bit is write only - it will always read as "0".<br>Settings: Write-only; Always reads as '0'. |
| 0<br>single_step | single_step<br>single step - execute a single instruction, then halt again<br>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).<br>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.<br>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.<br>This command bit is write only - it will always read as "0".<br>Settings: Write-only; Always reads as '0'. |

### 14.1.4 Debug Run Control Status register (RCSTATUS)

**Offset**

| Register | Offset |
|---|---|
| RCSTATUS | 808h |

**Function**

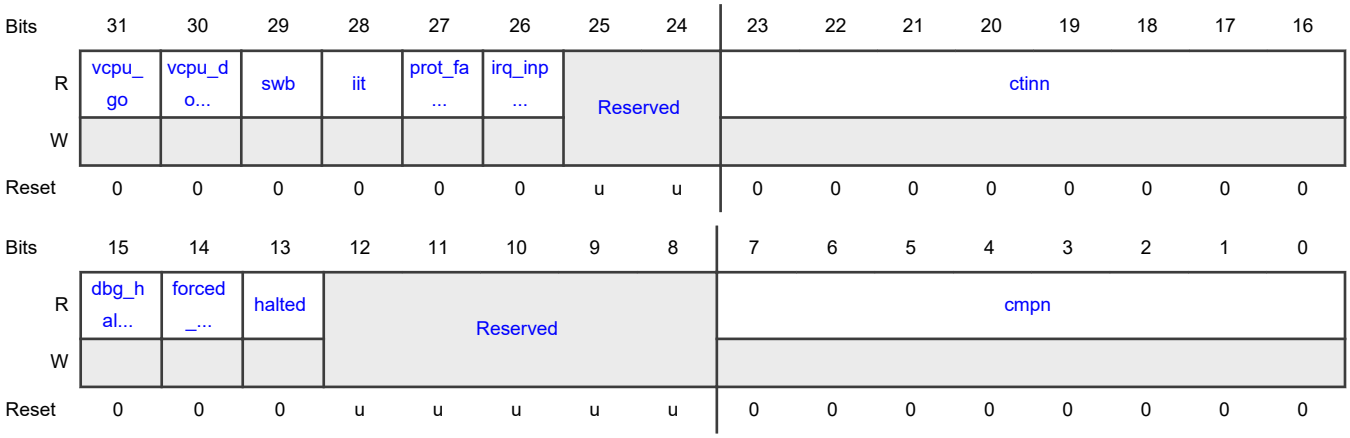
This 32-bit read-only status register is used to determine the reason for VSP being halted. Any or all these flags could potentially be asserted, indicating what caused the halt. These flags will self-clear upon exit from halting debug mode (resume).

**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




### Fields

| Field | Function |
|---|---|
| 31<br>vcpu_go | vcpu_go<br>VCPU Go event<br>When '1', this flag indicates that VSPA was halted due to occurrence of VCPU "Go" event.<br>Note that this flag self-clears on exit from halting debug mode.<br>0b - VCPU "Go" NOT reason for VSPA being halted.<br>1b - VCPU "Go" one reason for VSPA being halted. |
| 30<br>vcpu_done | vcpu_done<br>VCPU done instruction<br>When '1', this flag indicates that VSPA was halted due to execution of VCPU "Done" instruction.<br>Note that this bit self-clears on exit from halting debug mode.<br>0b - VCPU "Done" NOT reason for VSPA being halted.<br>1b - VCPU "Done" one reason for VSPA being halted. |
| 29<br>swb | swb<br>Software breakpoint<br>When '1', this flag indicates that VSPA was halted due to occurrence of Software Breakpoint.<br>Note that this flag self-clears on exit from halting debug mode.<br>0b - Software Breakpoint NOT reason for VSPA being halted.<br>1b - Software Breakpoint one reason for VSPA being halted. |
| 28 | iit |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| iit | VCPU illegal instruction<br>When '1', this flag indicates that VSPA was halted due to execution of VCPU illegal instruction.<br>Note that this bit self-clears on exit from halting debug mode.<br>0b - VCPU “illegal instruction” NOT reason for VSPA being halted.<br>1b - VCPU "illegal instruction" one reason for VSPA being halted. |
| 27<br>prot_fault | prot_fault<br>VCPU protection fault<br>When '1', this flag indicates that VSPA encountered a protection fault.<br>Note that this bit self-clears on exit from halting debug mode.<br>0b - VCPU "protection fault" NOT reason for VSPA being halted.<br>1b - VCPU "protection fault" one reason for VSPA being halted. |
| 26<br>irq_input | irq_input<br>VCPU interrupt request<br>When '1', This flag indicates that VCPU was interrupted as result of the input signal “vcpu_irq” assertion.<br>Note that this bit self-clears on exit from halting debug mode.<br>0b - VCPU “interrupt request” NOT reason for VSPA being halted.<br>1b - VCPU "interrupt request" one reason for VSPA being halted. |
| 25-24<br>— | -<br>Reserved |
| 23-16<br>ctinn | ctinn<br>dbg_trig_in_req[n] input triggered where n is 7-0<br>When '1', This flag indicates that VSPA was halted due to assertion of the dbg_trig_in_req[n] input.<br>Note that this flag self-clears on exit from halting debug mode.<br>00000000b - dbg_trig_in_req[n] not a reason for VSPA being halted.<br>00000001b - dbg_trig_in_req[n] one reason for VSPA being halted. |
| 15<br>dbg_halt_req | dbg_halt_req<br>dbg_halt_req dedicated input triggered<br>When '1', This flag indicates that VSPA was halted due to assertion of the dedicated halt request input.<br>Note that this flag self-clears on exit from halting debug mode.<br>0b - dbg_halt_req not a reason for VSPA being halted.<br>1b - dbg_halt_req one reason for VSPA being halted. |
| 14 | forced_halt |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| forced_halt | Debugger forced halt<br>When '1', This flag indicates that the debugger set the force_halt control bit in the RCR register.<br>Note that this flag self-clears on exit from halting debug mode.<br>0b - VSPA not halted due to debugger forced halt.<br>1b - VSPA halted due to debugger forced halt. |
| 13<br>halted | halted<br>Halted<br>When '1', This flag indicates that the VCPU, IPPU, and DMA engines have all halted.<br>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.<br>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.<br>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.<br>**NOTE**<br>This is self-clearing bit. Writing a one sets the bit. The bit will clear automatically and it will always read as zero.<br>0b - Halt action (if applicable) has not completed.<br>1b - Halt action has completed - all engines are halted. |
| 12-8<br>— | -<br>Reserved |
| 7-0<br>cmpn | cmpn<br>Comparator n event occurred where n is 7-0<br>When '1', This flag indicates that VSPA was halted due to occurrence of Comparator n event.<br>Note that this bit self-clears on exit from halting debug mode.<br>00000000b - Comparator n not reason for VSPA being halted.<br>00000001b - Comparator n one reason for VSPA being halted. |

### 14.1.5 Trace Control register 1 (TC1)

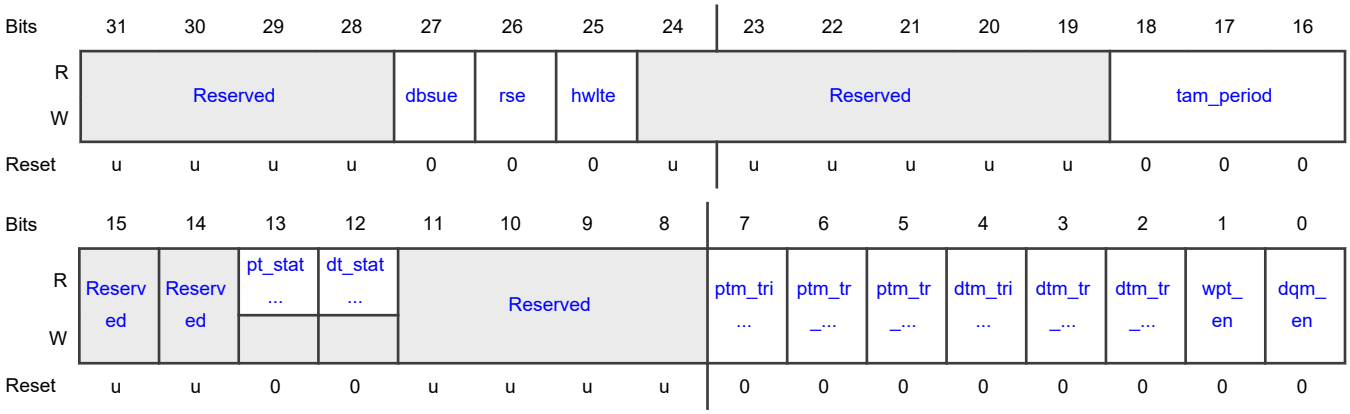
#### Offset

| Register | Offset |
|---|---|
| TC1 | 80Ch |

#### Function

This 32-bit read-write register is compatible with the recommendation in the Nexus standard for control register DC1. It contains fields which enable/disable the various trace modes of VSP trace and configurable aspects of program trace messaging (if enabled).

#### Diagram




#### Fields

| Field | Function |
|---|---|
| 31-28<br>— | -<br>Reserved |
| 27<br>dbsue | dbsue<br>Direct branch synchronization upgrade enable<br>Upgrade taken direct branches to produce full Program Trace Indirect Branch w/sync (PTIBw/sync) message.<br>When the dbsue bit is cleared, both taken and not-taken direct branches would be tallied as HISTORY bits – 0 for non-taken branch and 1 for taken branch.When the dbsue bit is set, only taken direct (excluding non-taken) branches would result in a PTIB messages. When dbsue bit is set, no HISTORY bit is tailed in the history buffer. When the PTIB message is sent, the History buffer is flushed (if not empty), and a PCM HIST message is sent with the H-CODE=0.<br>0b - Taken direct branch is reflected in PCM HIST message.<br>1b - Taken direct branch produces PTIBw/sync message. |
| 26 | rse |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| rse | Return stack enable<br>Enables use of return stack tracking (if implemented). If return stack tracking is implemented and this bit is enabled, rts instructions can be tracked as a bit in the PCM message History field, which reduces trace output. Otherwise, a full PTIBw/sync message is produced to provide the full target address (change of flow) for the rts instruction.<br>0b - rts instruction produces PTIBw/sync message.<br>1b - rts instruction tracked in PCM message HIST field. If the TC1[dbsue] bit is set, this bit will be ignored and the rts instruction will produce a PTIBw/sync message. |
| 25<br>hwlte | hwlte<br>Hardware loop trace enable<br>Enables trace of VSP hardware loops.<br>0b - VSP hardware loop tracking disabled - no PTM messages produced.<br>1b - VSP hardware loop tracking enabled - PCM messages are generated as necessary to be able to track hardware loop execution. |
| 24-19<br>— | -<br>Reserved |
| 18-16<br>tam_period | tam_period<br>Trace alignment message period<br>Enables and sets the period of periodically generated Trace Alignment Messages (TAM). When this field is 0, no periodic TAMs will be produced. When it is set to any other value, periodic TAMs will be produced after every N 32-bit trace packets, where<br>N = 2 (tam_period + 5)<br>000b - No periodic TAM will be produced.<br>001b - Produce a TAM after every 64 32-bit trace packets<br>010b - Produce a TAM after every 128 32-bit trace packets<br>011b - Produce a TAM after every 256 32-bit trace packets<br>100b - Produce a TAM after every 512 32-bit trace packets<br>101b - Produce a TAM after every 1024 32-bit trace packets<br>110b - Produce a TAM after every 2048 32-bit trace packets<br>111b - Produce a TAM after every 4096 32-bit trace packets |
| 15<br>— | -<br>Reserved |
| 14<br>— | -<br>Reserved |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 13<br>pt_status | pt_status<br>Program trace status<br>This flag indicates whether Program Trace Messaging is switched on or off. It is read only and self-clearing.<br>0b - PTM switched off.<br>1b - PTM switched on. |
| 12<br>dt_status | dt_status<br>Data trace status<br>This flag indicates whether Data Trace Messaging is switched on or off. It is read only and self-clearing.<br>0b - DTM switched off.<br>1b - DTM switched on. |
| 11-8<br>— | -<br>Reserved |
| 7<br>ptm_trig_en | ptm_trig_en<br>Program trace message trigger On/Off enable<br>This bit enables program trace messaging to be triggered on/off via hardware (either various debug events configured to start/stop PTM trace or appropriately configured hardware cross-trigger inputs).<br>The functionality of this bit is overridden by both the ptm_tr_stop and ptm_tr_start (bits 6 and 5 respectively).<br>0b - HW events cannot trigger PTM on/off.<br>1b - HW events can trigger PTM on/off, if bits 5 and 6 are clear. |
| 6<br>ptm_tr_stop | ptm_tr_stop<br>Program trace messaging (force) stop/disable<br>This bit disables Program Trace Messaging. If this bit is set (1), then Program Trace messaging is disabled (off) irrespective of the settings of ptm_trig_en and ptm_tr_start (bits 7 and 5 respectively).In other words, setting this bit disables PTM and overrides the ability for PTM to be triggered on/off via hardware.<br>0b - PTM trace not disabled via this bit.<br>1b - PTM trace disabled and PTM Trace configuration bits 7 and 5 are overridden. |
| 5<br>ptm_tr_start | ptm_tr_start<br>Program trace messaging (force) start/enable<br>This bit enables program trace messaging. If this bit is set (1) and ptm_tr_stop is not set (0), then program trace messaging is enabled (on), irrespective of the state of ptm_trig_en (bit 7).In other words, setting this bit overrides the ability for PTM to be triggered on/off via hardware.<br>0b - PTM trace not enabled via this bit.<br>1b - PTM trace enabled if bit 6 is not set. PTM trace configuration bit 7 is overridden. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 4<br>dtm_trig_en | dtm_trig_en<br>Data trace message trigger on/off enable<br>This bit enables data trace messaging to be triggered on/off via hardware (either various debug events configured to start/stop DTM trace or appropriately configured hardware cross-trigger inputs).<br>The functionality of this bit is overridden by both the dtm_tr_stop and dtm_tr_start (bits 3 and 2 respectively).<br>0b - HW events cannot trigger DTM on/off.<br>1b - HW events can trigger DTM on/off, if bits 3 and 2 are clear. |
| 3<br>dtm_tr_stop | dtm_tr_stop<br>Data trace messaging (force) stop/disable<br>This bit disables data trace messaging. If this bit is set (1), then data trace messaging is disabled (off) irrespective of the settings of dtm_trig_en and dtm_tr_start (bits 4 and 2 respectively).In other words, setting this bit disables DTM and overrides the ability for DTM to be triggered on/off via hardware.<br>0b - DTM trace not disabled via this bit.<br>1b - DTM trace disabled and DTM Trace configuration bits 4 and 2 are overridden. |
| 2<br>dtm_tr_start | dtm_tr_start<br>Data trace messaging (force) start/enable<br>This bit enables data trace messaging. If this bit is set (1) and dtm_tr_stop is NOT set (0), then Data Trace messaging is enabled (on), irrespective of the state of dtm_trig_en (bit 4).In other words, setting this bit overrides the ability for DTM to be triggered on/off via hardware.<br>0b - DTM trace not enabled via this bit.<br>1b - DTM trace enabled if bit 3 is not set. DTM trace configuration bit 4 is overridden. |
| 1<br>wpt_en | wpt_en<br>Watchpoint message enable<br>This bit enables or disables watchpoint messaging. If enabled, watchpoint messages will be sent whenever and event occurs that is mapped to produce the action of generating a WPM (via the Debug WPM Generation Action Control register aka WPMACR).<br>0b - WPM disabled<br>1b - WPM enabled |
| 0<br>dqm_en | dqm_en<br>Data acquisition message enable<br>This bit enables or disables Data Acquisition Messaging. If enabled, data acquisition messages will be sent whenever VSP or the host writes to either of the following registers:<br>VSP to DQM Trace Small Outbox (DQM_SMALL)<br>VSP to DQM Trace Large Outbox LSB (DQM_LARGE_LSB) |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 0b - DQM disabled.<br>1b - DQM enabled. |

### 14.1.6 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 | vcpu_go | vcpu_do... | swb | vcpu_il... | Reserved | | | | ctinn | | | | | | | |
| W | | | | | | | | | | | | | | | | |
| 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 | Reserved | | | | | | | | cmpn | | | | | | | |
| W | | | | | | | | | | | | | | | | |
| Reset | u | u | u | u | u | u | u | u | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

#### Fields

| Field | Function |
|---|---|
| 31<br>vcpu_go | vcpu_go<br>Halt on VCPU Go<br>Controls whether VSPA will be halted on "Go" events.<br>0b - Disable VSPA halt on occurrence of a VCPU "Go" event.<br>1b - Enable VSPA halt on occurrence of a VCPU "Go" event. |
| 30<br>vcpu_done | vcpu_done<br>Halt on VCPU Done<br>Controls whether VSPa will be halted on "Done" events. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 0b - Disable VSPA halt on occurrence of a VCPU "Done" event.<br>1b - Enable VSPA halt on occurrence of a VCPU "Done" event. |
| 29<br>swb | swb<br>Halt on VCPU software breakpoint<br>Controls whether VSPA will be halted on Software Breakpoint events.<br>0b - Disable VSPA halt on occurrence of software breakpoint.<br>1b - Enable VSPA halt on occurrence of software breakpoint. |
| 28<br>vcpu_illop | vcpu_illop<br>Halt on illegal instruction trap.<br>Controls whether VSPA will be halted on illegal instruction trap events.<br>0b - Disable VSPA halt on occurrence of a VCPU illegal instruction trap event.<br>1b - Enable VSPA halt on occurrence of a VCPU illegal instruction trap event. |
| 27-24<br>— | -<br>Reseved |
| 23-16<br>ctinn | ctinn<br>Halt on VSPA cross trigger input n where n is 7-0<br>Controls whether VSPA will be halted on dbg_trig_in_req[n] events.<br>00000000b - Disable VSPA halt on occurrence of cross trigger in n.<br>00000001b - Enable VSPA halt on occurrence of cross trigger in n. |
| 15-8<br>— | -<br>Reserved |
| 7-0<br>cmpn | cmpn<br>Halt on VSPA comparator n match where n is 7-0<br>Controls whether or not VSPA will be halted on comparator n match events.<br>00000000b - Disable VSPA halt on occurrence of comparator n match.<br>00000001b - Enable VSPA halt on occurrence of comparator n match. |

### 14.1.7 Debug Resume Action Control register (RACR)

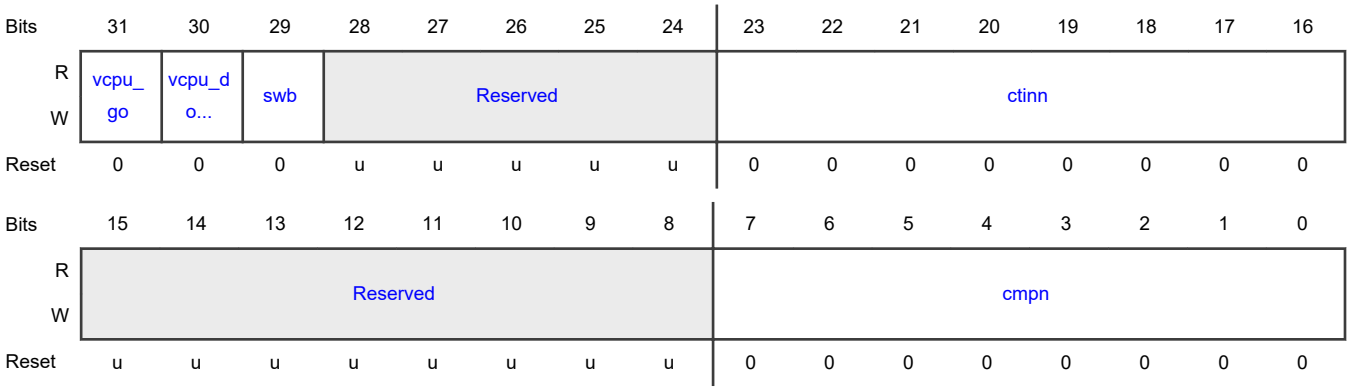
**Offset**

| 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<br>vcpu_go | vcpu_go<br>Resume on VCPU Go<br>Controls whether VSPA will resume (if halted) on "Go" events.<br>0b - Disable VSPA resume on occurrence of a VCPU "Go" event.<br>1b - Enable VSPA resume on occurrence of a VCPU "Go" event. |
| 30<br>vcpu_done | vcpu_done<br>Resume on VCPU Done<br>Controls whether VSPA will resume (if halted) on "Done" events.<br>0b - Disable VSPA resume on occurrence of a VSP "Done" event.<br>1b - Enable VSPA resume on occurrence of a VSP "Done" event. |
| 29<br>swb | swb<br>Resume on VCPU software breakpoint<br>Controls whether VSPA will resume (if halted) on Software Breakpoint events.<br>0b - Disable VSPA resume on occurrence of software breakpoint.<br>1b - Enable VSPA resume on occurrence of software breakpoint. |
| 28-24<br>— | -<br>Reserved |
| 23-16 | ctinn |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| ctinn | Resume on VSPA cross trigger input n where n is 7-0<br>Controls whether VSPA will resume (if halted) on dbg_trig_in_req[n] events.<br>00000000b - Disable VSPA resume on occurrence of cross trigger in n.<br>00000001b - Enable VSPA resume on occurrence of cross trigger in n. |
| 15-8<br>— | -<br>Reserved |
| 7-0<br>cmpn | cmpn<br>Resume on VSPA comparator n match where n is 7-0<br>Controls whether VSPA will resume (if halted) on comparator n match events.<br>00000000b - Disable VSPA Resume on occurrence of comparator n match.<br>00000001b - Enable VSPA Resume on occurrence of comparator n match. |

### 14.1.8 Debug Program Trace Sync Action Control register (PTSACR)

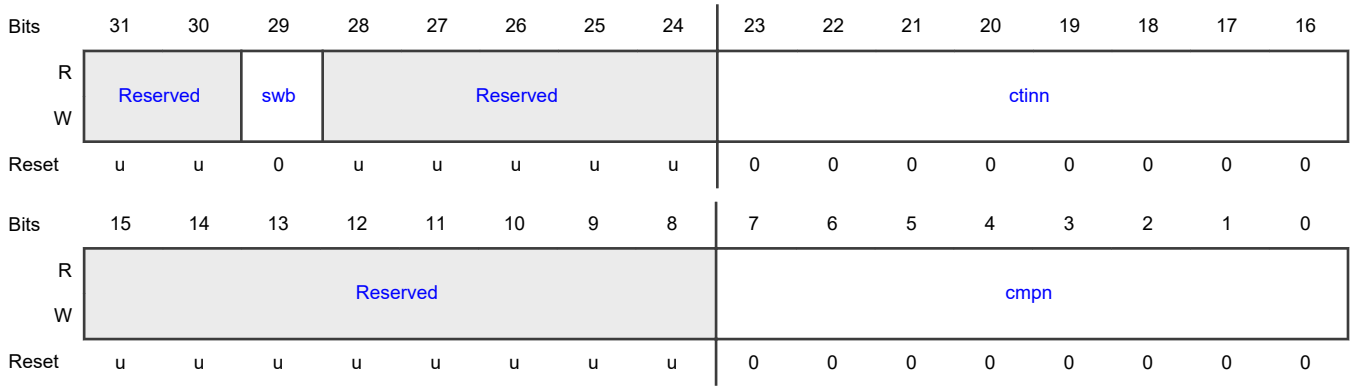
#### Offset

| Register | Offset |
|---|---|
| PTSACR | 844h |

#### Function

This 32-bit read-write register configures which events enable the action of generating a PTM Sync message. When any (or all) of the events enabled in this register occur, a program trace sync message is generated, providing the PC (program address) value with SYNC condition indicating external SYNC request (SYNC encoding 1110/1111). Note that the dedicated dbg_sync_req input is not controlled by this register. That is, on assertion of dbg_sync_req, a PTM sync message will be generated irrespective of the configuration of this register.

#### Diagram



**Fields**

| Field | Function |
|---|---|
| 31-30<br>— | -<br>Reserved |
| 29<br>swb | swb<br>Program trace sync generated on occurrence of software breakpoint event<br>PTM sync message generated on VCPU software breakpoint event.<br>0b - PTM sync message not generated on occurrence of software breakpoint event.<br>1b - PTM sync message generated on occurrence of software breakpoint event. |
| 28-24<br>— | -<br>Reserved |
| 23-16<br>ctinn | ctinn<br>Program trace sync generated on occurrence of cross-trigger input n where n is 7-0<br>PTM sync message generated upon trigger of dbg_trig_in_req[n].<br>00000000b - PTM sync message not generated on occurrence of dbg_trig_in_req[n] trigger.<br>00000001b - PTM sync message generated on occurrence of dbg_trig_in_req[n] trigger. |
| 15-8<br>— | -<br>Reserved |
| 7-0<br>cmpn | cmpn<br>Program trace sync generated on occurrence of comparator n match event where n is 7-0<br>PTM Sync Message generated upon occurrence of comparator n match event.<br>00000000b - PTM sync message not generated on occurrence of comparator n match event.<br>00000001b - PTM sync message generated on occurrence of comparator n match event. |

### 14.1.9 Debug Program Trace Start Action Control register (PSTACR)

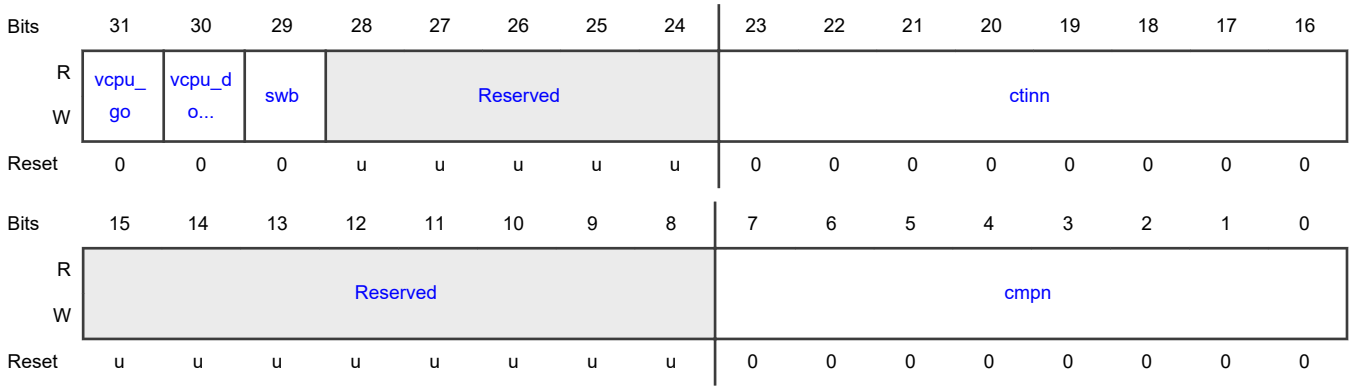
**Offset**

| Register | Offset |
|---|---|
| PSTACR | 848h |

**Function**

This 32-bit read-write register configures which events enable the action of triggering start Program Trace. Note that if the ptm_tr_start bit in TC1 register is set (program trace enabled), then the trigger action controlled by this register is masked, otherwise Program Trace is triggered on (start) at the occurrence of any event enabled by this register.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31<br>vcpu_go | vcpu_go<br>Program trace starts on VCPU "Go" event<br>Program trace started (triggered on) on occurrence of a VCPU "Go" event, if ptm_tr_start and ptm_tr_stop bits of TC1 register are not set.<br>0b - Program trace not started on occurrence of a VCPU "Go" event.<br>1b - Program trace started on occurrence of a VCPU "Go" event. |
| 30<br>vcpu_done | vcpu_done<br>Program trace start on execution of a VCPU "Done" instruction.<br>Program Trace started on a VCPU "Done" event.<br>0b - Program trace not started on occurrence of a VCPU "Done" event.<br>1b - Program trace started on occurrence of a VCPU "Done" event. |
| 29<br>swb | swb<br>Program trace start on occurrence of software breakpoint event.<br>Program Trace started (triggered on) on occurrence of Software Breakpoint event, if ptm_tr_start and ptm_tr_stop bits of TC1 register are not set.<br>0b - Program trace not started on occurrence of software breakpoint event.<br>1b - Program Trace started on occurrence of Software Breakpoint event. |
| 28-24<br>— | -<br>Reserved |
| 23-16<br>ctinn | ctinn<br>Program trace start on occurrence of cross-trigger input n where n is 7-0.<br>Program Trace started (triggered on) on occurrence of dbg_trig_in_req[n] event, if ptm_tr_start and ptm_tr_stop bits of TC1register are not set. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 00000000b - Program trace not started on occurrence of dbg_trig_in_req[n] event.<br>00000001b - Program trace started on occurrence of dbg_trig_in_req[n] event. |
| 15-8<br>— | -<br>Reserved |
| 7-0<br>cmpn | cmpn<br>Program trace start on occurrence of comparator n match event where n is 7-0.<br>Program trace started (triggered on) on occurrence of Comparator n match event, if ptm_tr_start and ptm_tr_stop bits of TC1 register are not set.<br>00000000b - Program trace not started on occurrence of comparator n match event.<br>00000001b - Program trace started on occurrence of comparator n match event. |

### 14.1.10 Debug Program Trace Stop Action Control register (PSPACR)

#### Offset

| Register | Offset |
|---|---|
| PSPACR | 84Ch |

#### Function

This 32-bit read-write register configures which events enable the action of triggering program trace stop. Note that if the ptm_tr_start bit in TC1 register is set (Program Trace enabled), then the trigger action controlled by this register is masked, otherwise program trace is triggered off (stop) at the occurrence of any event enabled by this register.

#### Diagram

| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R<br>W | vcpu_go | vcpu_do... | swb | Reserved | | | | | ctinn | | | | | | | |
| 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<br>W | Reserved | | | | | | | | cmpn | | | | | | | |
| Reset | u | u | u | u | u | u | u | u | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Fields**

| Field | Function |
|---|---|
| 31<br>vcpu_go | vcpu_go<br>Program trace stop on VCPU "Go" event<br>Program trace stopped (triggered off) on occurrence of a VCPU "Go" event, if ptm_tr_start and ptm_tr_stop bits of TC1 register are not set.<br>0b - Program trace not stopped on occurrence of a VCPU "Go" event.<br>1b - Program trace stopped on occurrence of a VCPU "Go" event. |
| 30<br>vcpu_done | vcpu_done<br>Program trace stop on execution of VCPU "Done" instruction<br>Program trace stopped (triggered off) on execution of a VCPU "Done" instruction, if ptm_tr_start and ptm_tr_stop bits of TC1 register are not set.<br>0b - Program trace not stopped on execution of a VCPU "Done" instruction.<br>1b - Program trace stopped on execution of a VCPU "Done" instruction. |
| 29<br>swb | swb<br>Program trace stop on occurrence of software breakpoint event.<br>Program trace stopped (triggered off) on occurrence of software breakpoint event, if ptm_tr_start and ptm_tr_stop bits of TC1 registers are not set.<br>0b - Program trace not stopped on occurrence of software breakpoint event.<br>1b - Program trace stopped on occurrence of software breakpoint event. |
| 28-24<br>— | -<br>Reserved |
| 23-16<br>ctinn | ctinn<br>Program trace stop on occurrence of cross-trigger input n where n is 7-0.<br>Program trace stopped (triggered off) on occurrence of dbg_trig_in_req[n] event, if ptm_tr_start and ptm_tr_stop bits of TC1 register are not set.<br>00000000b - Program trace not stopped on occurrence of dbg_trig_in_req[n] event.<br>00000001b - Program trace stopped on occurrence of dbg_trig_in_req[n] event. |
| 15-8<br>— | -<br>Reserved |
| 7-0<br>cmpn | cmpn<br>Program trace stop on occurrence of comparator n match event where n is 7-0.<br>Program trace stopped (triggered off) on occurrence of comparator n match event, if ptm_tr_start and ptm_tr_stop bits of TC1 registers not set.<br>00000000b - Program trace not stopped on occurrence of comparator n match event. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 00000001b - Program trace stopped on occurrence of comparator n match event. |

### 14.1.11 Debug Data Trace Start Action Control register (DSTACR)

#### Offset

| Register | Offset |
|---|---|
| DSTACR | 850h |

#### Function

This 32-bit read-write register configures which events enable the action of triggering start Data Trace. Note that if the dtm_tr_start bit in TC1 register is set (data trace enabled), then the trigger action controlled by this register is masked, otherwise data trace is triggered on (start) at the occurrence of any event enabled by this register.

#### Diagram

| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R | vcpu_go | vcpu_do... | swb | Reserved | | | | | ctinn | | | | | | | |
| W | | | | | | | | | | | | | | | | |
| 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 | Reserved | | | | | | | | cmpn | | | | | | | |
| W | | | | | | | | | | | | | | | | |
| Reset | u | u | u | u | u | u | u | u | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

#### Fields

| Field | Function |
|---|---|
| 31<br>vcpu_go | vcpu_go<br>Data trace start on VCPU "Go" event.<br>Data trace started (triggered on) on occurrence of a VCPU "Go" event, if dtm_tr_start and dtm_tr_stop bits of TC1 register are not set.<br>0b - Data trace not started on occurrence of a VCPU "Go" event.<br>1b - Data trace started on occurrence of a VCPU "Go" event. |
| 30<br>vcpu_done | vcpu_done<br>Data trace start on execution of a VCPU "Done" instruction |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | Data trace started on a VCPU "Done" event.<br>0b - Data trace not started on occurrence of a VCPU "Done" event.<br>1b - Data trace started on occurrence of a VCPU "Done" event. |
| 29<br>swb | swb<br>Data trace start on occurrence of software breakpoint event.<br>Data trace started (triggered on) on occurrence of software breakpoint event, if dtm_tr_start and dtm_tr_stop bits of TC1 register are not set.<br>0b - Data trace not started on occurrence of software breakpoint event.<br>1b - Data trace started on occurrence of software breakpoint event. |
| 28-24<br>— | -<br>Reserved |
| 23-16<br>ctinn | ctinn<br>Data trace start on occurrence of cross-trigger input n where n is 7-0.<br>Data trace started (triggered on) on occurrence of dbg_trig_in_req[n] event, if dtm_tr_start and dtm_tr_stop bits of TC1 register are not set.<br>00000000b - Data trace not started on occurrence of dbg_trig_in_req[n] event.<br>00000001b - Data trace started on occurrence of dbg_trig_in_req[n] event. |
| 15-8<br>— | -<br>Reserved |
| 7-0<br>cmpn | cmpn<br>Data Trace start on occurrence of comparator n match event where n is 7-0.<br>Data trace started (triggered on) on occurrence of comparator n match event, if dtm_tr_start and dtm_tr_stop bits of tc1 register are not set.<br>00000000b - Data trace not started on occurrence of comparator n match event.<br>00000001b - Data trace started on occurrence of comparator n match event. |

### 14.1.12 Debug Data Trace Stop Action Control register (DSPACR)

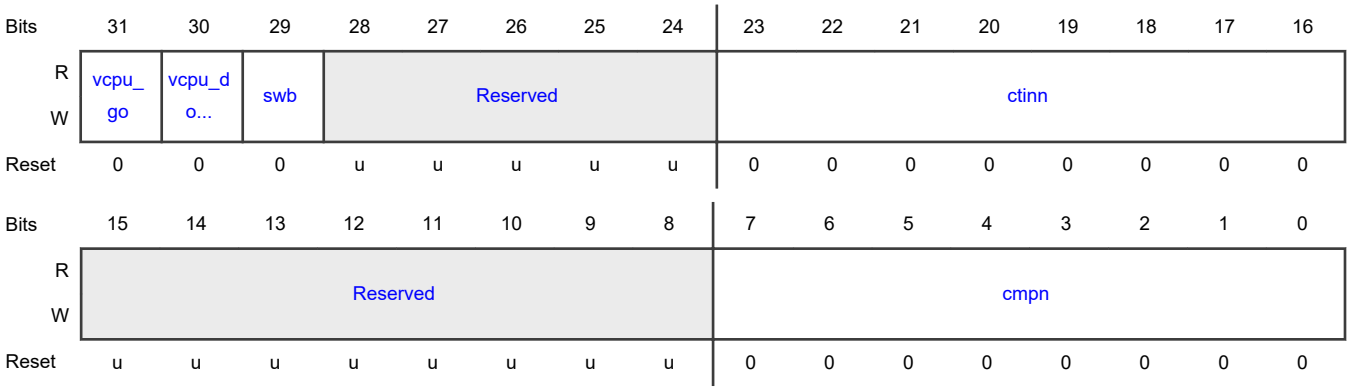
**Offset**

| Register | Offset |
|---|---|
| DSPACR | 854h |

### Function

This 32-bit read-write register configures which events enable the action of triggering data trace stop. Note that if the dtm_tr_start bit in TC1 register is set (data trace enabled), then the trigger action controlled by this register is masked, otherwise data trace is triggered off (stop) at the occurrence of any event enabled by this register.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31<br>vcpu_go | vcpu_go<br>Data trace stop on a VCPU "Go" event.<br>Data trace stopped (triggered off) on occurrence of a VCPU "Go" event, if dtm_tr_start and dtm_tr_stop bits of TC1 register are not set.<br>0b - Data trace not stopped on occurrence of a VCPU "Go" event.<br>1b - Data trace stopped on occurrence of a VCPU "Go" event. |
| 30<br>vcpu_done | vcpu_done<br>Data trace stop on execution of a VCPU "Done" instruction.<br>Data trace stopped (triggered off) on execution of a VCPU "Done" instruction, if dtm_tr_start and dtm_tr_stop bits of TC1 register are not set.<br>0b - Data trace not stopped on execution of a VCPU "Done" instruction.<br>1b - Data trace stopped on execution of a VCPU "Done" instruction. |
| 29<br>swb | swb<br>Data trace stop on occurrence of software breakpoint event.<br>Data trace stopped (triggered off) on occurrence of software breakpoint event, if dtm_tr_start and dtm_tr_stop bits of TC1 register are not set.<br>0b - Data Trace not stopped on occurrence of software breakpoint event.<br>1b - Data trace stopped on occurrence of software breakpoint event. |
| 28-24 | - |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| — | Reserved |
| 23-16<br>ctinn | ctinn<br>Data Trace stop on occurrence of cross-trigger input n where n is 7-0.<br>Data trace stopped (triggered off) on occurrence of dbg_trig_in_req[n] event, if dtm_tr_start and dtm_tr_stop bits of TC1 register are not set.<br>00000000b - Data trace not stopped on occurrence of dbg_trig_in_req[n] event.<br>00000001b - Data trace stopped on occurrence of dbg_trig_in_req[n] event. |
| 15-8<br>— | -<br>Reserved |
| 7-0<br>cmpn | cmpn<br>Data trace stop on occurrence of comparator n match event where n is 7-0<br>Data trace stopped (triggered off) on occurrence of comparator *n* match event, if dtm_tr_start and dtm_tr_stop bits of TC1 register are not set.<br>00000000b - Data Trace not stopped on occurrence of comparator n match event.<br>00000001b - Data trace stopped on occurrence of comparator n match event. |

### 14.1.13 Debug Data Trace Message Generation Action Control register (DTMACR)

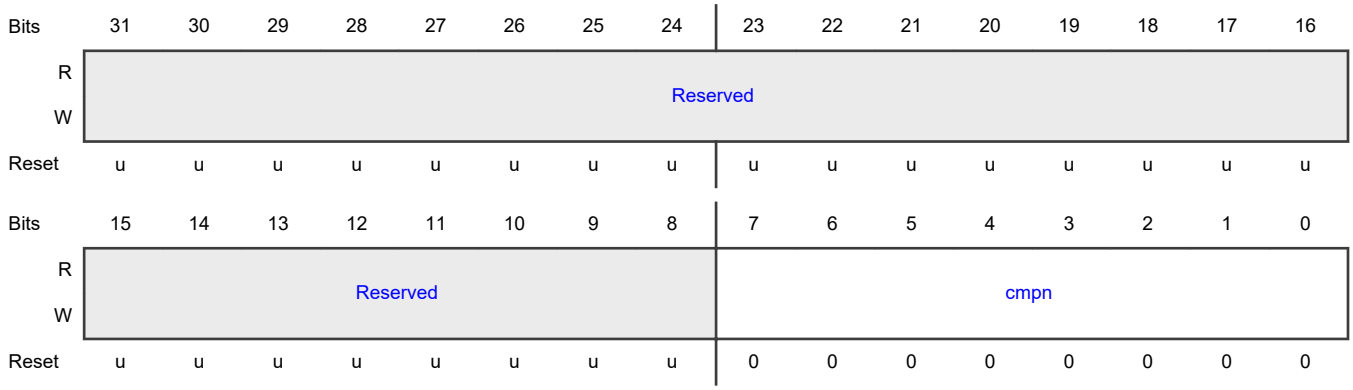
**Offset**

| Register | Offset |
|---|---|
| DTMACR | 858h |

**Function**

This 32-bit read-write register configures which events enable the action of generating a data trace message (DTM) to be generated. The comparators act as filters for the data trace. Note that DTM trace must be enabled in the TC1 register to enable data trace messaging or DTM start must be triggered via an appropriate event enabled in the DSTACR register.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-8<br>— | -<br>Reserved |
| 7-0<br>cmpn | cmpn<br>Data trace message generation on comparator n match event where n is 7-0<br>DTM generated on occurrence of comparator match, if DTM trace is enabled in the TC1 register or DSTACR.SWB bit is set to trigger data trace start.<br>00000000b - DTM message not generated on occurrence of associated comparator match.<br>00000001b - DTM message generated on occurrence of associated comparator match. |

## 14.1.14 Debug WPM Generation Action Control register (WPMACR)

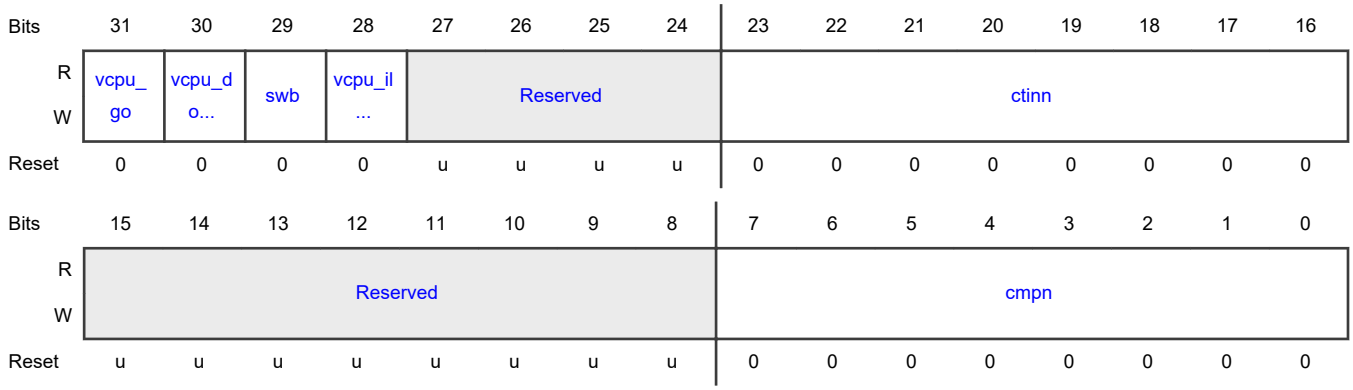
### Offset

| Register | Offset |
|---|---|
| WPMACR | 85Ch |

### Function

This 32-bit read-write register configures which events enable the action of generating a watchpoint message. Note that the wpt_en bit in the TC1 register must be set to enable watchpoint messages.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31<br>vcpu_go | vcpu_go<br>Watchpoint message generation for watchpoint payload bit 18<br>WPM generated on a VCPU "Go" event.<br>0b - Watchpoint message not generated on occurrence of a VCPU "Go" event.<br>1b - Watchpoint message generated on occurrence of a VCPU "Go" event. |
| 30<br>vcpu_done | vcpu_done<br>Watchpoint message generation for watchpoint payload bit 17.<br>WPM generated on a VCPU "Done" event.<br>0b - Watchpoint message not generated on execution of a VCPU "Done" instruction.<br>1b - Watchpoint message generated on execution of a VCPU "Done" instruction. |
| 29<br>swb | swb<br>Watchpoint message generation for watchpoint payload bit 16.<br>WPM generated on software breakpoint event.<br>0b - Watchpoint message not generated on occurrence of software breakpoint event.<br>1b - Watchpoint message generated on occurrence of software breakpoint event. |
| 28<br>vcpu_illop | vcpu_illop<br>Watchpoint message generation for watchpoint payload bit 19.<br>WPM is generated on a VCPU illegal instruction trap event.<br>0b - Watchpoint message not generated on execution of a VCPU illegal instruction trap event.<br>1b - Watchpoint message generated on execution of a VCPU illegal instruction trap event. |
| 27-24<br>— | -<br>Reserved |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 23-16<br>ctinn | ctinn<br>Watchpoint message generated for watchpoint payload bits 15-8.<br>WPM generated on dbg_trig_in_req[n] event where n is 7-0.<br>00000000b - Watchpoint message not generated on occurrence of dbg_trig_in_req[n] event.<br>00000001b - Watchpoint message generated on occurrence of dbg_trig_in_req[n] event. |
| 15-8<br>— | -<br>Reserved |
| 7-0<br>cmpn | cmpn<br>Watchpoint message generated for watchpoint payload bit 7-0.<br>WPM generated on comparator n match event where n is 7-0.<br>00000000b - Watchpoint message not generated on occurrence of comparator m match event.<br>00000001b - Watchpoint message generated on occurrence of comparator m match event. |

### 14.1.15 Debug TSM Control register (TSMCR)

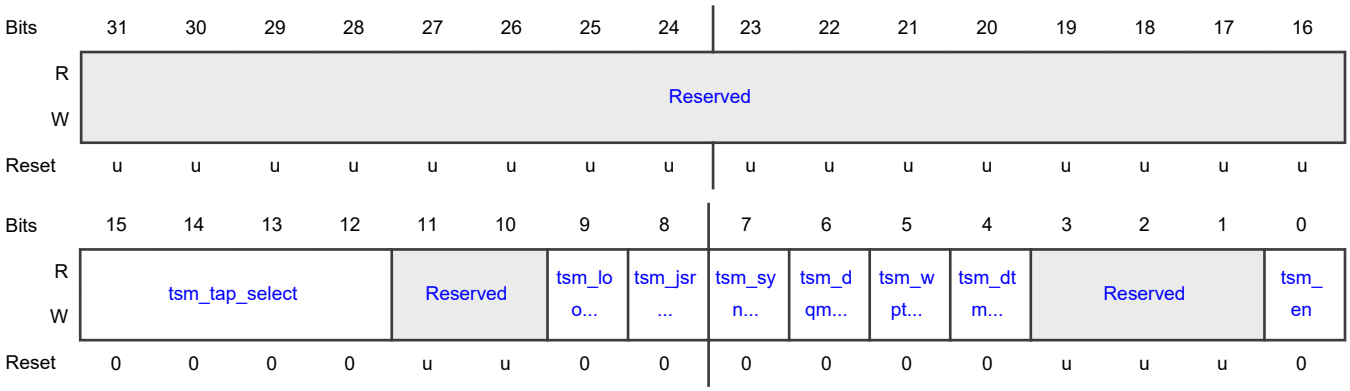
#### Offset

| Register | Offset |
|---|---|
| TSMCR | 860h |

#### Function

This 32-bit read-write register configures which events enable the action of generating a timestamp message. Note that the wpt_en bit in the TC1 register must be set to enable watchpoint messages.

#### Diagram



**Fields**

| Field | Function |
|---|---|
| 31-16<br>— | -<br>Reserved |
| 15-12<br>tsm_tap_select | tsm_tap_select<br>Timestamp tap point select.<br>Selects the tap point which will cause a timestamp rollover message to be generated.<br>0000b - TSM rollover is generated when timestamp bit 24 changes.<br>0001b - TSM rollover is generated when timestamp bit 22 changes.<br>0010b - TSM rollover is generated when timestamp bit 20 changes.<br>0011b - TSM rollover is generated when timestamp bit 18 changes.<br>0100b - TSM rollover is generated when timestamp bit 16 changes.<br>0101b - TSM rollover is generated when timestamp bit 14 changes.<br>0110b - TSM rollover is generated when timestamp bit 12 changes.<br>0111b - TSM rollover is generated when timestamp bit 10 changes.<br>1000-1111b - Reserved. |
| 11-10<br>— | -<br>Reserved |
| 9<br>tsm_loop_en | tsm_loop_en<br>Timestamp message generation for looping (loop start and loop end).<br>Controls whether a TSM message is generated when a VSP starts or ends a loop.<br>0b - Start/End loop do NOT cause a TSM message to be sent.<br>1b - Start/End loop causes a TSM message to be sent. |
| 8<br>tsm_jsr_en | tsm_jsr_en<br>Timestamp message generation for JSR / RTS (jump to and return from subroutine).<br>Controls whether a TSM message is generated when a VSP executes a taken JSR or RST instruction.<br>0b - Executing a JSR or RTS does NOT cause a TSM message to be sent.<br>1b - Executing a JSR or RTS causes a TSM message to be sent. |
| 7<br>tsm_sync_en | tsm_sync_en<br>Timestamp message generation for SYNC Messages.<br>Controls whether a TSM message is generated when a SYNC message is generated.<br>0b - a SYNC messages do not cause a TSM message to be sent.<br>1b - a SYNC messages cause a preceding TSM message to be sent. TS bit in all SYNC messages will be set. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 6<br>tsm_dqm_en | tsm_dqm_en<br>Timestamp message generation for DQM (Data Acquisition Messages).<br>Controls whether a TSM message is generated when a DQM message is generated.<br>0b - a WPT messages does not cause a TSM message to be sent.<br>1b - a WPT messages causes a preceding TSM message to be sent. TS bit in all WPT messages will be set. |
| 5<br>tsm_wpt_en | tsm_wpt_en<br>Timestamp message generation for WPT (Watchpoint Trace Messages).<br>Controls whether a TSM message is generated when a WPT message is generated. |
| 4<br>tsm_dtm_en | tsm_dtm_en<br>Timestamp message generation for DTM (Data Trace Messages).<br>Controls whether a TSM message is generated when a DTM message is generated.<br>0b - a DTM messages does not cause a TSM message to be sent.<br>1b - a DTM messages causes a preceding TSM message to be sent. TS bit in all DTM messages will be set. |
| 3-1<br>— | -<br>Reserved |
| 0<br>tsm_en | tsm_en<br>Timestamp message generation enable<br>Control timestamp messages are disabled or enabled.<br>0b - Timestamp is off/disabled. No timestamp messages are sent.<br>1b - Timestamp id enabled. Timestamp messages are sent when an event occur and the corresponding enable bit is set. |

### 14.1.16 Debug VSPA Timestamp Counter (Upper) register (TSMCNT_U)

**Offset**

| Register | Offset |
|---|---|
| TSMCNT_U | 864h |

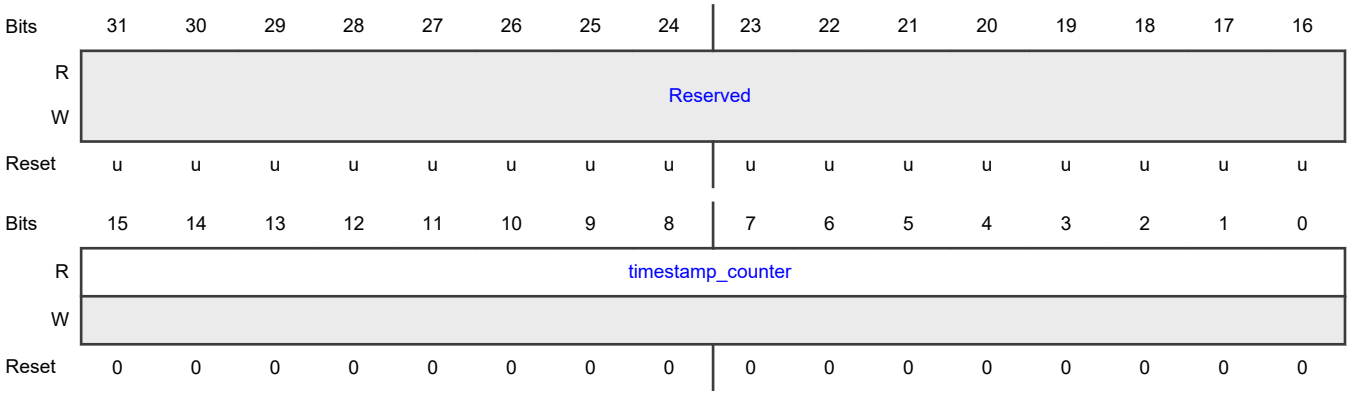
**Function**

TSMCNT_U register provides the upper bits of the 48-bit Timestamp.

The VSPA timestamp message uses a 48-bit timestamp which is provided by a centralized timestamp generator on the SoC. The timestamp can be coherently sampled and read by the debugger by performing a read access to the TSMCNT_U (upper bits) register first followed by a read of the TSMCNT_L (lower bits) register.

When reading the TSMCNT_U register, the value of the lower 32 bits of the timestamp input is simultaneously captured in a timestamp_shadow register. When reading the TSMCNT_L register, the value read is the value captured in the timestamp_shadow register.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-16<br>— | -<br>Reserved |
| 15-0<br>timestamp_counter | timestamp_counter<br>Timestamp counter. Provides a portion of a 48-bit Timestamp value from a centralized timestamp generator on the SoC. |

## 14.1.17 Debug VSPA Timestamp Counter (Lower) register (TSMCNT_L)

### Offset

| Register | Offset |
|---|---|
| TSMCNT_L | 868h |

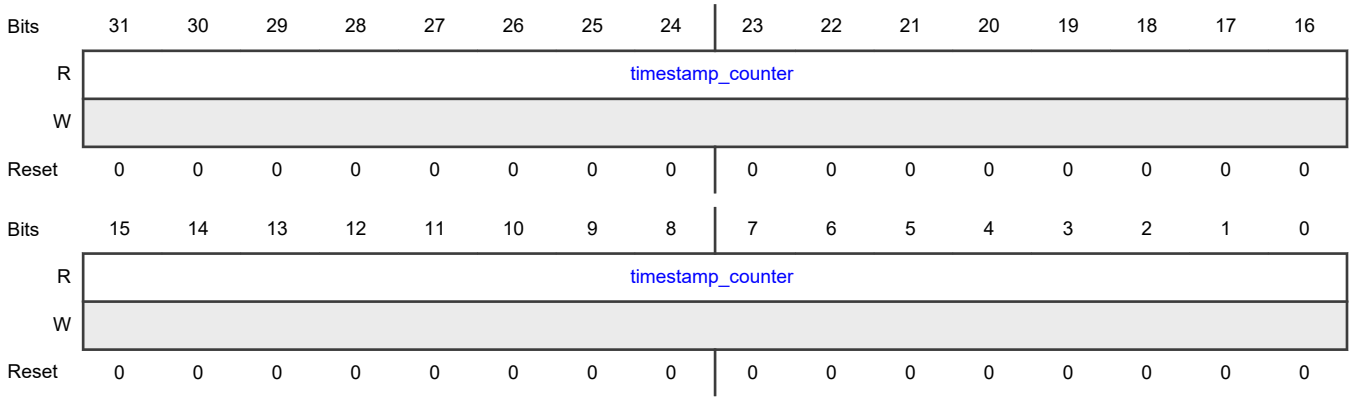
### Function

TSMCNT_L register provides the lower 32 bits of the 48-bit Timestamp input.

The VSPA timestamp message uses a 48-bit timestamp which is provided by a centralized timestamp generator on the SoC. The timestamp can be coherently sampled and read by the debugger by performing a read access to the TSMCNT_U (upper bits) register first followed by a read of the TSMCNT_L (lower bits) register.

When reading the TSMCNT_U register, the value of the lower 32 bits of the timestamp input is simultaneously captured in a timestamp_shadow register. When reading the TSMCNT_L register, the value read is the value captured in the timestamp_shadow register.

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-0<br>timestamp_counter | timestamp_counter<br>Timestamp counter. Provides 32-bit Timestamp value from a centralized timestamp generator on the SoC. |

## 14.1.18 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 122 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 122. VSP/IPPU Architectural Access Modes**

| a_mode[27:24] | Description |
|---|---|
| 0 | VSP Architectural Register list. Refer to Table 123.<br>**NOTE**<br>Data is LSB justified. |
| 1 | VSP DMEM Memory.<br>a_index addresses 32-bit words<br>Note return data format:<br>data[31:0] =DMEM[a_index]. |
| 2 | IPPU DMEM Memory.<br>a_index addresses 32-bit words<br>Note return data format:<br>data[31:0] =DMEM[a_index]. |
| 3 | VSP PRAM Memory.<br>a_index addresses 32-bit words.<br>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 |

*Table continues on the next page...*

**Table 122. VSP/IPPU Architectural Access Modes (continued)**

| a_mode[27:24] | Description |
|---|---|
| | 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.<br>a_index addresses 32-bit words.<br>Note return data format:<br>data[31:0] =IPPU PRAM [a_index]. |
| 6 | VSP Register File (RF) R0-R7 registers.<br>a_index addresses a 32-bit word starting with least significant element at R0.<br>Note return data format:<br>data[31:0] =Rn[a_index] (where n = a_index/32) |
| 7 | H Register. |
| 8 | IPPU Architectural Register list. Refer to Table 124.<br>**NOTE**<br>Data is LSB justified. |
| 9 | IPPU Register File (RF) R0-R1-R2 registers.<br>a_index addresses a 32-bit word starting with least significant element at R0.<br>Note: return data format:<br>data[31:0] =Rn[a_index] (where n = a_index/32)<br>0 <= a_index < 3*32 |
| 10 | Reserved |

**Table 123. 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. |
| 14 | MAG hardware stack pointer (SP) value. |
| 15-18 | MAG general purpose address registers a0 - a3. |
| 19-20 | MAG a2.min, MAG a2.max. |

*Table continues on the next page...*

**Table 123. VSP Architectural Register list (a_mode = 0) (continued)**

| a_index[18:2] | VSP Architectural Register |
|---|---|
| 21-22 | MAG a3.min, MAG a3.max. |
| 23-25 | Reserved, always reads 0. |
| 26-41 | MAG address storage registers as0 - as15. |
| 42 | MAG bit reversal, flip mode register (flip_mode).<br>Note return data format:<br>data[31:0] ={29'b0,flip_mode[2:0]}. |
| 43 | CREG bits: AuOut (soft bits, hard bits, threshold bits).<br>Note return data format:<br>data[31:0] ={30'b0, auout[1:0]}. |
| 44 | CREG bits (cc_update, previous cc bits, current cc bits).<br>Note return data format:<br>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.<br>Note return data format:<br>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).<br>Note return data format:<br>data[31:0] ={16'b0, S2mode[3:0], S1mode[3:0], 2'b0, S0conj, S0chs, S0mode[3:0]}. |
| 89 | RF rot_mode register.<br>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). |

*Table continues on the next page...*

**Table 123. VSP Architectural Register list (a_mode = 0) (continued)**

| a_index[18:2] | VSP Architectural Register |
|---|---|
| 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 |
| 114 | Reserved |
| 115 | RAG au ptr register. |
| 116 | RAG au incr register. |

*Table continues on the next page...*

**Table 123. VSP Architectural Register list (a_mode = 0) (continued)**

| a_index[18:2] | VSP Architectural 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 124. IPPU Architectural Register list (a_mode = 8)**

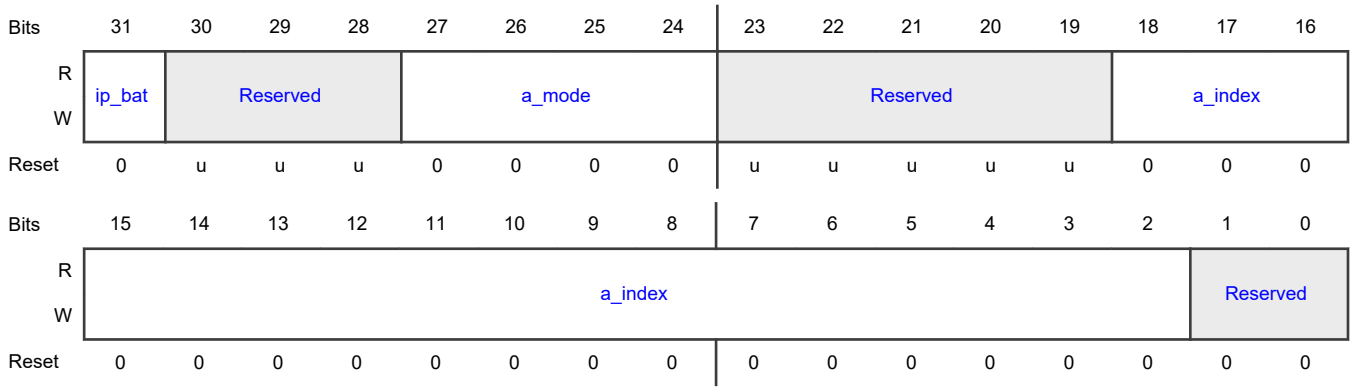
| a_index[18:2] | VSP Architectural Register |
|---|---|
| 0 | Reserved |
| 1 | a0 - MAG general purpose address registers a0. |
| 2 | a1 - MAG general purpose address registers a1. |
| 3 | as0 - MAG address storage register as0. |
| 4 | as1 - MAG address storage register as1. |
| 5 | as2 - MAG address 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. |
| 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. |

*Table continues on the next page...*

**Table 124. IPPU Architectural Register list (a_mode = 8) (continued)**

| a_index[18:2] | VSP Architectural Register |
|---|---|
| 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 |
| 25 | a2 - MAG general purpose address registers a2 |
| 26 | a3 - MAG general purpose address registers a3 |

### Diagram




### Fields

| Field | Function |
|---|---|
| 31<br>ip_bat | ip_bat<br>VSP IPbus gateway - IP bridge access type. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | This bit specifies which identity should be used for the access to the VSP IPbus (host or VSP).<br>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.<br>0b - Access is executed as if it were from the host.<br>1b - Access is executed as if it were from VSP. |
| 30-28<br>— | -<br>Reserved |
| 27-24<br>a_mode | a_mode<br>Run control visibility - access mode.<br>These bits select between the VSP internal memories and the architectural registers.<br>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.<br>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.<br>0000b - VSP architectural registers mode - registers selected via the a_index field.<br>0001b - VSP DMEM mode.<br>0010b - VSP IPPU DMEM mode.<br>0011b - VSP PRAM mode.<br>0100b - Reserved.<br>0101b - VSP IPPU PRAM mode.<br>0110b - VSP RF register mode (R0-R7).<br>0111b - VSP H register mode.<br>1000b - IPPU architectural registers mode - registers selected via the a_index field.<br>1001b - IPPU RF register mode (R0-R1).<br>1010b - Reserved. |
| 23-19<br>— | -<br>Reserved |
| 18-2<br>a_index | a_index<br>Run control visibility - access index.<br>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", |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | restricting access to full 32-bit words only. Refer to Table 123 for the index value definitions for VSP register access. Refer to Table 124 for the index value definitions for IPPU register access.<br>Settings: Index value for a given data object, depending on mode selected by a_mode field. |
| 1-0<br>— | -<br>Reserved |

### 14.1.19 Debug VSP Architecture Visibility Fixed Data register (RAVFD)

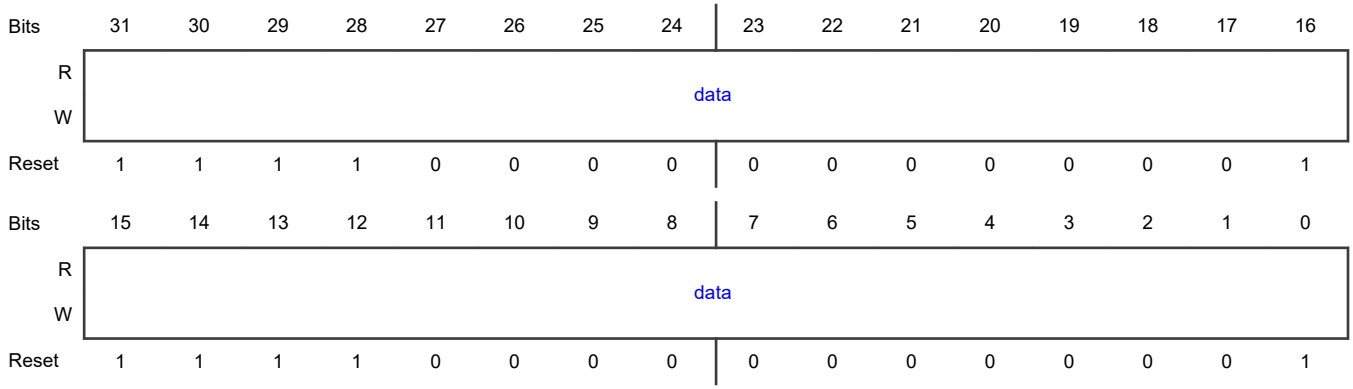
#### Offset

| 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




#### Fields

| Field | Function |
|---|---|
| 31-0<br>data | data<br>Run Control Visibility - Fixed Addressing Mode Data Register.<br>Data to be written or read (depending on access type to this register) to the data object at address configured by the RAVAP register. |

*Table continues on the next page...*

| Field | Function |
|---|---|
| | Note accesses to this register will NOT cause the RAVAP to auto-increment.<br>Settings: Data to read/write, depending on configuration of RAVAP register. |

### 14.1.20 Debug VSP Architecture Visibility Incrementing Data register (RAVID)

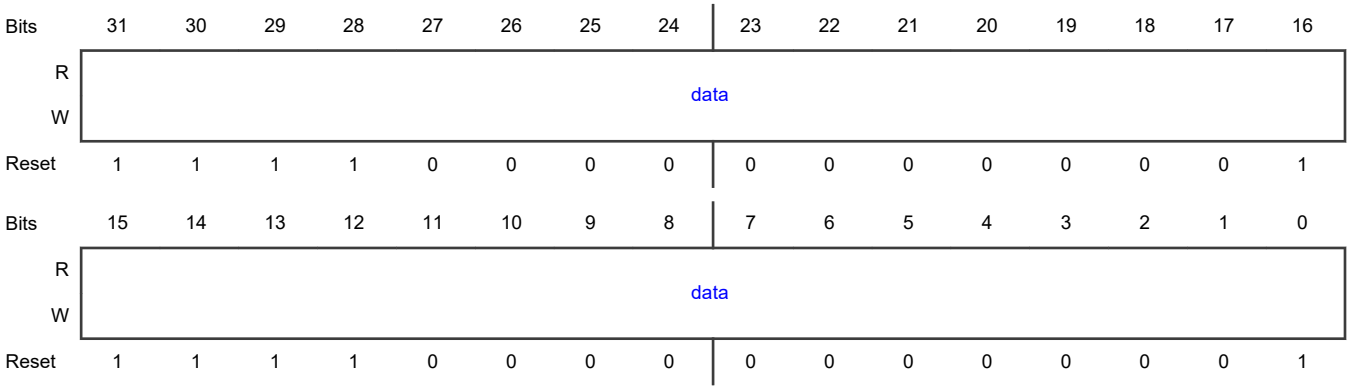
#### Offset

| 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<br>data | data<br>Run control visibility - Incrementing addressing mode data register.<br>Data to be written or read (depending on access type to this register) to the data object at address configured by the RAVAP register.<br>Note accesses to this register will cause the RAVAP to auto-increment.<br>Settings: Data to read/write, depending on configuration of RAVAP register. |

### 14.1.21 Debug Verification register (DVR)

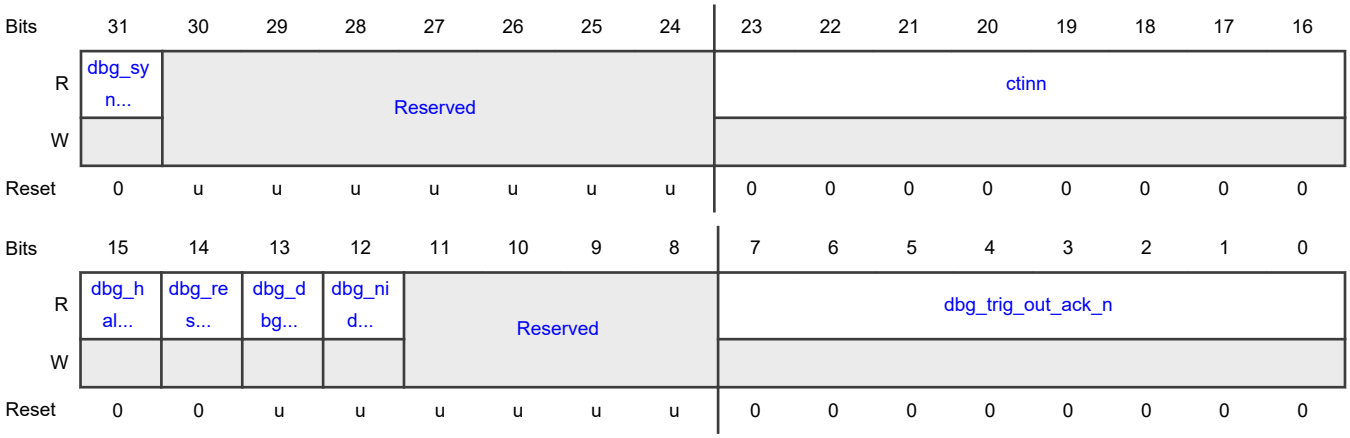
#### Offset

| 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<br>dbg_sync_req | dbg_sync_req<br>Verification - Value driven on debug sync request pin.<br>This bit reflects the value driven on the dbg_sync_req debug input pin for integration verification purposes.<br>0b - dbg_sync_req input driven low.<br>1b - dbg_sync_req input driven high. |
| 30-24<br>— | -<br>Reserved |
| 23-16<br>ctinn | ctinn<br>Verification - Value driven on cross-trigger in request input n pin where n is 7-0.<br>This bit reflects the value driven on the dbg_trig_in_req[n] input pin for integration verification purposes.<br>00000000b - dbg_trig_in_req[n] input driven low.<br>00000001b - dbg_trig_in_req[n] input driven high. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 15<br>dbg_halt_req | dbg_halt_req<br>Verification - Value driven on dbg_halt_req pin.<br>This bit reflects the value driven on the dbg_halt_r*eq* input pin for integration verification purposes.<br>0b - dbg_halt_req input driven low.<br>1b - dbg_halt_req input driven high. |
| 14<br>dbg_resume_req | dbg_resume_req<br>Verification - Value driven on dbg_resume_req pin.<br>This bit reflects the value driven on the dbg_resume_req input pin for integration verification purposes.<br>0b - dbg_resume_req input driven low.<br>1b - dbg_resume_req input driven high. |
| 13<br>dbg_dbgen | dbg_dbgen<br>Verification - Value driven on dbg_dbgen pin<br>This bit reflects the value driven on the dbg_dbgen input pin for integration verification purposes.<br>0b - dbg_dbgen input driven low.<br>1b - dbg_dbgen input driven high. |
| 12<br>dbg_niden | dbg_niden<br>Verification - Value driven on dbg_niden pin<br>This bit reflects the value driven on the dbg_niden input pin for integration verification purposes.<br>0b - dbg_niden input driven low.<br>1b - dbg_niden input driven high. |
| 11-8<br>— | -<br>Reserved |
| 7-0<br>dbg_trig_out_ack_n | dbg_trig_out_ack_n<br>Verification - Value driven on Cross-Trigger Out Ack input n pin where n is 7-0.<br>This bit reflects the value driven on the dbg_trig_out_ack[n] input pin for integration verification purposes.<br>00000000b - dbg_trig_out_ack[n] input driven low.<br>00000001b - dbg_trig_out_ack[n] input driven high. |

### 14.1.22 Debug Cross Trigger Out a Action Control registers (CTO0ACR - CTO7ACR)

**Offset**

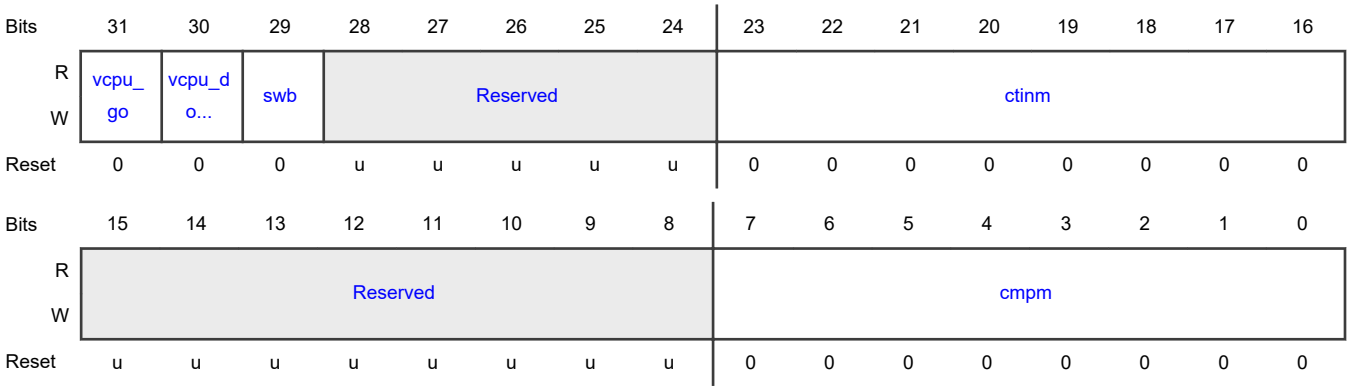
For a = 0 to 7:

| Register | Offset |
|---|---|
| CTOaACR | 880h + (a × 4h) |

### Function

These 32-bit read-write registers configure which events enable the action of triggering cross-trigger out n, where n is 0 -7 (*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<br>vcpu_go | vcpu_go<br>Enable Cross-Trigger Out n triggered on a VCPU "Go" event<br>*dbg_trig_out[n]*triggered high on occurrence of a VCPU "Go" event.<br>Disabled when CTOnACR = 0x0000000.<br>0b - *dbg_trig_out[n]* NOT generated on occurrence of a VCPU "Go" event.<br>1b - *dbg_trig_out[n]* generated on occurrence of a VCPU "Go" event. |
| 30<br>vcpu_done | vcpu_done<br>Enable Cross-Trigger Out n triggered on execution of a VCPU "Done" instruction<br>*dbg_trig_out[n]* triggered high on execution of a VCPU "Done" instruction.<br>Disabled when CTOnACR = 0x0000000.<br>0b - *dbg_trig_out[n]* NOT generated on execution of a VCPU "Done" instruction.<br>1b - *dbg_trig_out[n]* generated on execution of a VCPU "Done" instruction. |
| 29<br>swb | swb<br>Enable Cross-Trigger Out n triggered on Software Breakpoint event<br>*dbg_trig_out[n]* triggered high on occurrence of Software Breakpoint event. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | Disabled when CTOnACR = 0x0000000.<br>0b - *dbg_trig_out[n]* NOT generated on occurrence of Software Breakpoint event.<br>1b - *dbg_trig_out[n]* generated on occurrence of Software Breakpoint event. |
| 28-24<br>— | -<br>Reserved |
| 23-16<br>ctinm | ctinm<br>Enable Cross-Trigger Out n triggered on occurrence of Cross-Trigger In m event where m is 7-0<br>dbg_trig_out[m] triggered high on occurrence of event dbg_trig_in_req[m].<br>Disabled when CTOnACR = 0x0000000.<br>00000000b - dbg_trig_out[n] NOT generated on occurrence of dbg_trig_in_req[m] event.<br>00000001b - dbg_trig_out[n] generated on occurrence of dbg_trig_in_req[m] event. |
| 15-8<br>— | -<br>Reserved |
| 7-0<br>cmpm | cmpm<br>Enable Cross-Trigger Out n triggered on occurrence of Comparator m match event where m is 7-0<br>dbg_trig_out[n] triggered high on occurrence of Comparator m match event.<br>Disabled when CTOnACR = 0x0000000.<br>00000000b - *dbg_trig_out[n]* NOT generated on occurrence of Comparator m match event.<br>00000001b - *dbg_trig_out[n]* generated on occurrence of Comparator m match event. |

### 14.1.23 Debug Comparator Control and Status register (DC0CS - DC7CS)

**Offset**

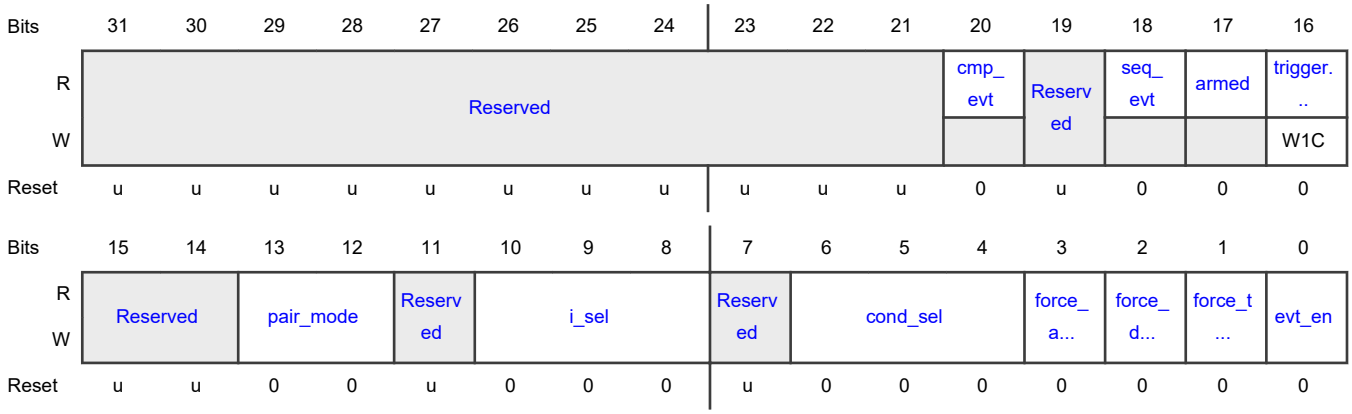
For a = 0 to 7:

| 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<br>— | -<br>Reserved |
| 20<br>cmp_evt | cmp_evt<br>Compare state status bit.<br>This read-only status bit indicates the current state of the comparator output signal *cmp_evt*.<br>0b - The comparator output signal *cmp_evt* is cleared.<br>1b - The comparator output signal *cmp_evt* is asserted. |
| 19<br>— | -<br>Reserved |
| 18<br>seq_evt | seq_evt<br>Comparator sequential trigger status bit.<br>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.<br>0b - The comparator is not triggered.<br>1b - The comparator is triggered. |
| 17<br>armed | armed<br>Comparator armed status bit.<br>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:<br>1) the force_arm bit in the corresponding Debug Comparator Control and Status Register (DCnCS) was set |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 2) the corresponding bit in the ARM Action Control Register (CnAACR) is set, and the corresponding activation event was active<br>0b - Comparator is not armed.<br>1b - Comparator is armed. |
| 16<br>triggered | triggered<br>Comparator triggered status bit.<br>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.<br>0b - Comparator was not triggered since this bit was last cleared (or reset).<br>1b - Comparator was triggered one or more times since this bit was last cleared (or reset). |
| 15-14<br>— | -<br>Reserved |
| 13-12<br>pair_mode | pair_mode<br>Comparator pair mode configuration bits.<br>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.<br>Only certain sets of comparators can be paired:<br>The legal combinations are 0 and 1, 2 and 3, 4 and 5, 6 and 7.<br>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.<br>00b - *comp_out* is the condition selected by *cond_sel* field.<br>01b - *comp_out* is the condition selected by *cond_sel* field AND *cmp_in* selected by *i_sel* field.<br>10b - *comp_out* is the condition selected by *cond_sel* field OR *cmp_in* selected by *i_sel* field.<br>11b - Reserved. |
| 11<br>— | -<br>Reserved |
| 10-8<br>i_sel | i_sel<br>Comparator input select configuration bits.<br>These bits are used to select the inputs (address bus) for in_data values.<br>Note - When the comparator is used to generate DTM trace messages, i_sel must be set to '001 - Select VSP DMEM'.<br>000b - Select none (constant zero). |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 001b - Select VSP DMEM.<br>010b - Select IPPU DMEM.<br>011b - Select VSP PMEM.<br>100b - Select VSP Peripheral Bus -IPbus.<br>101b - Select IPPU PMEM.<br>110-111b - Reserved. |
| 7<br>— | -<br>Reserved |
| 6-4<br>cond_sel | cond_sel<br>Comparator condition selects configuration bits.<br>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.<br>000b - Select none (comparator off).<br>001b - *data_in == data_reg*.<br>010b - *data_in != data_reg*.<br>011b - *data_in > data_reg*.<br>100b - *data_in >= data_reg*.<br>101b - *data_in < data_reg*.<br>110b - *data_in <= data_reg*.<br>111b - Reserved. |
| 3<br>force_arm | force_arm<br>Force comparator arm bit.<br>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.<br>0b - Do not force the ARMED register to one.<br>1b - Force the ARMED register to one. |
| 2<br>force_disarm | force_disarm<br>Force comparator disarm bit.<br>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.<br>0b - Do NOT force the ARMED register to zero.<br>1b - Force the ARMED register to zero. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 1<br>force_trig | force_trig<br>Force comparator trigger bit.<br>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.<br>0b - Do not force the comparator output signal *cmp_evt*.<br>1b - Assert (drive to 1) the comparator output signal *cmp_evt*. |
| 0<br>evt_en | evt_en<br>Comparator event enable bit.<br>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.<br>0b - Comparator is disabled. The comparator output *cmp_evt* is de-asserted and driven to 0.<br>1b - Comparator is enabled. A comparator event may be generated and output *cmp_evt* asserted when conditions defined by this comparator are met. |

### 14.1.24 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 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 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

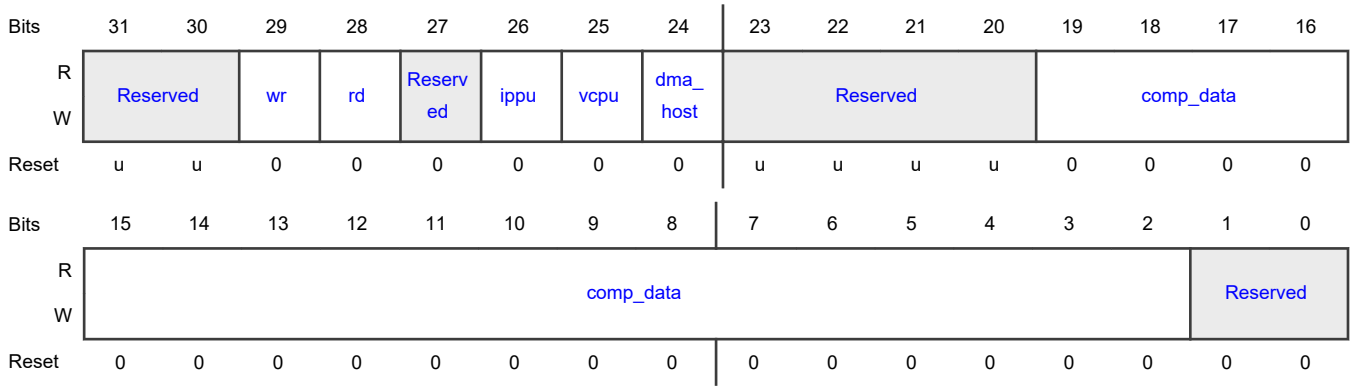
d. The selected input is VSP Peripheral Bus AND

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 {IPPU, VSP, dma_host}

### Diagram




### Fields

| Field | Function |
|---|---|
| 31-30<br>— | -<br>Reserved |
| 29<br>wr | wr<br>Write access attribute bit.<br>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.<br>0b - Do NOT enable match when a write access occurs.<br>1b - Enable match when a write access occurs. |
| 28<br>rd | rd<br>Read access attribute bit.<br>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.<br>0b - Do NOT enable match when a read access occurs. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 1b - Enable match when a read access occurs. |
| 27<br>— | -<br>Reserved |
| 26<br>ippu | ippu<br>IPPU access attribute bit.<br>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.<br>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.<br>0b - Do not enable match when an IPPU access occurs.<br>1b - Enable match when an IPPU access occurs. |
| 25<br>vcpu | vcpu<br>VCPU access attribute bit.<br>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.<br>0b - Do not enable match when a VCPU engine access occurs.<br>1b - Enable match when a VCPU engine access occurs. |
| 24<br>dma_host | dma_host<br>DMA or host attribute bit.<br>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.<br>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.<br>It matches a DMA access to DMEM or IPPU DMEM. It matches a host access to the VSP IP Bus.<br>0b - Do not enable match when a DMA or host access occurs.<br>1b - Enable match when a DMA or host access occurs. |
| 23-20<br>— | -<br>Reserved |
| 19-2<br>comp_data | comp_data<br>Compare data value.<br>The comp_data field contains 18-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.<br>17-bit compare data value. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 1-0<br>— | -<br>Reserved |

### 14.1.25 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

| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R<br>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<br>W | Reserved | | | | | | | | cmpm | | | | | | | |
| Reset | u | u | u | u | u | u | u | u | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

#### Fields

| Field | Function |
|---|---|
| 31<br>vcpu_go | vcpu_go<br>Enable arm of comparator n on a VCPU "Go" event<br>Select whether comparator n will be armed as a result of a VCPU "Go" event.<br>Note that comparator n must first be enabled prior to arming.<br>0b - Comparator n not armed on occurrence of a VCPU "Go" event.<br>1b - Comparator n armed on occurrence of a VCPU "Go" event. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| 30<br>vcpu_done | vcpu_done<br>Enable arm of comparator n on execution of a VCPU "Done" instruction<br>Select whether comparator n will be armed as a result of the execution of a VCPU "Done" instruction.<br>Note that comparator n must first be enabled prior to arming.<br>0b - Comparator n not armed on execution of a VCPU "Done" instruction.<br>1b - Comparator n armed on execution of a VCPU "Done" instruction. |
| 29<br>swb | swb<br>Enable arm of comparator n on software breakpoint event<br>Select whether comparator n will be armed as a result of the execution of an instruction tagged with a SWB.<br>Note that comparator n must first be enabled prior to arming.<br>0b - Comparator n not armed on occurrence of Software Breakpoint event.<br>1b - Comparator n armed on occurrence of Software Breakpoint event. |
| 28-24<br>— | -<br>Reserved |
| 23-16<br>ctinm | ctinm<br>Enable arming of comparator n on occurrence of cross-trigger in m event where m is 7-0.<br>Select whether comparator n will be armed as a result of the detection of a dbg_trig_in_req[m] event.<br>Note that comparator n must first be enabled prior to arming.<br>00000000b - Comparator n not armed on occurrence of *dbg_trig_in_req[m]* event.<br>00000001b - Comparator n armed on occurrence of *dbg_trig_in_req[m]* event. |
| 15-8<br>— | -<br>Reserved |
| 7-0<br>cmpm | cmpm<br>Enable arming of comparator n on occurrence of comparator m match event where m is 7-0<br>Select whether comparator n will be armed as a result of the detection of a comparator m match event.<br>Note that comparator n must first be enabled prior to arming.<br>00000000b - Comparator n not armed on occurrence of Comparator m match event.<br>00000001b - Comparator n armed on occurrence of Comparator m match event. |

### 14.1.26 Debug Comparator a Disarm Action Control registers (C0DACR - C7DACR)

**Offset**

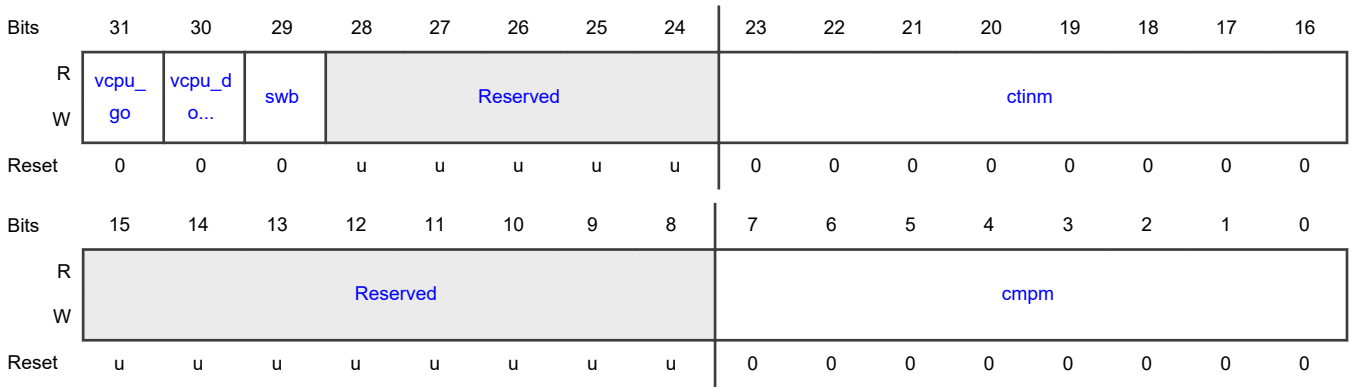
For a = 0 to 7:

| 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<br>vcpu_go | vcpu_go<br>Enable Disarm of Comparator n on a VCPU "Go" event<br>Select whether comparator n will be disarmed as a result of a VCPU "Go" event.<br>0b - Comparator n not disarmed on occurrence of a VCPU "Go" event.<br>1b - Comparator n disarmed on occurrence of a VCPU "Go" event. |
| 30<br>vcpu_done | vcpu_done<br>Enable Disarm of Comparator n on execution of a VCPU "Done" instruction<br>Select whether comparator n will be disarmed as a result of the execution of a VCPU "Done" instruction.<br>0b - Comparator n not disarmed on execution of a VCPU "Done" instruction.<br>1b - Comparator n disarmed on execution of a VCPU "Done" instruction. |
| 29<br>swb | swb<br>Enable Disarm of Comparator n on Software Breakpoint event<br>Select whether comparator n will be disarmed as a result of the execution of an instruction tagged with a SWB. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Field | Function |
|---|---|
| | 0b - Comparator n not disarmed on occurrence of Software Breakpoint event.<br>1b - Comparator n disarmed on occurrence of Software Breakpoint event. |
| 28-24<br>— | -<br>Reserved |
| 23-16<br>ctinm | ctinm<br>Enable disarming of comparator n on occurrence of cross-trigger in m event where m is 8-0<br>Select whether comparator n will be disarmed as a result of the detection of a dbg_trig_in_req[m] event.<br>00000000b - Comparator n not disarmed on occurrence of *dbg_trig_in_req[m]* event.<br>00000001b - Comparator n disarmed on occurrence of *dbg_trig_in_req[m]* event. |
| 15-8<br>— | -<br>Reserved |
| 7-0<br>cmpm | cmpm<br>Enable disarming of comparator n on occurrence of comparator m match event where m is 7-0<br>Select whether comparator n will be disarmed as a result of the detection of a Comparator m match event.<br>00000000b - Comparator n not disarmed on occurrence of Comparator m match event.<br>00000001b - Comparator n disarmed on occurrence of Comparator m match event. |

### 14.1.27 Debug Comparator a Trigger Action Control registers (C0TACR - C7TACR)

#### Offset

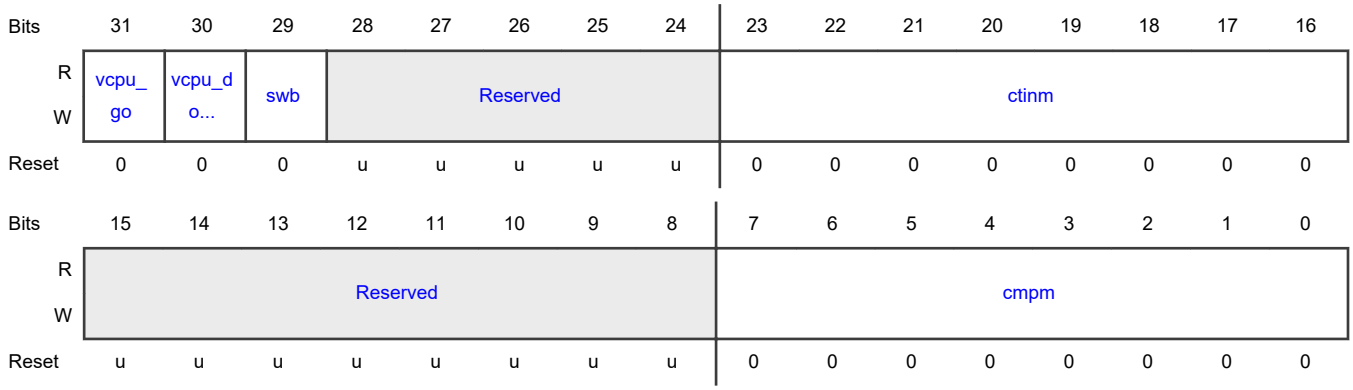
For a = 0 to 7:

| 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




### Fields

| Field | Function |
|---|---|
| 31<br>vcpu_go | vcpu_go<br>Enable Trigger of Comparator n on a VCPU "Go" event<br>Select whether comparator n will be triggered as a result of a VCPU "Go" event.<br>0b - Comparator n not triggered on occurrence of a VCPU "Go" event.<br>1b - Comparator n triggered on occurrence of a VCPU "Go" event. |
| 30<br>vcpu_done | vcpu_done<br>Enable Trigger of Comparator n on execution of a VCPU "Done" instruction<br>Select whether comparator n will be triggered as a result of the execution of a VCPU "Done" instruction.<br>0b - Comparator n not triggered on execution of a VCPU "Done" instruction.<br>1b - Comparator n triggered on execution of a VCPU "Done" instruction. |
| 29<br>swb | swb<br>Enable Trigger of Comparator n on Software Breakpoint event<br>Select whether comparator n will be triggered as a result of the execution of an instruction tagged with a SWB.<br>0b - Comparator n not triggered on occurrence of Software Breakpoint event.<br>1b - Comparator n triggered on occurrence of Software Breakpoint event. |
| 28-24<br>— | -<br>Reserved |
| 23-16<br>ctinm | ctinm<br>Enable triggering of comparator n on occurrence of cross-trigger in m event where m is 7-0<br>Select whether comparator n will be triggered as a result of the detection of a dbg_trig_in_req[m] event.<br>00000000b - Comparator n not triggered 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 triggered on occurrence of *dbg_trig_in_req[m]* event. |
| 15-8<br>— | -<br>Reserved |
| 7-0<br>cmpm | cmpm<br>Enable triggering of comparator n on occurrence of comparator m match event where m is 7-0<br>Select whether comparator n will be triggered as a result of the detection of a Comparator m match event.<br>00000000b - Comparator n not triggered on occurrence of Comparator m match event.<br>00000001b - Comparator n triggered on occurrence of Comparator m match event. |

## 14.1.28 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

| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R | | | | | | | | | | | | | | | | |
| W | debug_outbox32 | | | | | | | | | | | | | | | |
| 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 | debug_outbox32 | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

### Fields

| Field | Function |
|---|---|
| 31-0 | debug_outbox32 |

*Table continues on the next page...*

| Field | Function |
|---|---|
| debug_outbox32 | Debug to VSP 32-bit Outbox<br>Writes cause the value written to be sent to VSP.<br>This register is write only - reads always return 0's.<br>Data value to be sent to VSP inbox. |

### 14.1.29 Debug to VSP 64-bit MSB Outbox register (OUT_64_MSB)

#### Offset

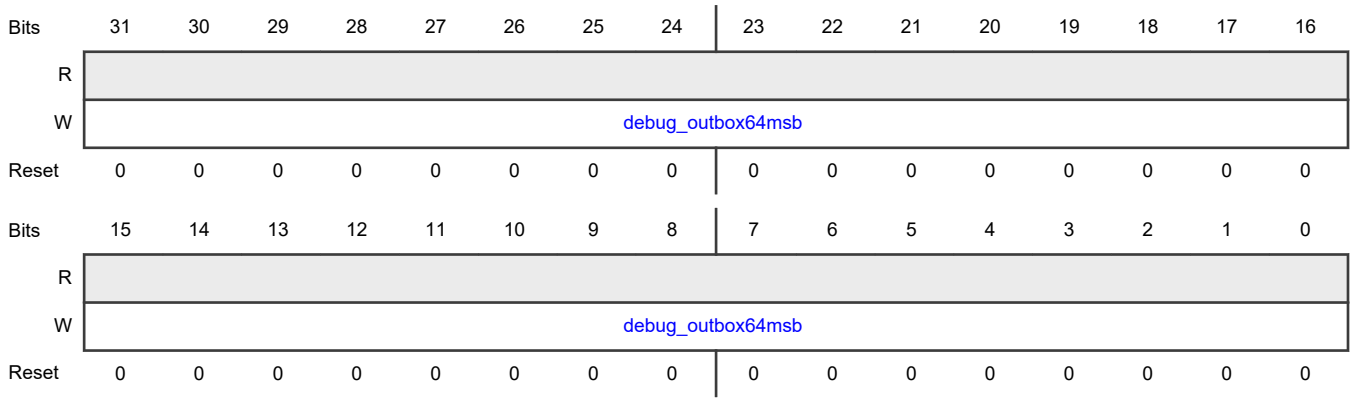
| Register | Offset |
|---|---|
| OUT_64_MSB | E24h |

#### 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<br>debug_outbox64msb | debug_outbox64msb<br>Debug 64-bit output data (MSB)<br>Writes update the 32 MSBs of a pending 64-bit message that will be sent to the VSP's 64-bit inbox.<br>The mail message will be sent in its entirety only after a write to the DBG_OUT_64_LSB register.<br>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. |

*Table continues on the next page...*

| Field | Function |
|---|---|
| | This register is write only - reads always return 0's.<br>Settings: MSB 32-bit data value to be sent to VSP's 64-bit inbox. |

### 14.1.30 Debug to VSP 64-bit LSB Outbox register (OUT_64_LSB)

#### Offset

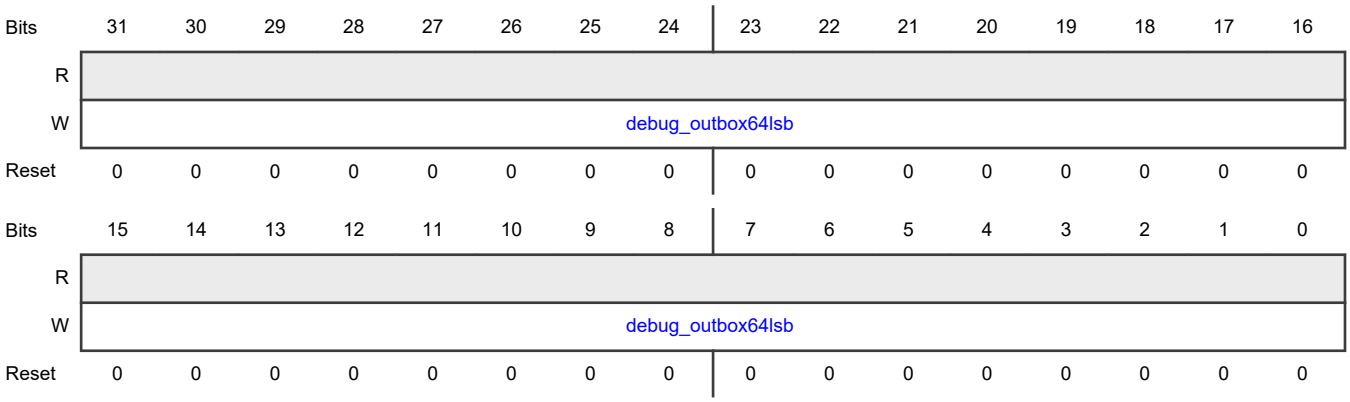
| Register | Offset |
|---|---|
| OUT_64_LSB | E28h |

#### 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<br>debug_outbox64lsb | debug_outbox64lsb<br>Debug 64-bit output data (LSB)<br>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.<br>This register is write only - reads always return 0's.<br>Settings: LSB 32-bit data value to be sent to VSP's 64-bit inbox. |

## 14.1.31 VSP to Debugger 32-bit Inbox register (IN_32)

### Offset

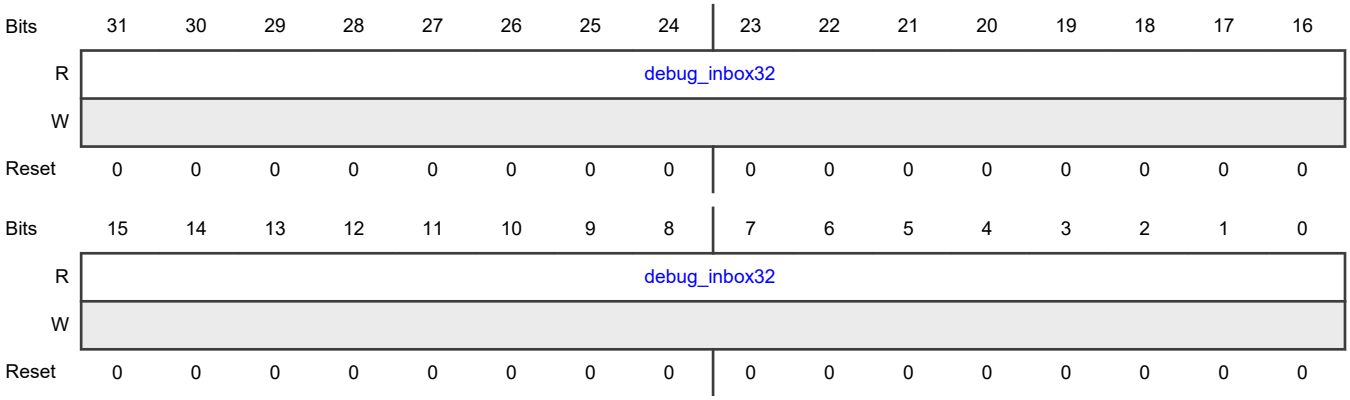
| Register | Offset |
|---|---|
| IN_32 | E2Ch |

### 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<br>debug_inbox32 | debug_inbox32<br>VSP to Debugger 32-bit Inbox<br>Reads the 32-bit mail message sent by VSP.<br>This register is read only - writes have no effect.<br>Settings: Reads 32-bit data value sent by VSP outbox. |

## 14.1.32 VSP to Debugger 64-bit MSB Inbox register (IN_64_MSB)

### Offset

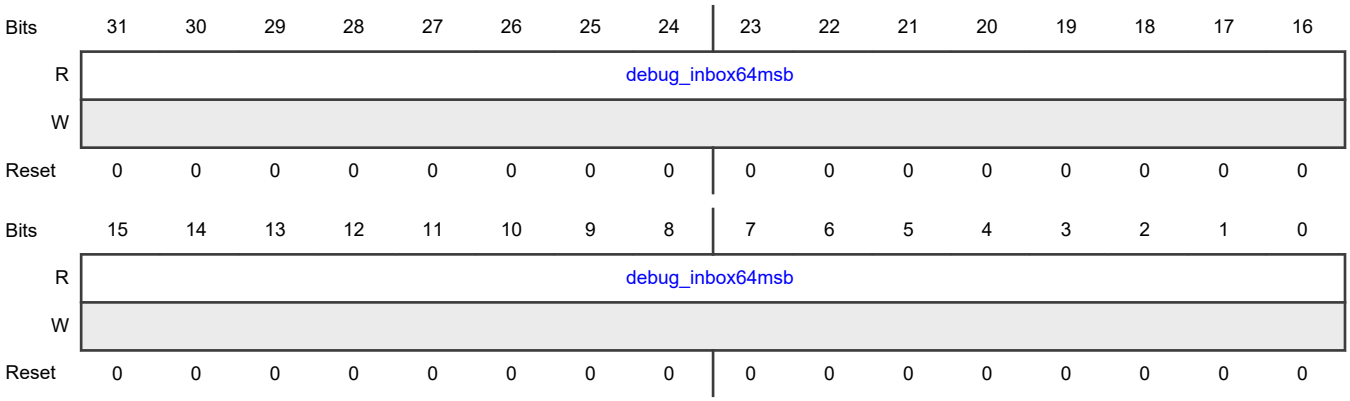
| Register | Offset |
|---|---|
| IN_64_MSB | E30h |

### 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<br>debug_inbox64msb | debug_inbox64msb<br>Debug 64-bit input data (MSB)<br>Reads the 32 MSBs of the 64-bit mail message sent by VSP.<br>This register is read only - writes have no effect.<br>Settings: Reads return 32 MSBs of the 64-bit data value sent via VSP's 64-bit outbox. |

## 14.1.33 VSP to Debugger 64-bit LSB Inbox register (IN_64_LSB)

### Offset

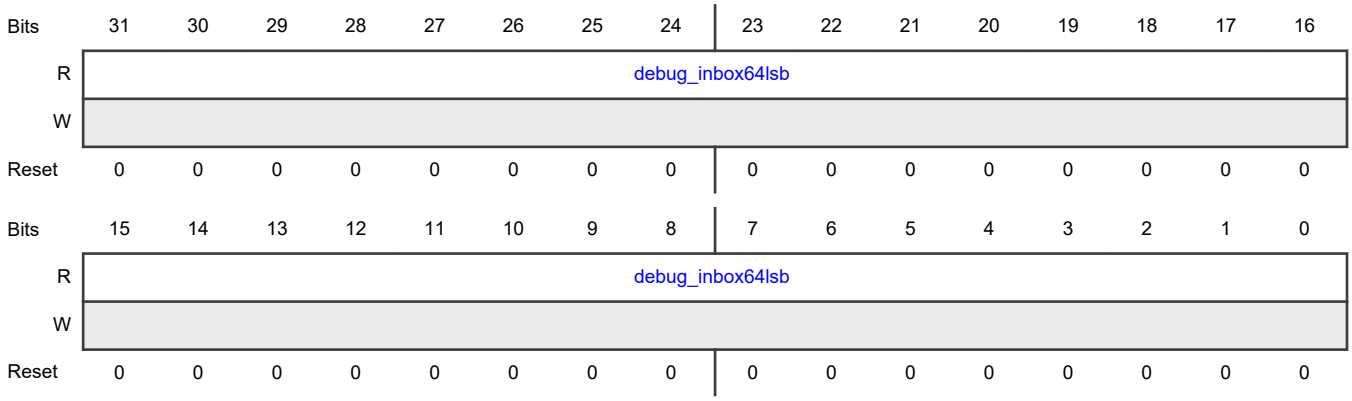
| Register | Offset |
|---|---|
| IN_64_LSB | E34h |

### 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<br>debug_inbox64lsb | debug_inbox64lsb<br>Debug 64-bit input data (LSB)<br>Reads the 32 LSBs of the 64-bit mail message sent by VSP.<br>This register is read only - writes have no effect.<br>Settings: Reads return 32 LSBs of the 64-bit data value sent via VSP's 64-bit outbox. |

## 14.1.34 Debugger to VSP Mailbox Status register (MBOX_STATUS)

### Offset

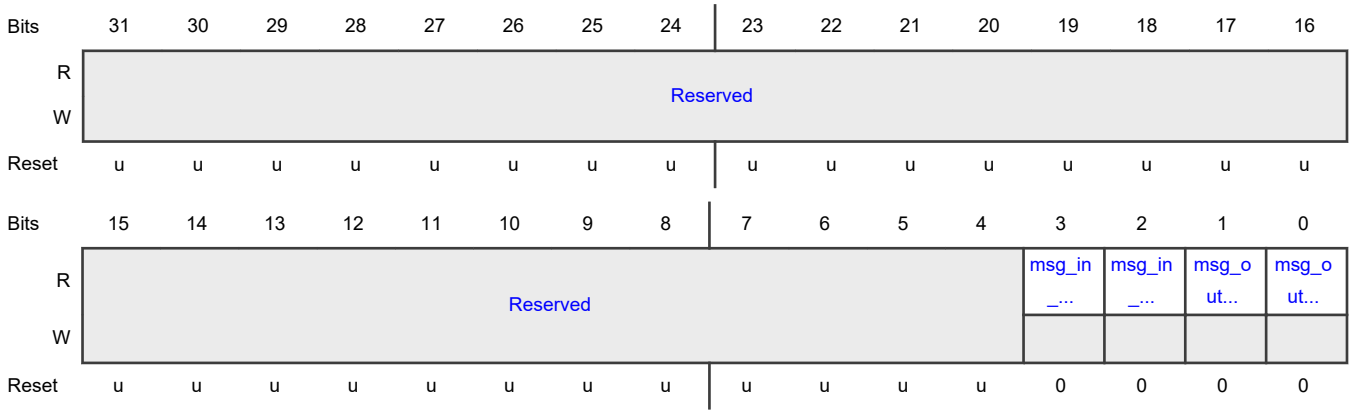
| Register | Offset |
|---|---|
| MBOX_STATUS | E38h |

### 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<br>— | -<br>Reserved |
| 3<br>msg_in_valid_64bit | msg_in_valid_64bit<br>64-bit message inbox valid<br>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).<br>0b - Data in the 64-bit inbox registers is NOT valid.<br>1b - Data in the 64-bit inbox registers is valid. |
| 2<br>msg_in_valid_32bit | msg_in_valid_32bit<br>32-bit message inbox valid<br>Shows the validity/invalidity of the data in the debugger 32-bit message inbox register (VSP to Debugger 32-bit Inbox register).<br>0b - Data in the 32-bit inbox register is NOT valid.<br>1b - Data in the 32-bit inbox register is valid. |
| 1<br>msg_out_valid_64bit | msg_out_valid_64bit<br>64-bit message outbox valid<br>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).<br>0b - No unread data is in the 64-bit outbox registers.<br>1b - Data in the 64-bit outbox registers has not yet been read by VSP. |
| 0<br>msg_out_valid_32bit | msg_out_valid_32bit<br>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).<br>0b - No unread data is in the 32-bit outbox register.<br>1b - Data in the 32-bit outbox register has not yet been read by VSP. |

### 14.1.35 Debug Parameter 0 Register (PARAM_0)

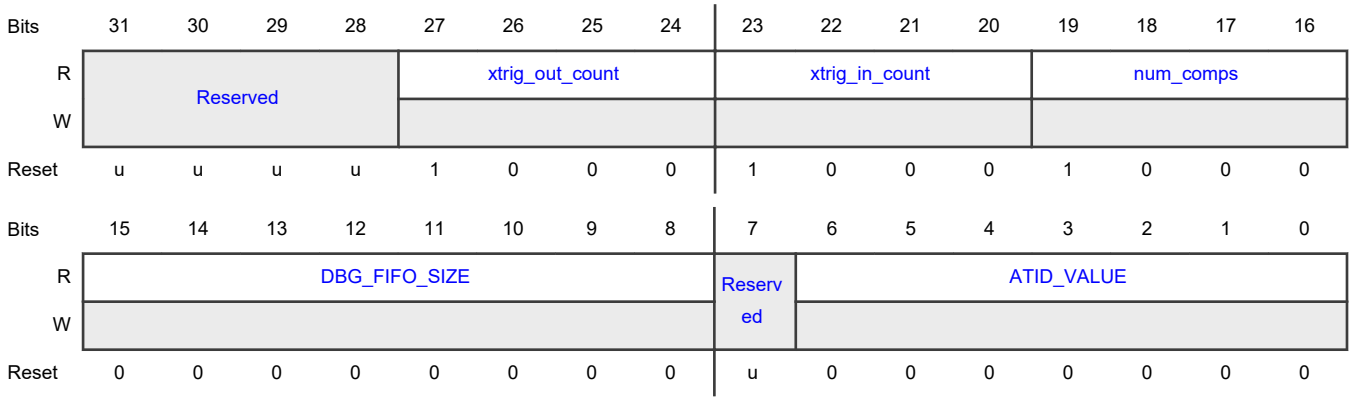
#### Offset

| Register | Offset |
|---|---|
| PARAM_0 | F00h |

#### Function

Debug Parameter 0 Register

#### Diagram




#### Fields

| Field | Function |
|---|---|
| 31-28<br>— | -<br>Reserved |
| 27-24<br>xtrig_out_count | xtrig_out_count<br>Cross trigger output count.<br>These status bits indicate the number of cross trigger outputs supported by this instance of VSPA.<br>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<br>xtrig_in_count | xtrig_in_count<br>Cross trigger input count.<br>These status bits indicate the number of cross trigger inputs supported by this instance of VSPA.<br>Settings: Reads return the number of cross trigger inputs supported by this instance of VSPA. Writes have no affect. |
| 19-16<br>num_comps | num_comps<br>Number of debug comparators (comparator count).<br>Debug comparator count.These status bits indicate the number of debug comparators supported by this instance of VSPA.<br>Settings: Reads return the number of debug comparators supported by this instance of VSPA. Writes have no affect. |
| 15-8<br>DBG_FIFO_SIZE | DBG_FIFO_SIZE<br>Number of entries in the Debug FIFO (debug FIFO size).<br>These status bits indicate the number of entries in the debug FIFO in this instance of VSPA.<br>Settings: Reads return the Number of entries in the Debug FIFO in this instance of VSPA. Writes have no affect.<br>00000000b - 0 entries<br>00000101b - 32 entries<br>00000110b - 64 entries<br>11111111b - Illegal |
| 7<br>— | -<br>Reserved |
| 6-0<br>ATID_VALUE | ATID_VALUE<br>These status bits indicate the value this VSPA places on the ATID lines when a trace message is sent on the ATB.<br>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. |

### 14.1.36 Peripheral ID4 register (PIDR4)

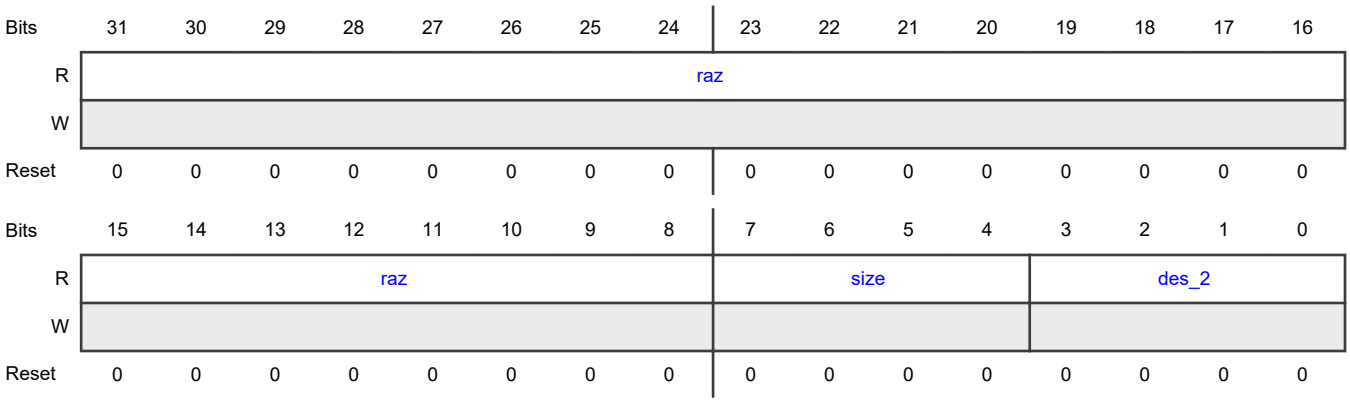
**Offset**

| 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<br>raz | raz<br>Reads As Zero |
| 7-4<br>size | size<br>4KByte Count Value<br>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)<br>VSP 4KByte Count = 0x0 = 4KB |
| 3-0<br>des_2 | des_2<br>JEP106 Continuation Code<br>DES_2, DES_1, DES_0 indicates the designer of the component.<br>DES_2 is the JTAG106 Continuation Code (for NXP, DES_2=0x0) |

## 14.1.37 Peripheral ID5 register (PIDR5)

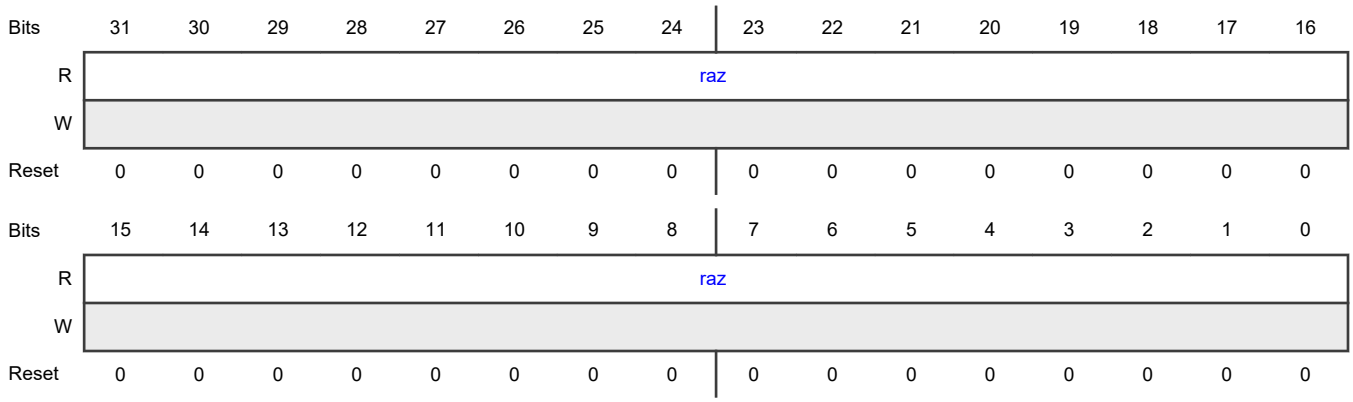
### Offset

| 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<br>raz | raz<br>Reads As Zero |

## 14.1.38 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

| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R | raz | | | | | | | | | | | | | | | |
| 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 | raz | | | | | | | | | | | | | | | |
| W | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

### Fields

| Field | Function |
|---|---|
| 31-0<br>raz | raz<br>Reads As Zero |

## 14.1.39 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

| Bits | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R | raz | | | | | | | | | | | | | | | |
| 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 | raz | | | | | | | | | | | | | | | |
| W | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

### Fields

| Field | Function |
|---|---|
| 31-0<br>raz | raz<br>Reads As Zero |

## 14.1.40 Peripheral ID0 register (PIDR0)

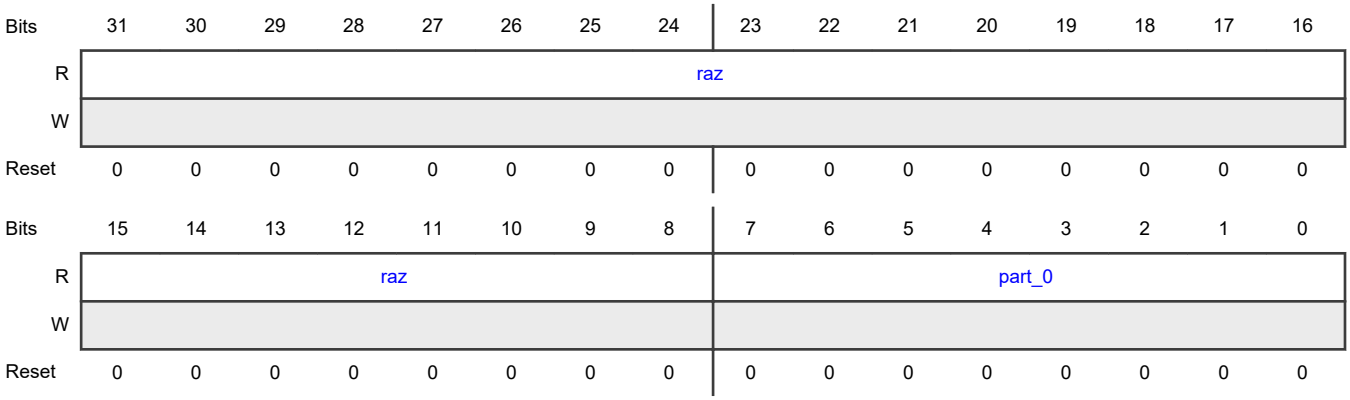
### Offset

| 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<br>raz | raz<br>Reads As Zero |
| 7-0<br>part_0 | part_0<br>Part Number bits[7:0]<br>{PART_1, PART_0} = PART[11:0] - indicates the component part number selected by the designer.<br>PART[11:0] = 0x000 indicates a VSP. |

### 14.1.41 Peripheral ID1 register (PIDR1)

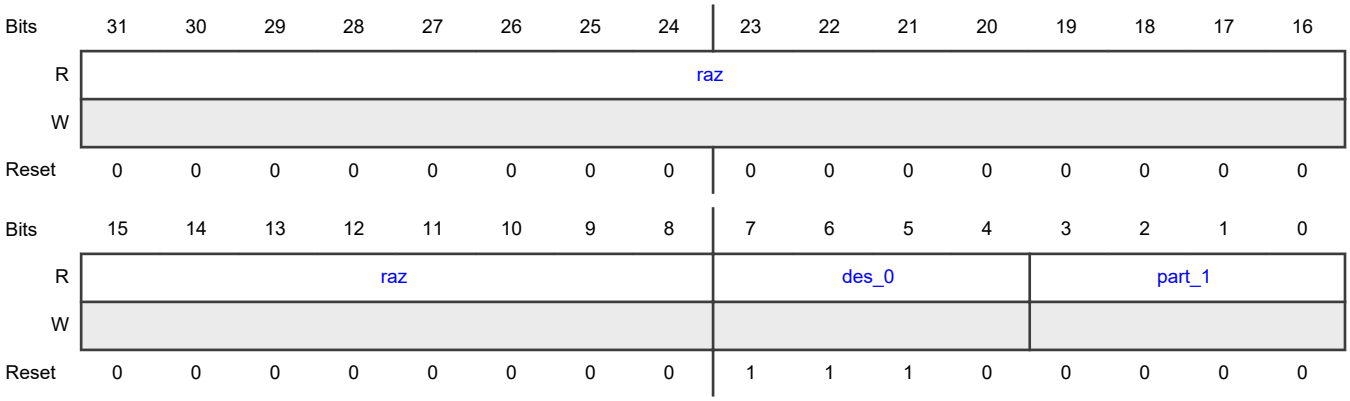
#### Offset

| 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<br>raz | raz<br>Reads As Zero |
| 7-4<br>des_0 | des_0<br>JEP106 Identification Code bits[3:0]<br>The concatenated {DES_2, DES_1, DES_0} indicates the designer of the component.<br>DES_0 are the upper bits of the JEP106 identification code (for NXP, DES_0=0xE) |
| 3-0<br>part_1 | part_1<br>Part Number bits[11:8]<br>{PART_1, PART_0} = PART[11:0] - indicates the component part number selected by the designer.<br>PART[11:0] == 0x0 indicates a VSP. |

### 14.1.42 Peripheral ID2 register (PIDR2)

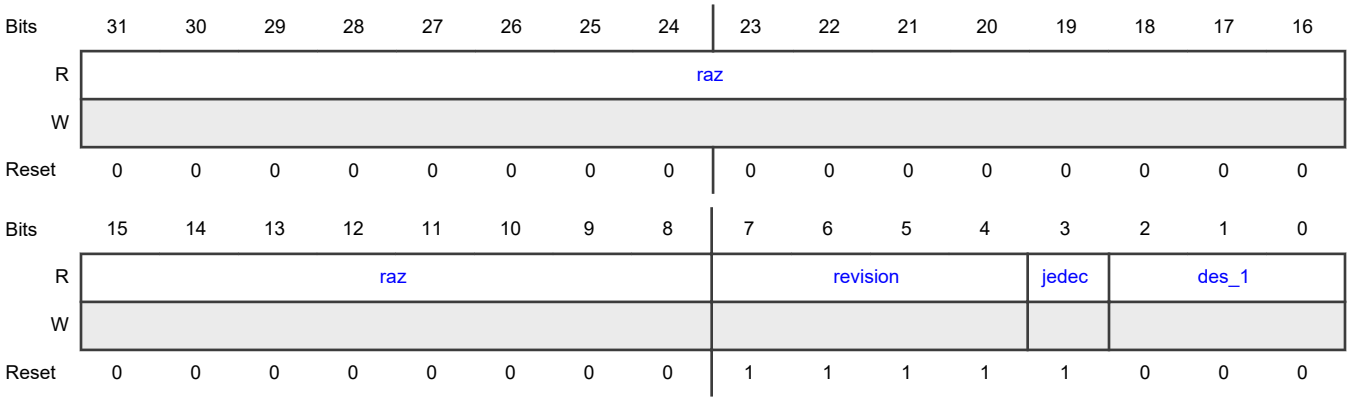
#### Offset

| 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<br>raz | raz<br>Reads As Zero |
| 7-4<br>revision | revision<br>Revision<br>The revision is hard-coded to 4'b1111 which should be ignored.<br>Refer to the VSP HWVERSION register for the VSP version details. |
| 3<br>jedec | jedec<br>Indicates a JEDEC assigned values are used (always 1'b1). |
| 2-0<br>des_1 | des_1<br>JEP106 Identification Code bits[6:4]<br>The concatenated {DES_2, DES_1, DES_0} indicates the designer of the component.<br>DES_1 are the upper bits of the JEP106 identification code (for NXP, DES_1=0x0) |

## 14.1.43 Peripheral ID3 register (PIDR3)

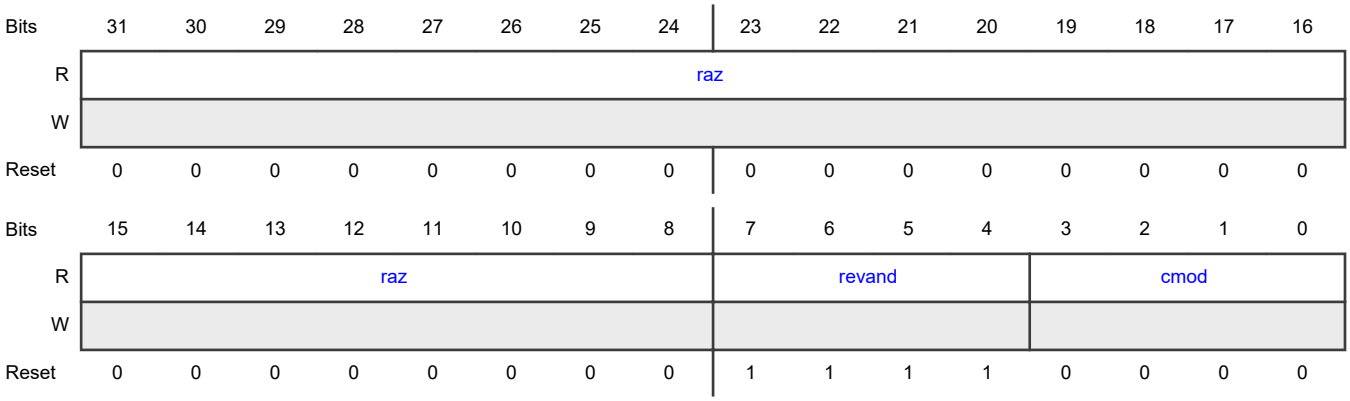
### Offset

| 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<br>raz | raz<br>Reads As Zero |
| 7-4<br>revand | revand<br>RevAnd<br>The minor revision is hard-coded to 4'b1111 which should be ignored.<br>Refer to the VSP HWVERSION register for the VSP version details. |
| 3-0<br>cmod | cmod<br>Customer Modified<br>Currently hard-coded to 4'b0000. |

## 14.1.44 Component ID0 register (CIDR0)

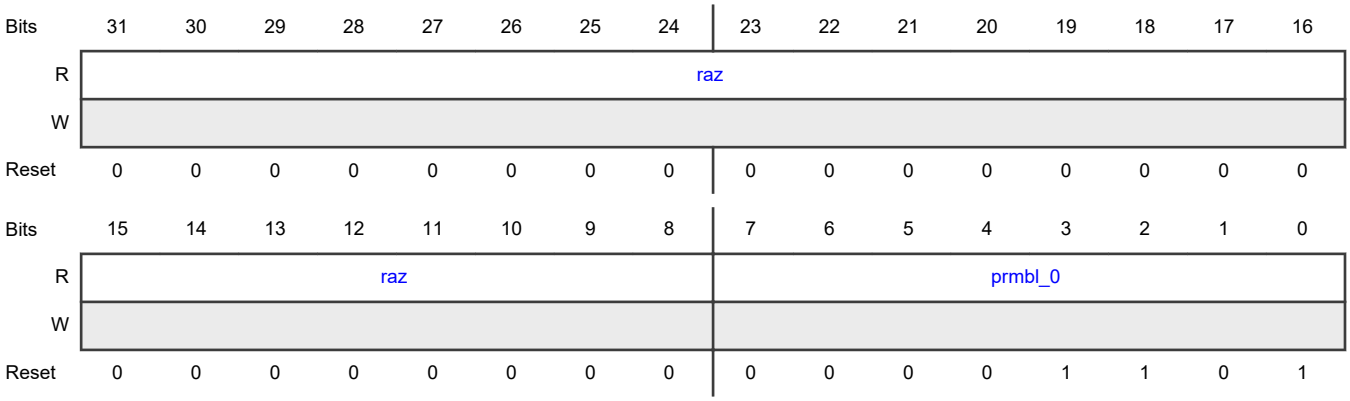
### Offset

| 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<br>raz | raz<br>Reads As Zero |
| 7-0<br>prmbl_0 | prmbl_0<br>Preamble<br>Hard-coded to 0x0D |

## 14.1.45 Component ID1 register (CIDR1)

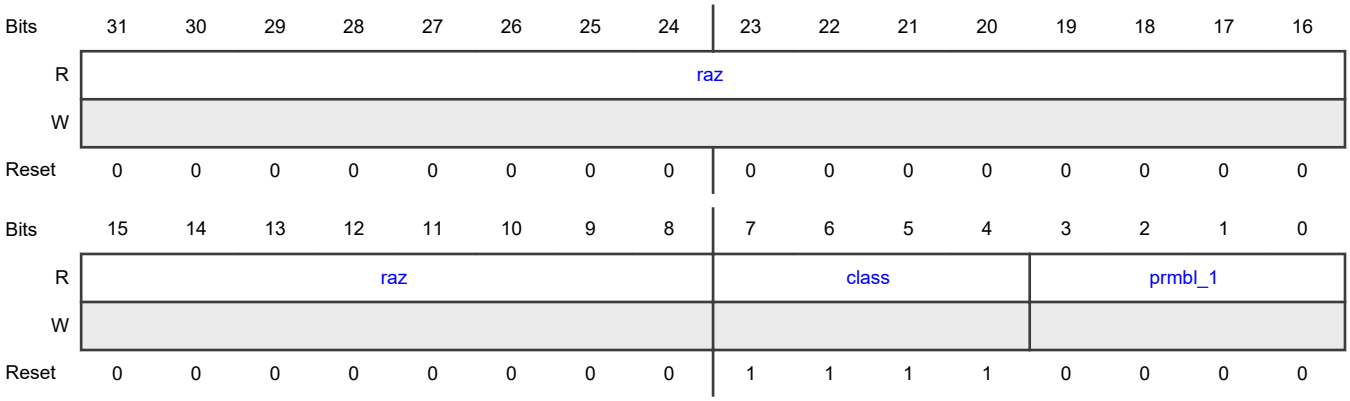
### Offset

| 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<br>raz | raz<br>Reads As Zero |
| 7-4<br>class | class<br>Component Class<br>Hard-coded to 0xF<br>(indicates a CoreLink, PrimeCell, or System component with no standardized register layout) |
| 3-0<br>prmbl_1 | prmbl_1<br>Preamble<br>Hard-coded to 0x0 |

## 14.1.46 Component ID2 register (CIDR2)

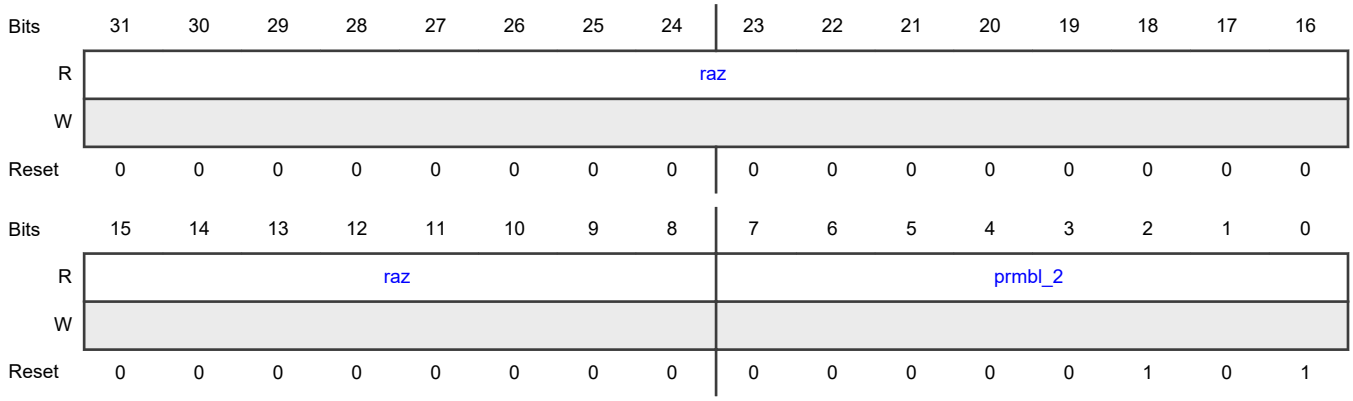
### Offset

| 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<br>raz | raz<br>Reads As Zero |
| 7-0<br>prmbl_2 | prmbl_2<br>Preamble<br>Hard-coded to 0x05 |

## 14.1.47 Component ID3 register (CIDR3)

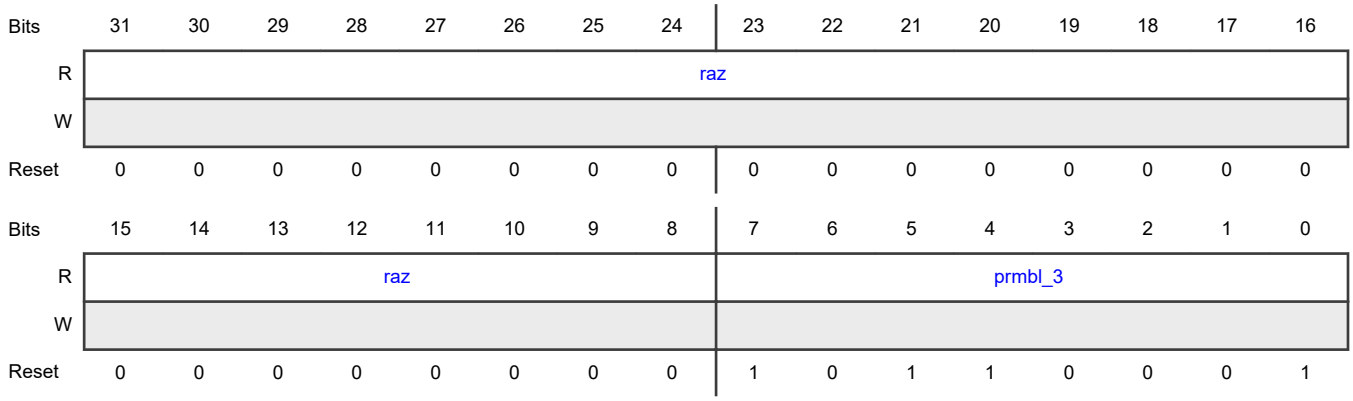
### Offset

| 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<br>raz | raz<br>Reads As Zero |
| 7-0<br>prmbl_3 | prmbl_3<br>Preamble<br>Hard-coded to 0xB1 |

# Appendix A
# Revision History

| | | |
|---|---|---|
| Rev 0 | 04/2021 | Initial 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: 04/2021**
**Document identifier: vspa3_ism_16au_la1224**