Remarks before Reading This Document

* **THIS DOCUMENT MUST BE DELIVERED UNDER PROPER LICENSE OR CONTRACT.**
* **Blue colored descriptions are for V3M Ver.2.0 and V3H.**
* **Red colored descriptions are for V3U. All descriptions for V3U are tentative.**

Revision History

|  |  |  |
| --- | --- | --- |
| ****Revision**** | ****Date**** | ****Description**** |
| **0.01.32** | **Jan. 15, 2018** | **Initial release.** |
| **0.01.33** | **Jan. 19, 2018** | **Added 64-bit access to LWM in** Table 5‑1**.**  **Added constraints on changing values of CR16, CR17, CR18, CR19 and CR20 in section 6.2.**  **Added base addresses of cluster 2 to cluster 4 for V3H in section 12.**  **Corrected usage of FCMP.UNE instruction in section 9.** |
| **0.01.34** | **Mar.2, 2018** | **Added description about rounding mode of reciprocal operation in section 6.1.11.**  **Changed description of setting MSTEN and SLVEN to same value in section 12.1.49.**  **Added description to disable a thread to section 3.1.**  **Added description about behavior of ACTST and SYNCG instructions in section 3.2.** |
| **0.01.35** | **Mar.16, 2018** | **Corrected range of Xlen and Ylen in section 13.3.1.2** |
| **0.01.36** | **Mar.30, 2018** | **Added Terminology in Preliminary section.**  **Made description clear to distinguish TRAP command for CL and TRAP instruction for threads in section 1.2.1, 3.1, 3.2, 8.6, 12.1.20, 12.1.50.**  **Unified multiple aliases (e.g. unified shader, OCV core, etc) to CVengine in entire section of the document.** |
| **0.02.36** | **Apr.6, 2018** | **Added GWM size for V3U in section 1.4 and 5.4.**  **Added Figure 5-3 in section 5.1.**  **Added data sharing of LWM for V3U in section 5.3.**  **Added blocking semaphore for V3U in section 5.5.1.**  **Added no-increment area of the increment register for V3U in section 5.5.2.**  **Added instructions CMP.NEQ8, CMP.NEQ16, CMP.SGT8, CMP.SGT16, CMP.SLE8, CMP.SLE16, DMAWAIT, GMADBSL, GMADBUL, GMADHSL, GMADHUL, LMBD, PLSB8, PLSB16, SHUF, UPLSB8, UPLSB16 in section 9.**  **Changed instructions CMP.UGT8 and CMP.UGT16 to CMP.ULE8 and CMP.ULE16 in section 9.**  **Added description about no pipeline stall for V3U in section 10.3.3.**  **Added description about 64-bit forwarding for V3U in section 10.3.5.**  **Added configuration register SCMSKR in section 12.** |
| **0.02.37** | **Apr.24, 2018** | **Changed expression of “unified processor” to “thread” through the entire document.**  **Revised section 8.9 for better description.**  **Added description of PBCOVF in section 12.1.5.**  **Corrected section 10.3.4 that this stall can happen not only for V3M Ver.2.0 and V3H.** |
| **0.03.38** | **May 18, 2018** | **Added description for master/slave debugging in section 8.4.**  **Corrected DMAWAITS to DMAWAIT in section 10.2**  **Added operand type to DMAWAIT and LPS instructions in section 9.**  **Added description of no pipeline stall for V3U on Figure 10-12 in section 10.3.1.**  **Removed "temporary name" from INT\_PBCOVF in section 13.7.1.2.** |
| **0.04.38** | **Jun.1, 2018** | **Corrected memory addresses for semaphore areas in section 5.5.1 and Figure 5-3.**  **Added description why DMAWAIT instruction was added in section 8.3.**  **Added behavior model for DMAWAIT instruction in section 9.**  **Corrected operand type for LPS instruction in section 9.**  **Corrected behavior model for PLSB8 and PLSB16 instructions in section 9.**  **Corrected SAM type of ABS8 and ABS16 instructions to SAM4 in section10.2.**  **Changed example and description to make clearer in section 10.3.4.**  **Corrected long names for MULUS8 and MULUS16 instructions in section 9.** |
| **0.05.39** | **Jun.29, 2018** | **Added description about no-longer-required NOP in section 5.5.**  **Added requirement of NOP for blocking semaphore in section 5.5.1.**  **Added note for non-blocking semaphore in section 5.5.1.**  **Added information about SCMSK register in section 12.**  **Corrected behavior model for vector comparison instructions in section 9.**  **Corrected I-bit for MAXS8 and following instructions on** Table 10‑2 **in section 10.2.** |
| **0.06.40** | **Jul.27, 2018** | **Added differences between IMP-X5+ and IMP-X6 in the** Preliminary **section.**  **Added detailed description of blocking semaphore in section 5.5.1.**  **Added section** A.1.5  **Tiling Approach.**  **Corrected operand type of MULLU16 instruction in section 9.**  **Replaced most of the word “IMP-X5+” to “CVengine” through the entire document.**  **Replaced most of the word “processor” to “thread” through the entire document.** |
| **0.07.41** | **Aug.31, 2018** | **Added description of EXTSIZE bits on TGDMCR0 register in section 12.1.52.**  **Removed words “Invalidate cache” from the function description of SYNCP instruction in section 9.**  **Added Figure 10-3 in section 10.1.**  **Corrected PC addressing for LPS/LPE instructions from byte to word in section 9.**  **Corrected bit length of D0 for GSADBS/GSADBU instructions from 64 to 32 in section 9.**  **Corrected code for MULS instruction from 0x08 to 0x09 in section 10.2.**  **Corrected description about required SYNCS/SYNCM commands for POINT/POINTS/RECT/TRAP commands in sections 13.1, 13.2, 13.3, 13.5.**  **Added description that DMA can be done to common memory regions in section 8.3.**  **Added register VCR0 value for V3U in section 12.1.1.**  **Added description about clearing 0th level cache upon TRAP instruction in section 1.2.1.**  **Added description about exact delaying cycle in section 10.3.4.** |
| **0.08.42** | **Sep.28, 2018** | **Added restriction of UNR16-63 in section 12.1 and section 12.1.58.**  **Corrected number of execution steps of MULSS8 and MULSS16 in section 10.2.**  **Corrected LWM, GWM and Scratchpad sizes in sections 5.1, 5.3, 5.4, 5.7 and 5.8.**  **Added section 5.8.3 to describe LWM data sharing.**  **Added description about controlling LWM-TGDMAC for slave threads in section 6.2, 8.3 and 9.**  **Added section 8.10 Data Cache Bypassing.**  **Added figures 5-6, 5-10, 5-13, 5-16 and 5-17 in section 5.8.**  **Removed 8 (R8-R15) as a possible operand from register block load/store instructions (e.g. LDSRB) in section 6.1.12 and section 9.**  **Changed name of DMAWAIT instruction to DMAWAITS in sections 8.3, 9 and 10.2.**  **Corrected that SBOCNTR0 must be cleared manually in sections 12.1.15 and 12.1.32.** |
| **0.09.43** | **Oct.31, 2018** | **Added restriction of software reset in section12.1.2.**  **Corrected address and WPR support of USCTRL0 register in section 12 Table 12-1.**  **Corrected value to disable PFTCHEN bit in section 12.1.52.**  **Added description about inserting commands between 2 WUPs in section 0.**  **Added notice that pbuf is connected to D$ only in sections 1.2.1 and A.1.1.**  **Added description about both common/non-common accesses in section 5.1 Table 5-1.**  **Added LWM size for V3U in section 1.3.** |
| **0.09.44** | **Nov.30, 2018** | **Simplified description related R6 register for all instructions accessing Uniform storage and employed #R6IMM in section 9.**  **Corrected description of CR17 register to align to CR16/CR18 in section 6.2.** |
| **0.10.45** | **Dec.28, 2018** | **Corrected links to configuration registers in section 8.6.**  **Removed WPR support of SYNCCRn registers in section 12.**  **Added base addresses of clusters in section 12.**  **Removed “tentative” annotation about accessing slave thread’s LWM-TGDMAC function in section 6.2, 8.3 and 9.**  **Changed size of Scratchpad memory area and employment of the bus system in section 5 and 5.6.**  **Added possible SYNCCn values for V3M Ver.2.0, V3H and V3U in section 13.13.** |
|  |  |  |

Table of Contents

[Preliminary 8](#_Toc531095242)

[1. Overview of IMP-X5+/IMP-X6 CVengine 9](#_Toc531095243)

[1.1 Input Data 11](#_Toc531095244)

[1.2 Cache Memory 11](#_Toc531095245)

[1.2.1 Data Cache 11](#_Toc531095246)

[1.2.2 Instruction Cache 12](#_Toc531095247)

[1.3 Local Working Memory 12](#_Toc531095248)

[1.4 Global Working Memory 12](#_Toc531095249)

[1.5 Uniform Storage 12](#_Toc531095250)

[1.6 Streaming Buffer Output 12](#_Toc531095251)

[1.7 Data Output Port 13](#_Toc531095252)

[2. Overview of Thread Hardware 14](#_Toc531095253)

[3. Thread State 15](#_Toc531095254)

[3.1 Program Start and Stop 15](#_Toc531095255)

[3.2 Master and Slave 15](#_Toc531095256)

[4. Memory Space for Program and Data Planes 18](#_Toc531095257)

[4.1 Program Area 18](#_Toc531095258)

[4.2 Data Planes 18](#_Toc531095259)

[5. Memory Space for Variables and IO 21](#_Toc531095260)

[5.1 Memory Map and Attributes 21](#_Toc531095261)

[5.2 Constant Memory Area 23](#_Toc531095262)

[5.3 Local Working Memory Area 24](#_Toc531095263)

[5.4 Global Working Memory Area 25](#_Toc531095264)

[5.5 Semaphore and Increment Registers Area 25](#_Toc531095265)

[5.5.1 Semaphore Register Area 25](#_Toc531095266)

[5.5.2 Increment Register Area 26](#_Toc531095267)

[5.6 Scratchpad Memory Area 26](#_Toc531095268)

[5.7 Queue Arbiter Register Area 26](#_Toc531095269)

[5.8 Common and Non Common Area 27](#_Toc531095270)

[5.8.1 Non-Common Access to LWM 27](#_Toc531095271)

[5.8.2 Common Access to LWM 27](#_Toc531095272)

[5.8.3 Access Collision to LWM Common Area 28](#_Toc531095273)

[5.8.4 Common and Non-Common Access to GWM 29](#_Toc531095274)

[5.8.5 Common and Non-Common Access to Scratchpad 31](#_Toc531095275)

[6. CVengine Thread Registers 33](#_Toc531095276)

[6.1 List of CVengine Thread Registers 33](#_Toc531095277)

[6.1.1 Coordinate Input Registers (R0 and R1) 35](#_Toc531095278)

[6.1.2 Inter-Thread Register (R2) 35](#_Toc531095279)

[6.1.3 Zero Register (R3) 36](#_Toc531095280)

[6.1.4 Uniform Index Register (R6) 36](#_Toc531095281)

[6.1.5 Sine Operation Register (R8) 36](#_Toc531095282)

[6.1.6 Cosine Operation Register (R9) 36](#_Toc531095283)

[6.1.7 Inverse of Square Root Operation Register (R10) 37](#_Toc531095284)

[6.1.8 Exponent Operation Register (R11) 37](#_Toc531095285)

[6.1.9 Logarithm Operation Register (R12) 37](#_Toc531095286)

[6.1.10 Square Root Register (R13) 37](#_Toc531095287)

[6.1.11 Reciprocal Operation Registers (R14 and R15) 38](#_Toc531095288)

[6.1.12 R4, R5, R7, R16 to R47 and 64-bit Pair Register 38](#_Toc531095289)

[6.1.13 Status Register 38](#_Toc531095290)

[6.2 Control Registers of CVengine Thread 39](#_Toc531095291)

[7. Floating-Point 41](#_Toc531095292)

[7.1 Floating-Point Data Format 41](#_Toc531095293)

[7.2 IEEE754 Conformance 41](#_Toc531095294)

[8. Peripheral Functions 42](#_Toc531095295)

[8.1 Streaming Buffer Output (SBO) 42](#_Toc531095296)

[8.2 SOU and SAM 43](#_Toc531095297)

[8.3 DMA 44](#_Toc531095298)

[8.4 Debug Support 46](#_Toc531095299)

[8.5 Interrupt 47](#_Toc531095300)

[8.6 Performance Counter 48](#_Toc531095301)

[8.7 Image Clipping 48](#_Toc531095302)

[8.8 Border Padding 49](#_Toc531095303)

[8.9 Guard Timer 51](#_Toc531095304)

[8.10 Data Cache Bypassing 53](#_Toc531095305)

[9. Instruction Set 54](#_Toc531095306)

[9.1 Conditional instruction execution 55](#_Toc531095307)

[10. Instruction Format, Code and Pipeline Operation 124](#_Toc531095308)

[10.1 Instruction Format 125](#_Toc531095309)

[10.2 Instruction Code List 128](#_Toc531095310)

[10.3 Pipeline Operation of CVengine Thread 136](#_Toc531095311)

[10.3.1 Pipeline operations for each instruction group 136](#_Toc531095312)

[10.3.2 Pipeline stall on arbitration 140](#_Toc531095313)

[10.3.3 Pipeline stall on SR flag 140](#_Toc531095314)

[10.3.4 Pipeline stall on FMAD/FMSU and FADD/FSUB instructions 141](#_Toc531095315)

[10.3.5 Forwarding with 32-bit and 64-bit instructions 141](#_Toc531095316)

[10.3.6 Forwarding with R6 register 141](#_Toc531095317)

[10.4 Number of Execution Cycles of Special Instructions of CVengine Thread 143](#_Toc531095318)

[11. Vertex Issue 144](#_Toc531095319)

[12. Configuration Registers 146](#_Toc531095320)

[12.1 Register Description 149](#_Toc531095321)

[12.1.1 Version Control Register 0 (VCR0) 149](#_Toc531095322)

[12.1.2 ReSeT Register (RSTR) 150](#_Toc531095323)

[12.1.3 Control Register (CR) 151](#_Toc531095324)

[12.1.4 Status Register 0 (SR0) 152](#_Toc531095325)

[12.1.5 Status Register 1 (SR1) 153](#_Toc531095326)

[12.1.6 Status Clear Register 1 (SCR1) 155](#_Toc531095327)

[12.1.7 Interrupt Control Register 1 (ICR1) 156](#_Toc531095328)

[12.1.8 Interrupt Mask Register 1 (IMR1) 157](#_Toc531095329)

[12.1.9 Status Register 2 (SR2) 158](#_Toc531095330)

[12.1.10 Status Clear Register 2 (SCR2) 159](#_Toc531095331)

[12.1.11 Interrupt Control Register 2 (ICR2) 160](#_Toc531095332)

[12.1.12 Status Register 3 (SR3) 161](#_Toc531095333)

[12.1.13 Shader Core MaSK Register (SCMSKR) 162](#_Toc531095334)

[12.1.14 Thread Group Control Register 0 (TGCR0) 163](#_Toc531095335)

[12.1.15 SBO CouNT Register0 (SBOCNTR0) 164](#_Toc531095336)

[12.1.16 Descriptor List Start Address Register (DLSAR) 165](#_Toc531095337)

[12.1.17 CLiPping MINimum Register (CLPMINR) 166](#_Toc531095338)

[12.1.18 CLiPping MAXimum Register (CLPMAXR) 167](#_Toc531095339)

[12.1.19 Shader Performance Control Register (SPCR) 168](#_Toc531095340)

[12.1.20 Shader Performance BuSY CouNTer Register (SPBSYCNTR) 169](#_Toc531095341)

[12.1.21 Shader Performance Busy Counter THReshold Register (SPBCTHRR) 170](#_Toc531095342)

[12.1.22 Data Cache BYPass Area 0 MIN (DCBYPA0MIN) 171](#_Toc531095343)

[12.1.23 Data Cache BYPass Area 0 MAX (DCBYPA0MAX) 172](#_Toc531095344)

[12.1.24 Data Cache BYPass Area Enable (DCBYPAEN) 173](#_Toc531095345)

[12.1.25 Data Cache BYPass Area ConTroL (DCBYPACTL) 174](#_Toc531095346)

[12.1.26 Vs PC Start Address Register (VPCSAR) 175](#_Toc531095347)

[12.1.27 Vs SINI Start Address Register (VSINISAR) 176](#_Toc531095348)

[12.1.28 Ps PC Start Address Register (PPCSAR) 177](#_Toc531095349)

[12.1.29 Ps SINI Start Address Register (PSINISAR) 178](#_Toc531095350)

[12.1.30 Vs I$ Base Address Register (VIBAR) 179](#_Toc531095351)

[12.1.31 Ps I$ Base Address Register (PIBAR) 180](#_Toc531095352)

[12.1.32 SBO Base Address Register 0 (SBOBAR0) 181](#_Toc531095353)

[12.1.33 SBO Max Memory Size Register 0 (SBOMMSR0) 182](#_Toc531095354)

[12.1.34 SBO Control Register 0 (SBOCR0) 183](#_Toc531095355)

[12.1.35 DMac Control Register 0 (DMCR0) 184](#_Toc531095356)

[12.1.36 DMac EXTernal MEMory OFfSet ADDress Register (DMEXTMEMOFSADDR) 186](#_Toc531095357)

[12.1.37 DMac GWM Base ADDress Register (DMGWMBADDR) 187](#_Toc531095358)

[12.1.38 DMac GWM STRide Register (DMGWMSTRR) 188](#_Toc531095359)

[12.1.39 DMac LeNGth Register (DMLNGR) 189](#_Toc531095360)

[12.1.40 IMaGe Base Address Register n (IMGBARn) (n = 0 to 7) 190](#_Toc531095361)

[12.1.41 IMaGe STRide Register n (IMGSTRn) (n = 0 to 7) 191](#_Toc531095362)

[12.1.42 IMaGe Control0 Register n (IMGC0Rn) (n = 0 to 7) 192](#_Toc531095363)

[12.1.43 IMaGe SIZE Register 0 (IMGSIZER0) 194](#_Toc531095364)

[12.1.44 IMaGe CoNSTant Register 0 (IMGCNSTR0) 195](#_Toc531095365)

[12.1.45 SYNChronization Control Register 0 (SYNCCR0) 196](#_Toc531095366)

[12.1.46 SYNChronization Control Register 1 (SYNCCR1) 197](#_Toc531095367)

[12.1.47 SYNChronization Control Register 2 (SYNCCR2) 198](#_Toc531095368)

[12.1.48 SYNChronization Control Register 3 (SYNCCR3) 199](#_Toc531095369)

[12.1.49 TG Shader Enable Register 1 (TGSEN1) 200](#_Toc531095370)

[12.1.50 Unified Shader PerFormance Counter ConTrol Register (USPFCCTLR) 202](#_Toc531095371)

[12.1.51 PerFormance Counter TOTAL Register0 (PFCTOTALR0) 204](#_Toc531095372)

[12.1.52 Unified Shader ConTRoL Register 0 (USCTRL0) 205](#_Toc531095373)

[12.1.53 TGDMac Control Register 0 (TGDMCR0) 206](#_Toc531095374)

[12.1.54 TGDMac EXTernal MEMory OFfSet ADDress Register (TGDMEXTMEMOFSADDR) 208](#_Toc531095375)

[12.1.55 TGDMac LWM Base ADDress Register (TGDMLWMBADDR) 209](#_Toc531095376)

[12.1.56 TGDmac LWM STRide Register (TGDMLWMSTRR) 210](#_Toc531095377)

[12.1.57 TGDmac LeNGth Register (TGLNGR) 211](#_Toc531095378)

[12.1.58 Uiform Register n (UNRn) (n = 0 to 63) 212](#_Toc531095379)

[13. Command List (CL) 213](#_Toc531095380)

[13.1 POINT (Point) 213](#_Toc531095381)

[13.2 POINTS (Point Separate) 213](#_Toc531095382)

[13.3 RECT (RECTangle) 214](#_Toc531095383)

[13.4 NOP(No OPeration) 216](#_Toc531095384)

[13.5 TRAP(TRAP) 216](#_Toc531095385)

[13.6 WPR(Write PaRameter) 216](#_Toc531095386)

[13.7 INT(INTerrupt) 217](#_Toc531095387)

[13.8 GOSUB(GOSUB) 217](#_Toc531095388)

[13.9 RET(RETurn) 218](#_Toc531095389)

[13.10 SYNCM(SYNChronize Memory) 218](#_Toc531095390)

[13.11 SYNCS(SYNChronize Shader) 218](#_Toc531095391)

[13.12 SINI(Shader INItialize) 219](#_Toc531095392)

[13.13 SLP(SLeeP) 219](#_Toc531095393)

[13.14 WUP(Wake UP) 220](#_Toc531095394)

[Appendix 222](#_Toc531095395)

[A.1.1 Effective Use of 0th Level Cache 222](#_Toc531095396)

[A.1.2 Kernel Processing for Each Area 223](#_Toc531095397)

[A.1.3 Loop Unrolling 223](#_Toc531095398)

[A.1.4 Effective Use of Instruction Buffers 223](#_Toc531095399)

[A.1.5 Tiling Approach 224](#_Toc531095400)

# Preliminary

This document describes the programming model and specifications of the CVengine incorporated in the IMP-X5+.

The CVengine in the IMP-X5+ extends features of IMP-X5 as listed below.

* SIMD unit and dedicated instructions
* New arithmetic logics, sine, cosine, inverse of square root
* Scratchpad memory (Former IMPC, function is the same, size is increased)
* Master/Slave architecture
* DMA transfer can be initiated by CVengine program.
* Comprehensive load/store instructions to access memory areas including the external DDR memory.

On the other hand, the following functionalities are removed from IMP-X5.

* Histogram processor
* Stack cache
* data planes 8 to 15
* SBO1 and SBO2
* Compatibility(VS/PS) mode

Major feature updates of the CVengine in the IMP-X6 from IMP-X5+ are as listed below.

* Added LIDAR and RADAR specialized SIMD instructions
* Doubled LWM and GWM memory size
* Expanded register file from 32 to 48
* Added blocking semaphore registers
* Added no-increment access to increment registers

**Terminology**

* “Shader” is an old name and an alias of “CVengine”. “OCV core” is also an alias of “CVengine”.
* “Unified Processor” is an alias of “Thread”.
* “Descriptor List” and “Display List” are old names of “Command List”
* There are two kinds of “TRAP”. “TRAP command” is used for Command List, “TRAP instruction” is used for Thread.

# Overview of IMP-X5+/IMP-X6 CVengine

The IMP-X5+/IMP-X6 CVengine is configured of a hierarchy of three types of unit referred to as the cluster, cores, and threads. Each of the units is described below.

A thread retains the thread state and is the element that executes a program based on the vertices it is assigned. As many programs as there are threads can be executed in parallel.

Multiple threads are collected in the units called cores. As well as the multiple threads, a core has local working memory (LWM) and Shared ArithMetic logic unit (SAM), both of which are shared among the multiple threads of the core.

Each thread in a core is configured by the command list as “master” or “slave”. The master and slave have own program area. And the master thread can fire up the slave thread by the ACTST instruction.

Multiple cores are collected in the unit called a cluster. A cluster serves as one processing unit of an IMP-X5+/IMP-X6 CVengine.

As well as the multiple cores, a cluster has various caches (for data and instructions denoted by D$ and I$), a stream buffer output (SBO), global working memory (GWM), Shared Operation Unit (SOU). These are all shared among the multiple cores of the cluster.

The scratchpad memory and external memory are implemented outside of clusters.

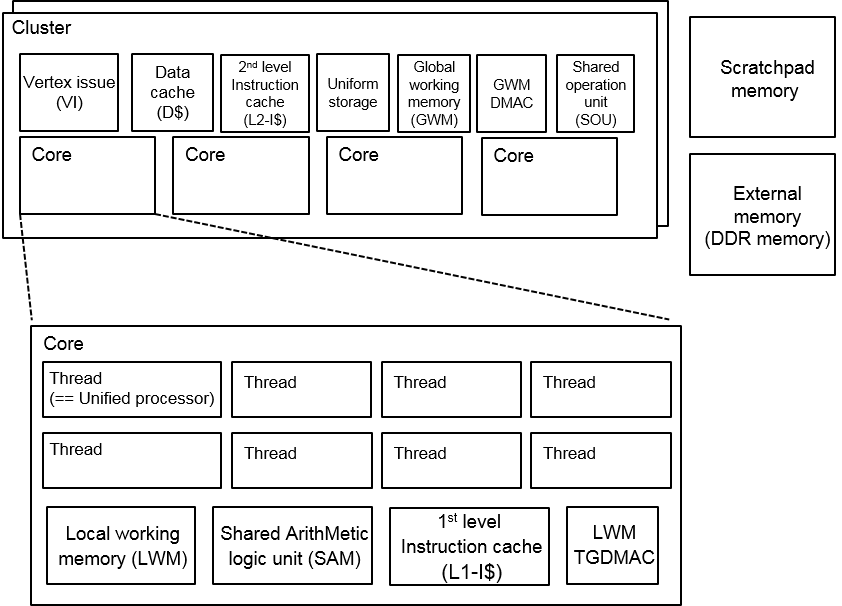


Figure 1‑1 Block Diagram of an CVengine Cluster

The programming model focuses on the thread as the unit of operations by CVengine and its specifications are described in the remainder of this document.

The CVengine incorporates a single type of processor that is the CVengine thread. The CVengine thread is connected to the following ports or modules:

(1) Data input port

(2) Data cache

(3) Instruction cache

(4) Local working memory

(5) Global working memory

(6) Uniform storage

(7) Main memory / Scratchpad memory

(8) Streaming buffer output

(9) Data output port

(10) Data cache bypassing (V3U only)

Figure 1.2 shows an overview of the CVengine thread.

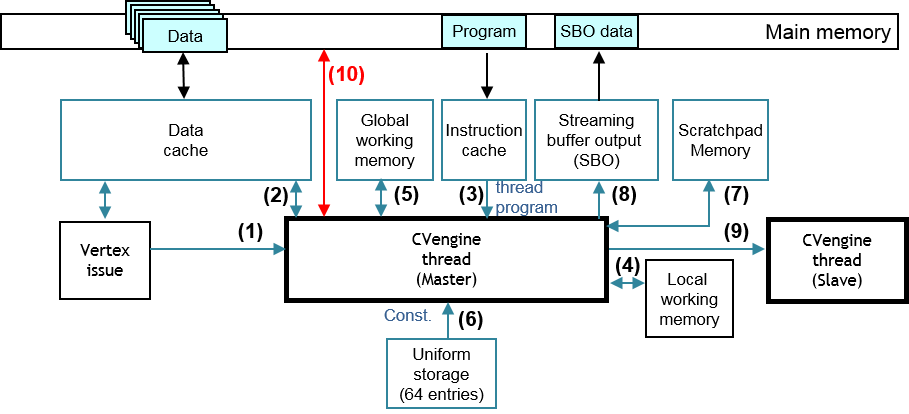


Figure 1‑2 Connection Diagram of CVengine Thread and Peripheral Blocks

## Input Data

The data input from the data input port is stored in registers from which the thread can read.

## Cache Memory

There are data cache and instruction cache between main memory and CVengine.

Table 1‑1 Cache Memory List

|  |  |  |  |
| --- | --- | --- | --- |
|  | Data Cache | L1 Instruction Cache | L2 Instruction Cache |
| Shared among | All threads in a cluster | All threads in a core | All cores in a cluster |
| Size | 8kB x 4/1/1bank(s)\*1 | 4kB | 8kB |
| Line size | 128byte | ← | ← |
| Structure | 4-way set associative | 2-way set associative | 4-way set associative |
| Tag | Upper 19bit of physical address | Upper 21bit of physical address | Upper 21bit of physical address |
| Refill | LRU | ← | ← |
| Purging | Write back | NA | NA |
| Stage | 7 stages (responded at stage 6) | ← | ← |
| Remarks |  |  |  |

\*1 8kB x 1 bank for V3M Ver.2.0, V3H and V3U.

### Data Cache

A data cache memory consists of 4banks of 8kbyte memory and is connected to CVengine for fast access to image data and variables in the external memory. For V3M Ver.2.0, V3H and V3U, the data cache memory consists of 1bank of 8kbyte memory. Access to the data cache from the thread is specified by X-Y coordinates or linear address on Shader memory space. Refer to Figure 5‑1 for this linear memory space. Addresses in the main memory are automatically calculated from the specified X-Y coordinates or linear address using the image, data format, base address, or stride, and that address is accessed. More than one base address can be handled and they are specified by configuration registers.

0th level cache (named as “pbuf”) has a 32byte long write buffer with write-back method and a 32byte read buffer which are implemented at the data cache (D$) access port of each thread to improve data access performance at D$ conflicts. This cache is automatically flushed and cleared at the thread end (TRAP instruction). For V3M Ver.2.0, V3H and V3U, this cache is automatically flushed but not cleared at the thread end (TRAP instruction). Clearing this cache is user responsibility .Also a dedicated instruction SYNCP is available to clear or flush the cache artificially and the content in pbuf is invalidated upon the SYNCP instruction. For details on improving performance by effectively using the 0th level cache, refer to sectionA.1.4 Effective Use of Instruction Buffers. Be noticed that pbuf is only connected to D$ thus it does not work for accessing to any other memory (e.g. LWM).

The multiple threads share the data cache. Use software to maintain data coherence between the threads if required.

For V3U, this data cache can be bypassed. Refer to section 8.10.

### Instruction Cache

The program which is executed by the thread is supplied from the instruction cache to the thread. The instruction cache consists of the level 1 cache which is 4kbyte of 2way set associative cache and shared all 8 threads in a core. The level 2 cache is 8kbyte of 4way set associative cache which is implemented per cluster and is shared by all 32 threads.

In addition to above, 0th level cache is also implemented as quad buffers of 32byte long between the thread and the instruction cache. 32 instructions can be stored in the 0th level instruction cache.

## Local Working Memory

The local working memory is readable and writable memory. In non-common access mode, each thread has its own dedicated local working memory bank and it is only accessible by the thread. In common access mode, the full local working memory address space can be accessed by any thread inside one core. The size of local working memory is 2kB per thread in non-common mode and 16kB per core in common mode. For V3U, the size of local working memory is 4kB per thread in non-common mode and 32kB per core in common mode. Refer to section 5.8 Common and Non Common Area.

The local working memory can also be used as space for holding temporary variables which cannot be stored in general-purpose registers.

## Global Working Memory

The global working memory is readable and writable and is shared by all threads of the CVengine. It can be used to hold common data which is shared by all thread. The size of global working memory is 4kB per cluster. For V3U, the size of global working memory is 8kB per cluster.

The global working memory has two types of special storage. One storage is a semaphore register. This register is a 1-bit register that can be read from and written to by all threads. When this register is read, the register value is forcibly changed to 1 after the read operation. This allows the shared resources between the threads to be managed.

The other storage is an increment register. This register is a 32-bit register which can be read from and written to by all threads. When this register is read, the register value is incremented by 1 after the read operation. This allows the order between the threads to be controlled.

Since all threads share the global working memory, use software to maintain data coherence.

## Uniform Storage

The uniform storage is used to hold constants that can be used by programs. Values that were stored as source operands from specific instructions can be used. Values need to be already set in the uniform storage by the Vertex Issue through the command list before programs are started. There are 64 entries for 32-bit long values.

## Streaming Buffer Output

The streaming buffer output is used to consecutively write data to the external memory. Up to three types of data can be written. The type of data to be written can be set by the control register.

As the streaming buffer output is shared by multiple threads, the order in which data is written to the external memory cannot be guaranteed between the threads.

## Data Output Port

The data output port of the thread is for use in starting slave threads by issuing ACTST instruction.

# Overview of Thread Hardware

The features and overview of the thread are shown in Figure 2‑1.

Features

* Input/output interface to register file
* 32 and 16-bit signed-integer arithmetic units and floating-point arithmetic unit operating in the IEEE 754 single-precision format
* SIMD operation for multiple-accumulation with 64-bit width input.
* Data read from the constant area (uniform storage) can be used in operations.
* Conditional execution instructions are supported.
* SAM supports special arithmetic operations such as multiplying, rotating are shared by multiple threads.
* SOU supports other special arithmetic operations such as square root are shared by multiple cores.

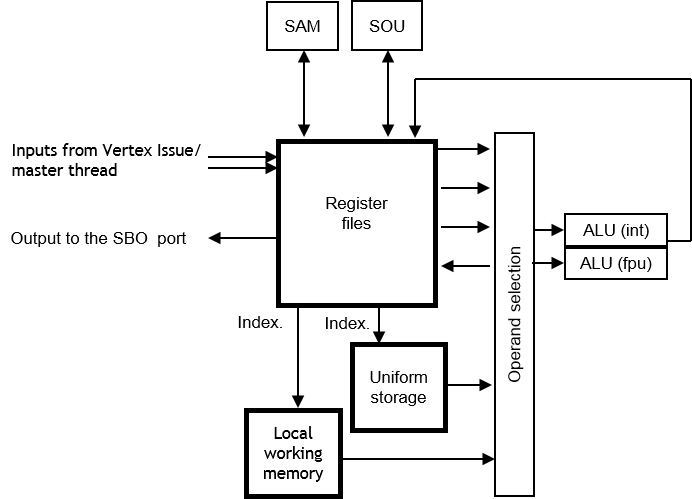


Figure 2‑1 Data Flow in CVengine Thread

Input/output ports to interface with external modules are connected to the register files, and the thread can make access by reading from registers or writing to registers. Each register file consists of thirty-two 32-bit registers. For V3U, each register file consists of forty eight 32-bit registers.

# Thread State

## Program Start and Stop

The thread can be in one of two states: Idle state and processing state. The PC (Program Counter) is stopped and no operation is done in the idle state. When setting values to the necessary input registers by the previous stage (vertex issue/master thread) of the thread is complete, an activation signal is input from the previous stage, the specified PC is loaded, and a transition to the processing state is made. To shift from the processing state to the stopped state, execute the TRAP instruction. The thread enters the idle state.

If a break occurs in the processing state due to a CVengine configuration register setting, the thread transits to the break state. When a break occurs, a transition is made to the break state after execution of the instruction being decoded has completed. Also when an interrupt happens with the INT instruction, the thread goes to the break state. The PC is stopped and the thread state can be read and changed by a CVengine configuration register in the break state. When the break is released due to a CVengine configuration register setting, the thread enters the processing state.

A thread can be disabled. Refer to section 12.1.49. The CVengine can be reset by a configuration register but resetting while a thread is running is prohibited and behavior is not defined. Refer to section 12.1.2.

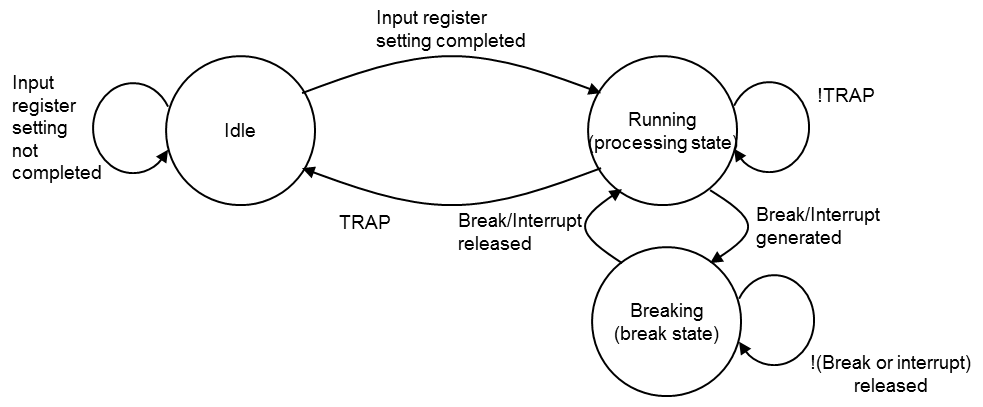


Figure 3‑1 State Transitions of CVengine Thread

## Master and Slave

Each thread in CVengine can take a role as master or slave. This role is configured for each thread by TGSEN1 configuration register. Differences between master and slave are as follows.

* The Vertex Issue starts master thread only.
* The slave threads are only called by master threads by ACTST, which is flexible assignment.
* The only master thread can issue SYNCG and wait for completion of the slave threads.
* Use of R2 to transfer parameter from master to slave (e.g. master ID or LWM base address).
* The slave thread starts from a dedicated instruction area which is specified by PPCSAR and PSINISAR.

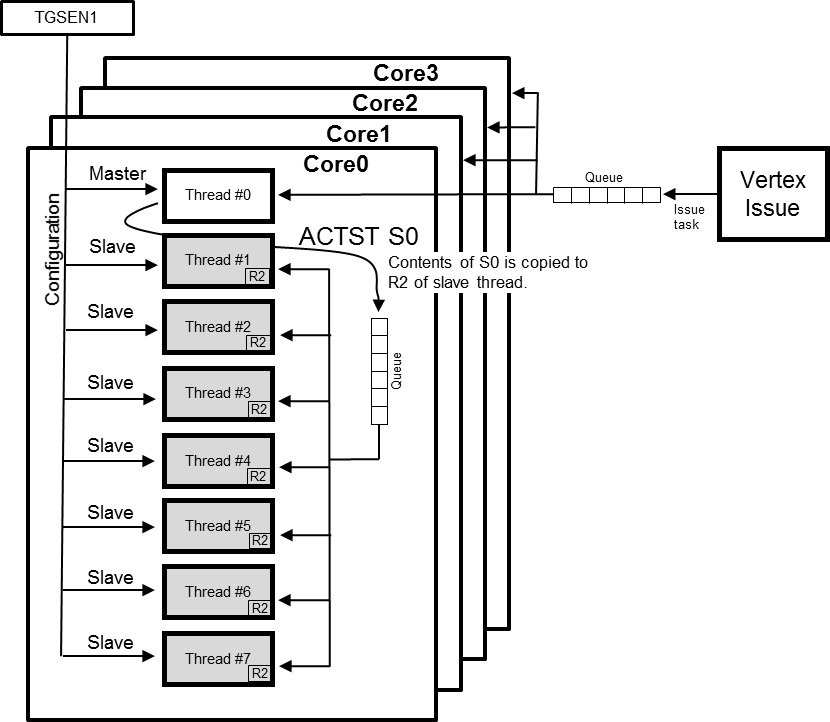


Figure 3‑2 Master/Slave Architecture (Case of 1 Master and 7 Slaves)

Each master thread with own slave threads are combined in a group. The master thread issues SYNCG instruction and wait for completion of such thread group. Issuing SYNCG or ACTST command by a slave thread causes undefined result. Hardware is not initialized automatically when a slave thread is started. Initialize manually with SINI if necessary.

The following figure shows a typical and basic master-slave sequence in case of 1 master and 7 slaves configuration.

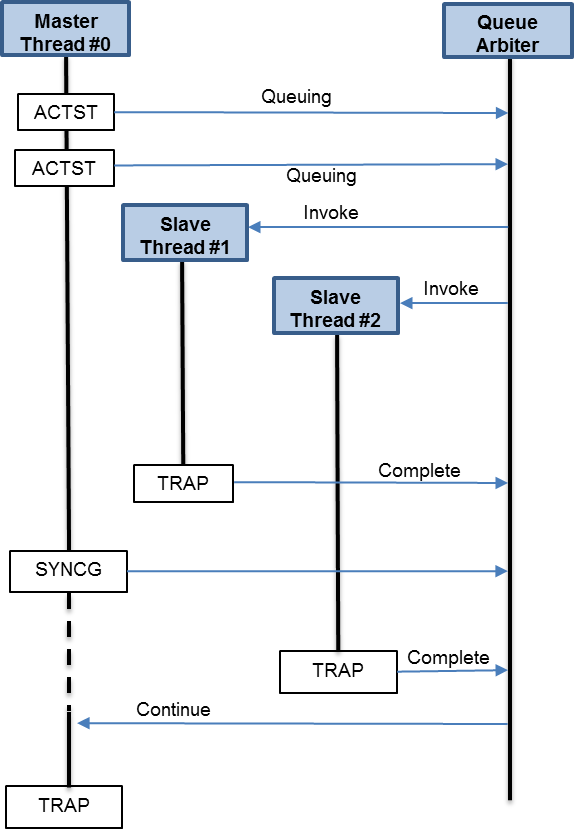


Figure 3‑3 Typical Master/Slave Sequence

In the above figure, the master thread deploys two slave threads by issuing two ACTST via the queue arbiter, then two slave threads proceed their task. The queue arbiter detects completions (TRAP instruction) from all slaves and resumes the master thread by SYNCG.

As derived cases from above;

* The master can issue TRAP without SYNCG.
* The master can issue SYNCG with no ACTST issuing. In this case, SYNCG is treated as NOP.

The master can issue ACTST any times. If there is no vacant slave, the master stalls. SYNCG makes the master to wait for the all the slave threads that are invoked with ACTST by the master thread.

Only master threads can initiate a LWM-DMA transfer, only 1 DMA transfer can be initiated at a time from each master.

All threads can be set to either master or slave with TGSEN1. This TGSEN1 setting is applied to all 4 cores. Refer to section 12.1.49. Any combination of master and slave is possible.

# Memory Space for Program and Data Planes

The CVengine has two type of memory space. One is the memory space for program and data planes. The threads access the program area as instruction fetch and the threads also access data planes indexed by coordinates on data planes with dedicated instructions like LDRXY, STRXY, etc. These memory areas are scope of this chapter.

On the other hand, another memory space is for variables and IO which is described in the next chapter.

## Program Area

The program of the CVengine thread is located in the area allocated in the main memory. Main memory can be DDR RAM or Scratchpad memory.

The start address of the area allocated in the main memory needs to be set in the CVengine configuration register VIBAR before program execution starts.

The program is accessed via the instruction cache (I$). There are two types of instruction caches (I$): a 4kbyte 2way set-associative first-level (L1) cache which is shared by all threads in a core and an 8kbyte 4way set-associative second-level (L2) cache which is shared by all cores in a cluster.

The thread regards the instruction located at the address specified by the VIBAR register as an instruction whose program counter (PC) is 0, and execution starts from the start PC set in the CVengine configuration register. When the SINI command of the vertex issue is used, execution starts from the PC set in the VSINISAR register. When other commands are used, execution starts from the PC set in the VPCSAR register.

The main memory space that can be accessed as the program area by the thread is 64Kbytes from the start address specified by the VIBAR register.

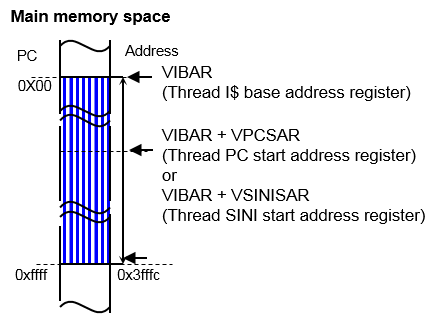


Figure 4‑1 Memory Space for Program Area

## Data Planes

This memory space holds image data or variables. The area that is accessed by the thread via the data cache (D$) is a two-dimensional space that is always represented by the X and Y coordinates. At the data cache, the X and Y coordinates requested by the thread are converted into a memory address and the address in the main memory is accessed.

With instructions for the loading and storing of integers (the FP bit of the instruction is 0), if the data format of the external memory is not 32-bit signed integer, in the case of loading, data from the external memory are converted into 32-bit signed integers and then provided to the thread, and, in the case of storing, the 32-bit signed integer from the thread is converted into the data format of the external memory, after which the value is written to the external memory.

With instructions for the loading and storing of floating-point numbers (the FP bit of the instruction is 1), if the data format of the external memory is not 32-bit floating-point number, in the case of loading, data from the external memory are converted into 32-bit floating-point numbers and then provided to the thread, and, in the case of storing, the 32-bit floating-point number from the thread is converted into the data format of the external memory, after which the value is written to the external memory.

The D$ is composed of four banks of 8-Kbyte 4-way set associative cache. For V3M Ver.2.0, V3H and V3U, the D$ is composed of one bank of 8-Kbyte 4-way set associative cache.

The CVengine configuration registers (IMGBAR0 to IMGBAR7, IMGSTR0 to IMGSTR7, and IMGC0R0 to IMGC0R7) need to be set before executing the program. The CVengine can handle up to 8 planes of memory space. The n-th plane registers are described as IMGBARn, IMGSTRn, and IMGC0Rn in this document. The planes are shared by all threads. The values of Control Registers CR8 and CR9 are used to select which plane to be read or written by a thread. Refer to section 6.2.

The start address of the data area of the plane should be set in the IMGBARn register. The memory width of the plane should be set in the IMGSTRn register. Both of the settings should be made based on the unit of the data format specified by the IMGC0Rn register. The external memory data format, rounding mode at conversion between integer and floating-point, and saturation specification at conversion of the plane are set in the IMGC0Rn register.

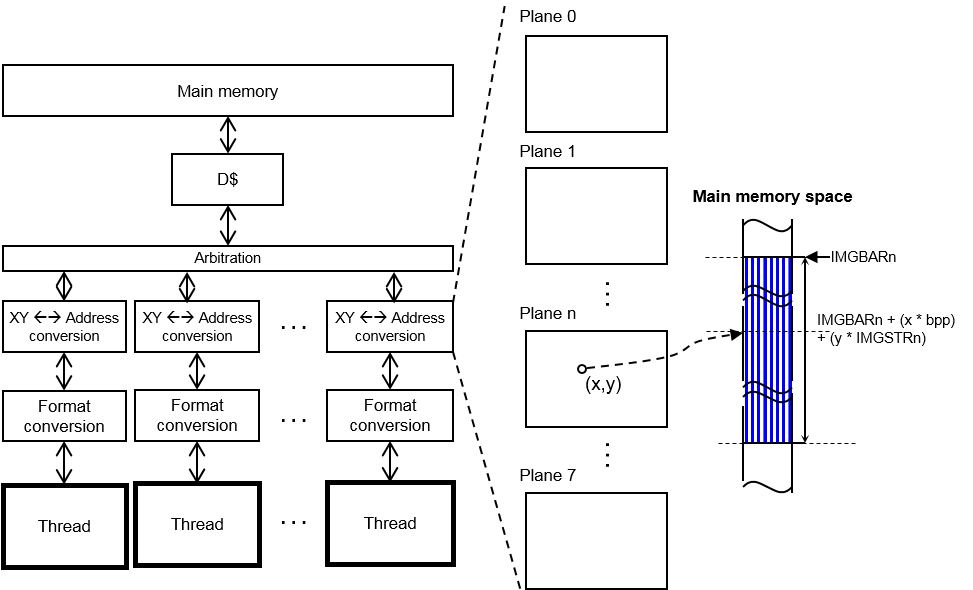


Figure 4‑2 Memory Space for Data Cache Area

**Note: Maximum size of data plane accessed by the thread**

The physical address accessed by a thread is calculated with the following formula: Physical Address = Base + (X × Format) + (Y × Stride). Where Base is the value of IMGBARn, Format is IMGC0Rn.PFORMAT/PFORMAT2 and Stride is IMGSTRn. Base is unsigned 32-bit (0 ~ 4294967295), X is signed 32-bit (-2147483648 ~ 2147483647), Format is 1/2/4 and Y and Stride are signed 16-bit (-32768 ~ 32767). The final offset to Base, that is (X × Format) + (Y × Stride), must not be a negative value. Also the final result, that is Physical Address, must be within the 32-bit range.

Since the final offset to Base cannot take a negative value, the maximum area size that can be accessed by a thread is 2147483648 × 1 bytes for one dimensional plane. And for ordinal two dimensional plane, maximum width is 32767 bytes since Stride is signed 16-bit, and the maximum height is 32768 lines since effective range of Y is 0 ≤ Y ≤ 32767. The maximum size in pixels varies depending on the Format. (On the other hand, Vertex Issue can handle maximum 65536 × 65536 pixels regardless of the Format. Refer to section 13.1, 13.2, 13.3.)

# Memory Space for Variables and IO

## Memory Map and Attributes

The CVengine has a 32-bit address space and any memory spaces for variables and I/O are mapped on it. And the CVengine can access host memory space with the same address, which is useful to read/write a DDR external memory directly. The Figure 5‑1 shows this address space with some attributions for each memory area.

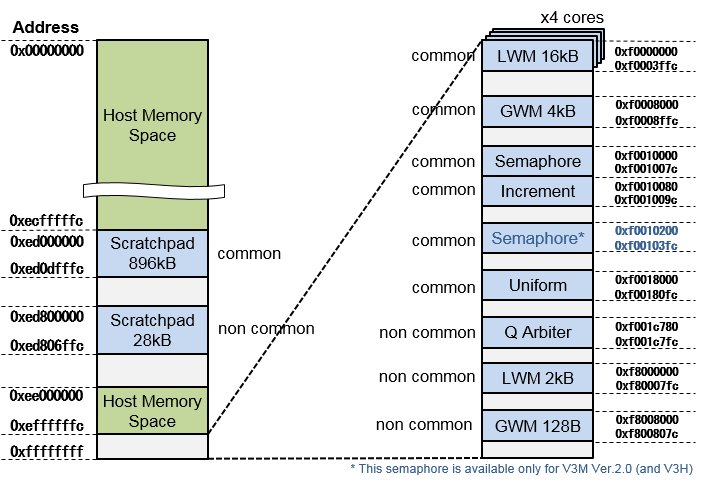


Figure 5‑1 Memory Map for Variables and IO (V3M / V3M Ver.2.0)

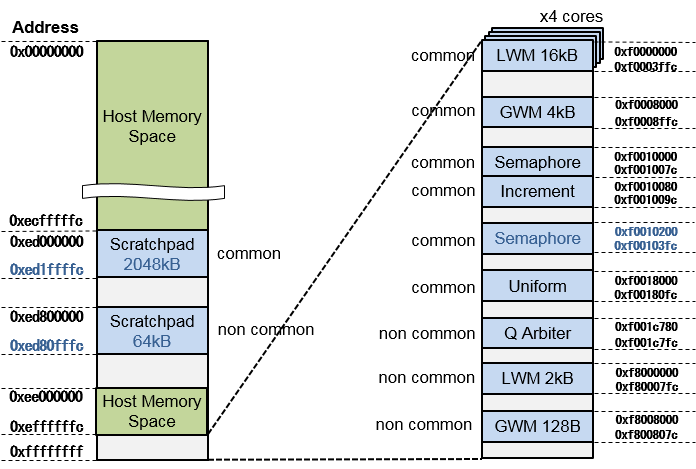


Figure 5‑2 Memory Map for Variables and IO (V3H)

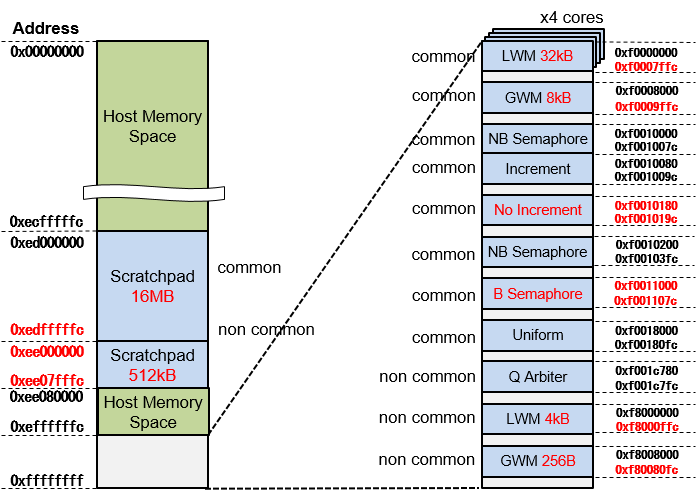


Figure 5‑3 Memory Map for Variables and IO (V3U)

For this address space, dedicated load/store instructions are used to access actual data. And these load/store instructions have variants to access memory with byte and half word size. For example, store instructions have ST (word size), STH (half word size) and STB (byte size). Any access to the area nothing is mapped has no effect, i.e. is dealt as NOP instruction. Regarding common area and non-common area, refer to the section 5.8. Table 5‑1 shows some attributes of each memory from each master.

Table 5‑1 Memory Attributes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Transferred Memory  Bus Master | LWM | GWM | Scratchpad | Host Memory Space  (External Memory) |
| Host (ARM Core, System DMA) | Cannot access | Cannot access | Can access  - Mapped on ARM virtual memory space  - Un-cachable \*1 | Can access  - Mapped on ARM virtual memory space  - Cachable |
| CVengine Thread | Can access  - Mapped on CVengine memory space  - Un-cachable, but fast.  - Thread dedicated. Cannot be shared with other thread.(non-common access)  - Can be shared with other thread in the same core.(common access)  - Data only. CVengine instruction cannot be loaded  - General CVengine load / store instructions are used.  - 8-bit, 16-bit, 32-bit access are available. For V3M Ver.2.0 / V3H and V3U, 64-bit access is also available. | Can access  - Mapped on CVengine memory space  - Un-cachable, but fast.  - Thread dedicated. Cannot be shared with other thread.(non-common access)  - Can be shared with other thread in the same cluster.(common access)  - Data only. CVengine instruction cannot be loaded  - General CVengine load / store instructions are used.  - 8-bit, 16-bit, 32-bit access are available. | Can access  - Mapped on CVengine memory space  - Cacahable  - Both data and CVengine instruction are loaded  - General CVengine load / store instructions are used.  - 8-bit, 16-bit, 32-bit access are available.  - Can be shared among threads and clusters  - IMP Core can access | Can access  - Mapped on CVengine memory space.  - Cachable  - Both data and CVengine instruction are loaded  - General CVengine load / store instructions are used.  - 8-bit, 16-bit, 32-bit access are available.  - Dedicated instructions eg. LDRXY/STRXY are also available to access the data. |
| CVengine DMA | Can access  - Master thread / program can use this DMA.  - ARM cannot use this DMA.  - Not through cache. Physical memory to memory. | Can access  - CVengine thread / program cannot use this DMA.  - ARM can use this DMA.  - Not through cache. Physical memory to memory. | Can access  - Not through cache. Physical memory to memory. | Can access  - Not through cache. Physical memory to memory. |
| SBO | Cannot access | Cannot access | Can access  - Write only | Can access  - Write only |

\*1 ARM can access the scratchpad memory with 4byte width only. Accessing the scratchpad memory via ARM data cache is not supported.

## Constant Memory Area

This memory area is for storing uniform storage data (constants that can be read from the CVengine thread). CVengine has 64 entries and each entry has 32-bit long and is addressed in a byte unit.

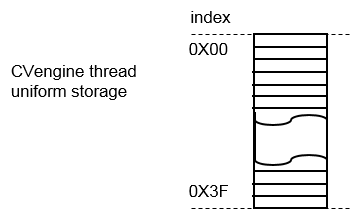


Figure 5‑4 Address Space for Constant Memory Area

The uniform storage area can be accessed by the dedicated instructions like MOVU, ORU, ADDU with an index. The uniform storage can always be accessed by these instructions without penalty cycles.

In addition, general memory load instruction LD is available to read the uniform storage area. This general load instruction can access the constant memory area described in the section 5.1 by word, half word and byte unit.

One note is, the dedicated instructions e.g. MOVU specifies an address on this area by “index”. Namely address is in a word unit. However the general load/store instructions specify an address by “byte address” as same as other memory areas.

## Local Working Memory Area

The local working memory size is total 16 Kbytes (V3M/V3H) / 32 Kbytes (V3U) for all 8 threads and it has a 32-bit width memory area. This memory is not cached. There are common and non-common accessing modes to this memory. Refer to section 5.8. In non-common mode this memory cannot be accessed from other threads, so it can be used as temporal data storage by each thread of CVengine. Each thread can access this local working memory via same memory window, but actual physical memory is dedicated to each thread. See Figure 5‑5 for this mapping. In common-access mode, all the threads in a core share one memory space. Accessing LWM in another core is impossible.

The local working memory can be accessed by general load/store instructions like LD/ST.

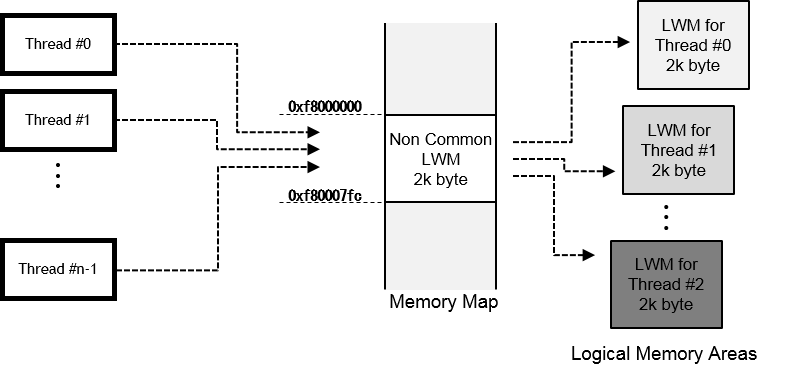


Figure 5‑5 Local Working Memory Space

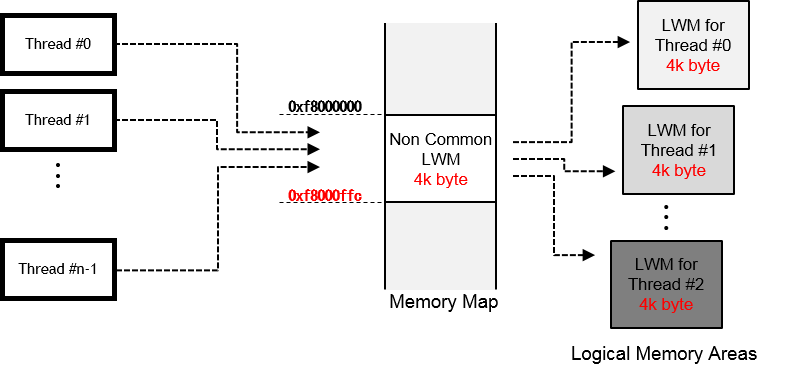


Figure 5‑6 Local Working Memory Space (V3U)

## Global Working Memory Area

The global working memory is an area that can be accessed by all threads of CVengine. This memory is not cached. There are common and non-common accessing modes to this memory. Refer to section 5.8. The global working memory of the CVengine is composed of a 4-Kbyte (V3M/V3H) / 8-Kbyte (V3U) data area and two types of special registers (semaphore register and increment register).

The data area is a 32-bit area with one entry, to which values can be read from or written to freely. When data coherency needs to be guaranteed, perform synchronization in the program.

The global working memory can be accessed by general load/store instructions like LD/ST.

## Semaphore and Increment Registers Area

This common area is divided into 2 parts: semaphore and increment areas. After accessing these common resources, several NOP must be inserted to release the resource. For V3M Ver.2.0, V3H and V3U, no NOP is required. These register area must be accessed with a word (32-bit) access instructions. Accessing these areas with instructions in 16-bit or 8-bit length is prohibited.

### Semaphore Register Area

The semaphore register area is from 0xF0010000 to 0xF001007C. There are 32 32-bit long semaphore registers.

The semaphore register holds only 0 or 1 and it can be read from and written to by all threads. When the semaphore register is read, the register value is forcibly changed to 1 after the read operation.

This feature can be used to control accesses to shared resources. Each register is assigned the state of a resource. 0 indicates that the resource is available, whereas 1 indicates that the resource is not available. During program execution, as long as the relevant semaphore register value is 1, a wait routine to wait for the semaphore register value to become 0 is executed. When the value has changed to 0 and is read, perform the processing using that resource and write 0 to the relevant semaphore register at the end. This allows another thread that was waiting to use that resource to occupy the resource.

Operation is not guaranteed when a value other than 0 or 0.0f is written to the semaphore register.

For V3M Ver.2.0, V3H and V3U, the semaphore register area is from 0xF0010200 to 0xF00103FC. There are 128 32-bit long semaphore registers. The first 32 of the 128 semaphores can also access with the address from 0xF0010000 to 0xF001007C, which is the same to V3M.

For V3U, the non-blocking (read without blocking) semaphore register area is from 0xF0010200 to 0xF00103FC, same as V3M Ver.2.0 and V3H. Additionally, a blocking (read with blocking) semaphore area is available from 0xF0011000 till 0xF001107C.

Same as the non-blocking semaphore register, the blocking semaphore register also holds only 0 or 1. Each blocking semaphore register is assigned the state of a resource. 0 indicates that the resource is available, whereas 1 indicates that the resource is not available. When the blocking semaphore register is read, the register value is forcibly changed to 1 after the read operation. Only 1 thread is able to read the blocking semaphore register at one time. While the register value is 1, any thread trying to read the blocking semaphore register stalls so that the read value of the blocking semaphore is always 0. When the register value is changed from 1 to 0, the stalling thread restarts and read the register value again. Only 0 can be written to the blocking semaphore register regardless of the holding value the register. Unlike the non-blocking semaphore registers explained above, waiting loop is not needed to wait for the register value becoming 0 to control resources. The round robin arbitration scheme is used to control the threads accessing to the blocking semaphore register. Writing access has higher priority than reading access.

After reading blocking semaphore registers 1 NOP must be inserted before next instruction, whereas after writing no NOP is needed. This NOP is required regardless of the NOP described at the beginning of this section 5.5. Generally compiler for CVengine does not insert this NOP automatically. Inserting this NOP is user responsible.

**Note:** For non-blocking semaphore, value is returned regardless of successful acquisition of a semaphore. Because of the hardware implementation, a semaphore may not be acquired forever depending on access timings from threads. This is an example case of deadlock. When the value of a shared variable is changed after acquisition of a semaphore, and if both reading and writing from/to the shared variable use the same semaphore, reading from the shared variable by a thread can disturb writing to the shared variable from another thread so that deadlock can happen. To avoid this kind of deadlock, logic must be designed carefully. For example, only the thread writing to the shared variable uses the semaphore.

### Increment Register Area

The increment register area is from 0xF0010080 to 0xF001009C. There are 8 32-bit long increment registers.

The increment register is a 32-bit register in the integer format and it can be read from and written to by all threads.

When the increment register is read, the register value is incremented by 1 after the read operation. This feature can be used by each thread to acquire unique serial numbers that are consecutive.

The default value of the increment register is 0. User can write 0 to reset the increment register. Any other value can also be written to the increment register.

For V3U, a second access area for the increment registers exists for read access from 0xF0010180 to 0xF001019C. When using this access area for a read access, the register value is read without incrementing the value by 1. A write to this access area is not allowed.

## Scratchpad Memory Area

CVengine has a 896kB scratchpad memory area which is the former “IMPC” and accessible by all threads of all clusters. This is useful to exchange data with any external core like IMP Core or ARM without using DDR memory. There are common and non-common accessing modes to this memory. For V3H, scratchpad memory area is 2048kB. For V3U, scratchpad memory area is 16MB. The CVengine memory map prepares 16MB area for scratchpad, the mapping to the real memory is decided by the bus system which is outside of the scope of CVengine.

To activate non-common access to the scratchpad memory, write the value 0xff7e0000 to the address 0xffa00404 first. Note that only ARM CPU can write to the address. Vertex Issue cannot write to the address with WPR command in Command List. This activation is required only for V3M. This is NOT required for V3M Ver.2.0, V3H and V3U.

## Queue Arbiter Register Area

This area has some configuration registers which are read/written by each thread to control its behavior. TGDMCR0, TGDMEXTMEMOFSADDR, TGDMLWMBADDR, TGDMLWMSTRR, TGLNGR, TGSBOCNTR0 are implemented to configure TGDMAC “Thread Group DMAC” and start/stop DMA transfer between external memory and local working memories.

The start address of this area is 0xf001c780 and the end address is 0xf001c7fc.

These registers are so called “non-common”, namely each thread has his own register set which cannot be accessed by other thread. And they must be accessed by word (32-bit) unit. Half word (16-bit) and byte access are prohibited.

## Common and Non Common Area

### Non-Common Access to LWM

Some memory areas have two memory regions. One is non-common area. Another is common area. The non-common area provides one virtual area on the memory map. Through this area, each thread can access a physical memory “bank” dedicated to each thread. This non-common access is typically used in the case of local working memory.

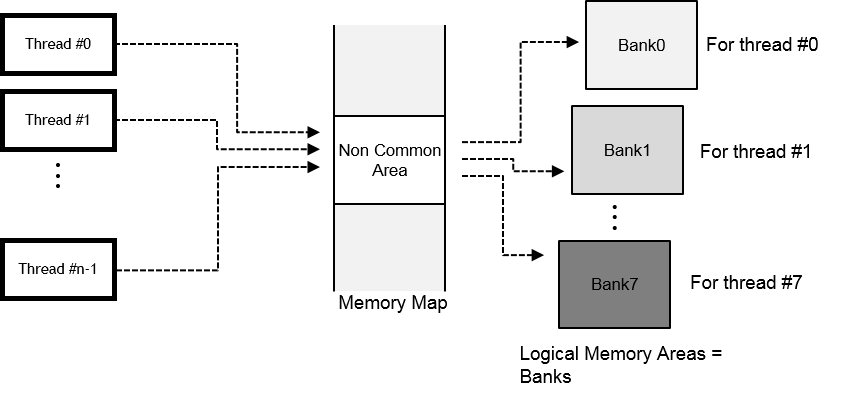


Figure 5‑7 Non-common Access to LWM

### Common Access to LWM

The common access provides capability to access all banks from any thread. For example, through common area, a thread can access the LWM owned by other threads. However each bank is not mapped on this common area continuously. A bank appears on the common area every 32 byte (4byte × 8threads). The Figure 5‑8 shows this mapping for LWM case. A thread can access any banks of other thread, but software should have responsible to manage memory consistency among all threads.

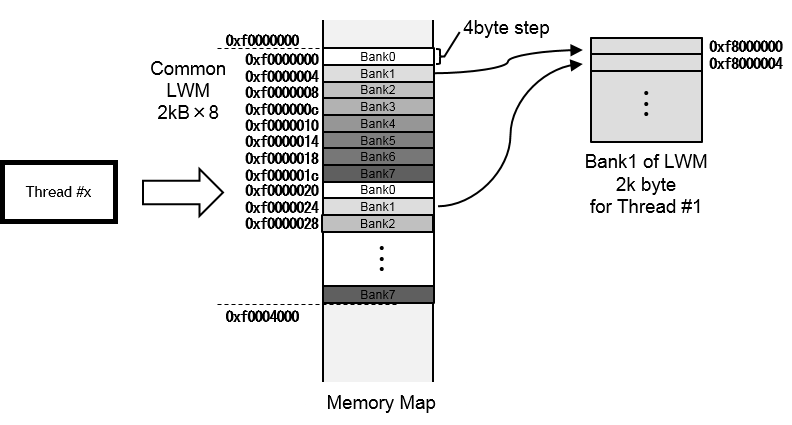


Figure 5‑8 Common Access to LWM

For V3M Ver.2.0, V3H and V3U, a bank appears on the common area every 64 byte (8byte × 8threads), because of the 64-bit access instructions to LWM. The Figure 5‑9 shows this mapping for LWM case. This 8 byte access is only to LWM. Access to other memories is in 4 byte.

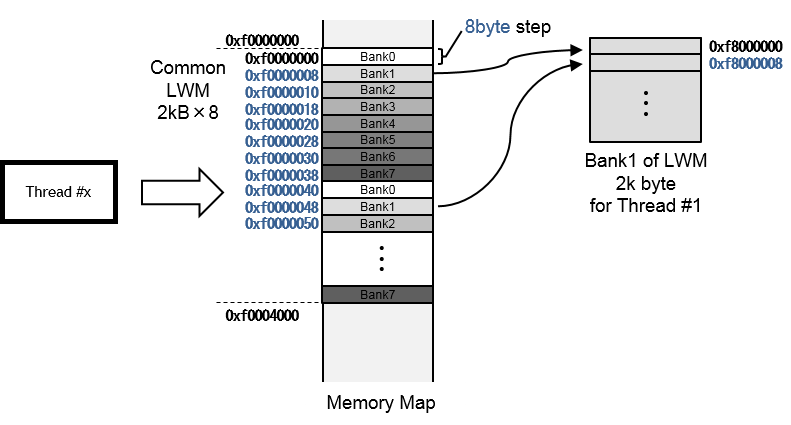


Figure 5‑9 Common Access to LWM (V3M Ver.2.0 / V3H)

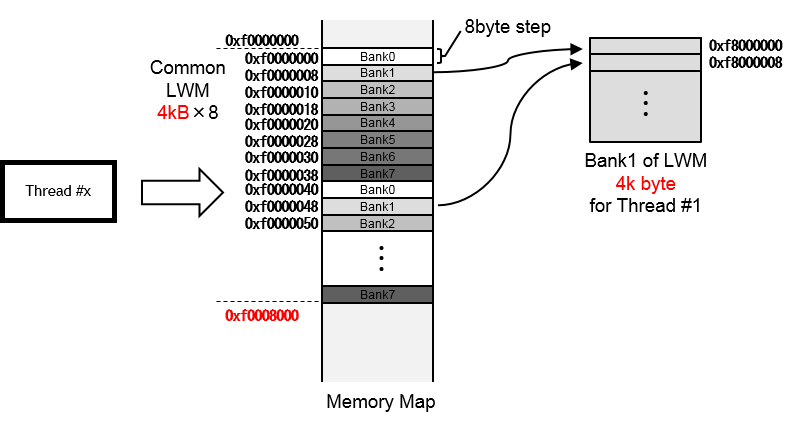


Figure 5‑10 Common Access to LWM (V3U)

### Access Collision to LWM Common Area

When multiple threads access to the same LWM bank at the same time in common mode, lower prioritized thread stalls until access is ready. This stall happens both reading and writing accesses. For the LWM bank structure in detail, refer to section 5.8.2.

For V3U, 8bit, 16bit, 32bit and 64bit LD read accesses to LWM in common mode, a data sharing is implemented so that if multiple threads are reading the same memory address in the same clock cycle, the read data is sent to all these requesting threads. For reading access no stall happens. For writing access stall still happens.

### Common and Non-Common Access to GWM

GWM can be also accessed in either common or non-common mode. GWM is designed to be used in common mode access in general, so that threads can share one global memory area. But if necessary, GWM can be accessed in non-common mode, threads can have slightly larger dedicated memory than LWM only.

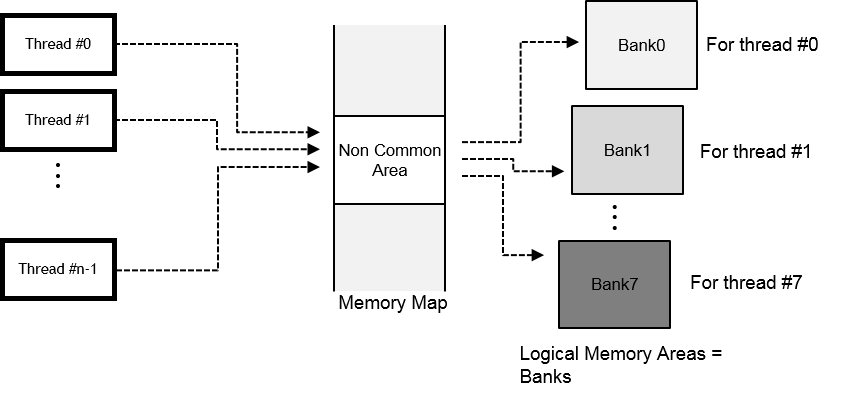


Figure 5‑11 Non-common Access to GWM

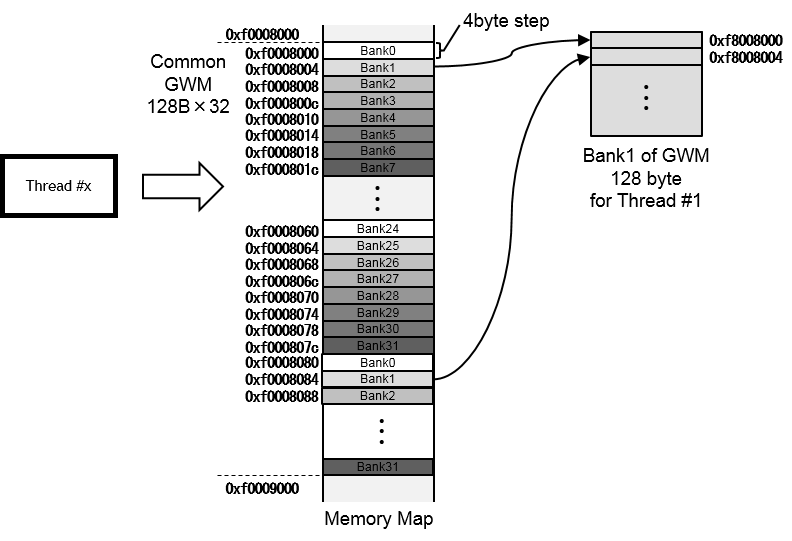


Figure 5‑12 Common Access to GWM

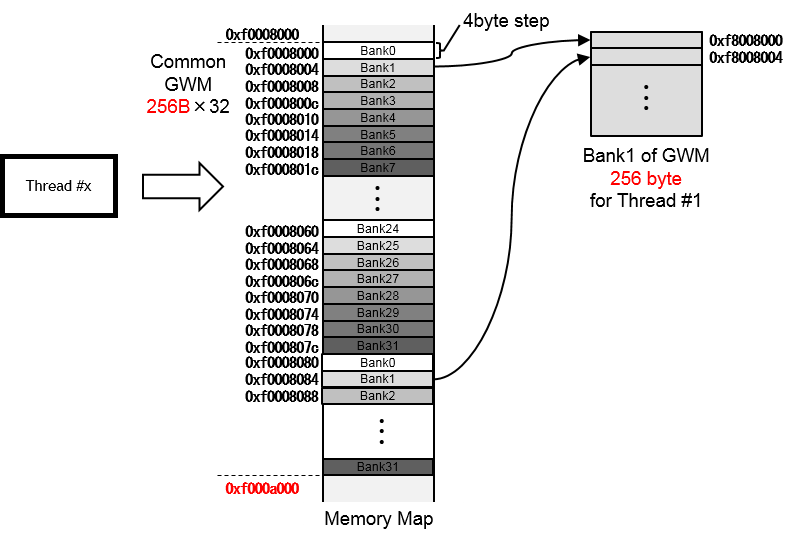


Figure 5‑13 Common Access to GWM (V3U)

### Common and Non-Common Access to Scratchpad

Scratchpad can be also accessed either in common or non-common mode.

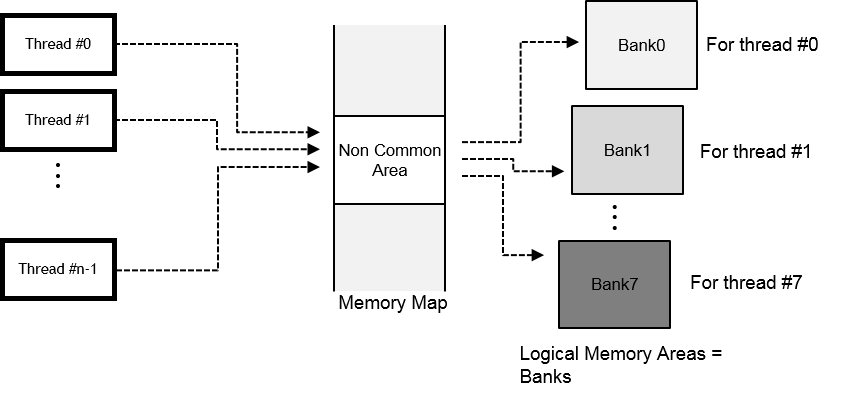


Figure 5‑14 Non-common Access to Scratchpad

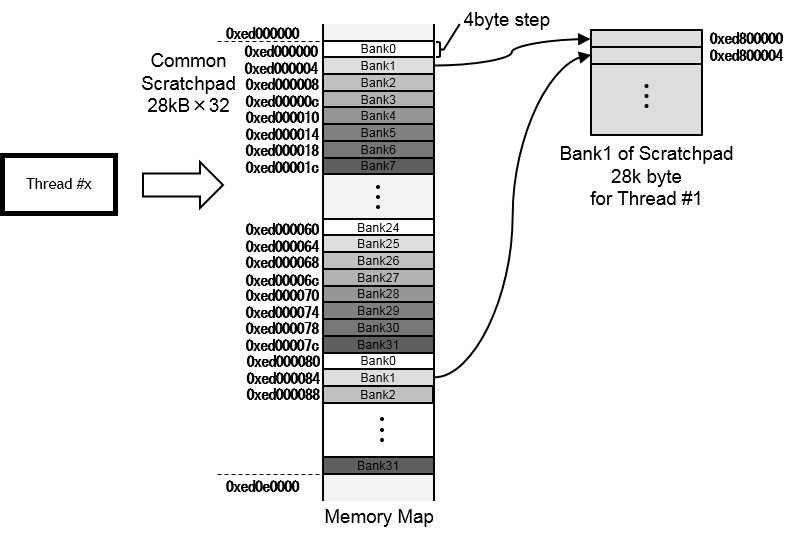


Figure 5‑15 Common Access to Scratchpad

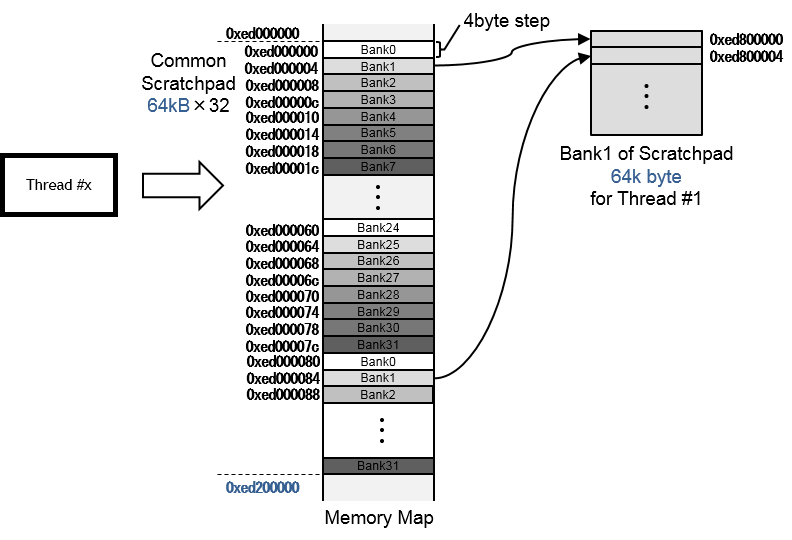


Figure 5‑16 Common Access to Scratchpad (V3M Ver2.0 / V3H)

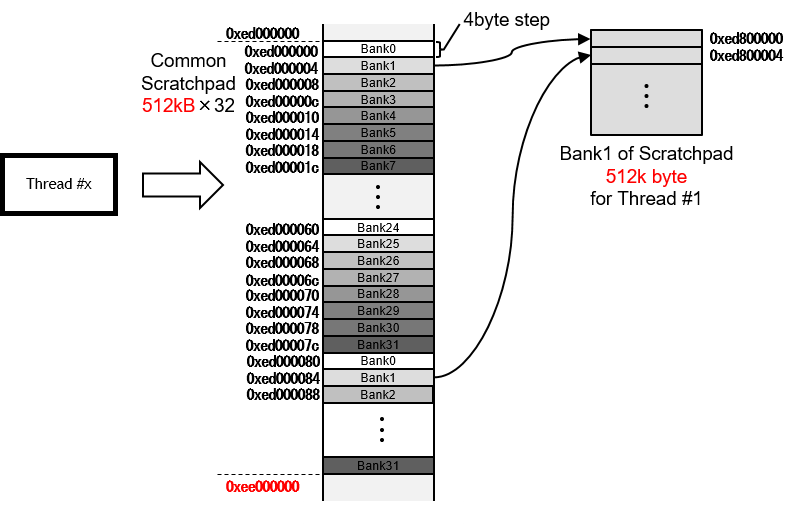


Figure 5‑17 Common Access to Scratchpad (V3U)

# CVengine Thread Registers

## List of CVengine Thread Registers

The CVengine thread has 32 registers with 32-bit width.

Registers are classified into input registers, output registers, address registers, and constant registers, each of which has special features. Also, there is one status register (SR) which has flags that are set to 1 or 0 depending on the results of compare instructions.

R0 and R1 Input registers in which X and Y coordinate information is set

Inter-Thread Register to pass a value by ACTST from master thread to slave thread

ZERO Reading this register returns always zero

R4/R5 Reserved(V3M). General purpose registers (V3M Ver.2.0, V3H and V3U)

R6 Reserved for Renesas internal use only. The register to specify an index address for use in access to the uniform storage. For V3M Ver.2.0, V3H and V3U, this register can be used as a general purpose register. Refer to Section 9 Note 4 for detail.

R7 General purpose register

R8 Register to calculate the sine of the written value

R9 Register to calculate the cosine of the written value

R10 Register to calculate the inverse of square root of the written value

R11 Register to calculate the exponent of the written value with 2 as the base

R12 Register to calculate the logarithm of the written value with 2 as the base

R13 Register to calculate the square root of the written value

R14 and R15 Registers to calculate the reciprocal of the written value. Independent values can be written in these two registers, and the reciprocal operations are executed in pipeline.

R16 to R31 General purpose registers

R32 to R47 General purpose registers (V3U)

SR Status register to which the results of compare instructions are set

For V3M Ver.2.0, V3H and V3U, R8 to R15 can be used as general purpose registers. Refer to Section 6.2 for detail.

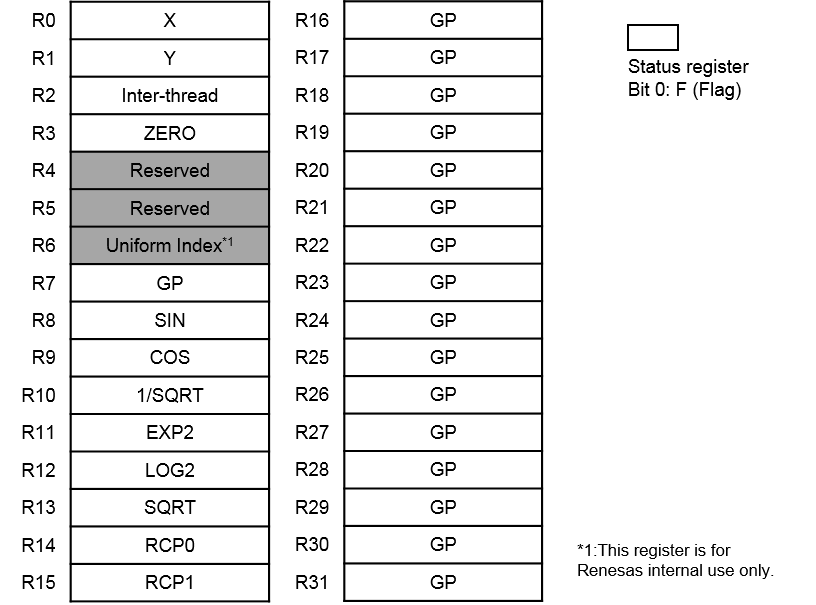


Figure 6‑1 Register Files of CVengine Thread (V3M)

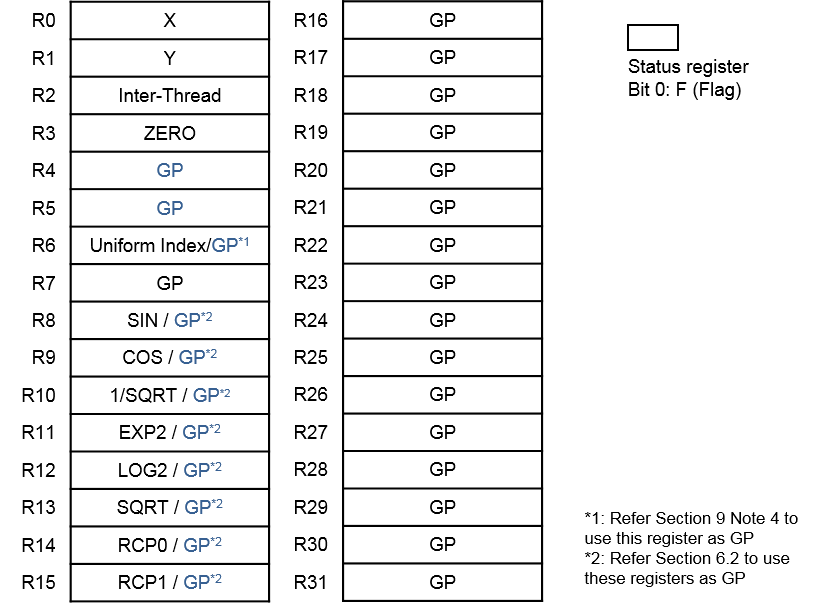


Figure 6‑2 Register Files of CVengine Thread (V3M Ver.2.0 / V3H)

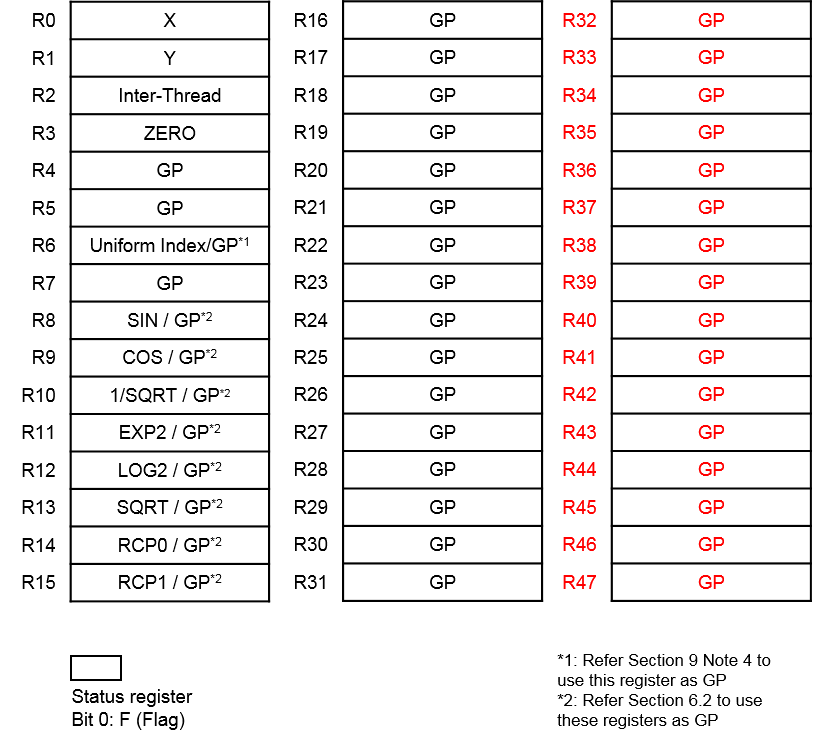


Figure 6‑3 Register Files of CVengine Thread (V3U)

The following subsections show details of the CVengine thread registers

### Coordinate Input Registers (R0 and R1)

These are registers to which the coordinates are input. The X coordinate is input to R0 and the Y coordinate is input to R1. These coordinate values are used when the coordinates are output to the SBO port. The coordinate information can be read as well as changed.

The coordinates to be input are the values of the X and Y coordinates issued by the vertex issue.

R0 and R1 can be modified by the thread.

### Inter-Thread Register (R2)

This register is for a slave thread to receive a 32-bit small information from a master thread. The master thread specifies the register having the value in an operand of ACTST. The slave thread can get the value in its register R2.

Software can use this register as one of general purpose register, if confliction by ACTST writing is managed by software accordingly.

### Zero Register (R3)

When CVengine program reads this register, 0 “zero” is always returned. Writing any value to this register is ignored.

### Uniform Index Register (R6)

**This register is for Renesas internal use only. A user software shall not write any value to this register (V3M).**

This register is used to specify an index (address offset in word unit) for use in access to uniform storage.

An instruction uses the value obtained by adding the index specified in this register to the address specified in the instruction as the address for access.

The value of this register is automatically cleared to 0 each time a thread is started.

For V3M Ver.2.0, V3H and V3U, this register can be used as a general purpose register. Refer to Section 9 Note 4 for detail.

### Sine Operation Register (R8)

This is a register to calculate the sine.

When a value (floating point) is written to this register, input value is multiplied by pi automatically in the hardware, then sine is calculated from the value, namely compute sin(pi\*x). Upon completion of operation, the result is written back to this register.

If this register is read from an instruction as a source operand before operation has completed, the thread stalls.

If this register is read before being written, the result is undefined

For V3M Ver.2.0, V3H and V3U, this register can be used as a general purpose register. Refer to Section 6.2 for detail.

### Cosine Operation Register (R9)

This is a register to calculate the cosine.

When a value (floating point) is written to this register, input value is multiplied by pi automatically in the hardware, then cosine is calculated from the value, namely compute cos(pi\*x). Upon completion of operation, the result is written back to this register.

If this register is read from an instruction as a source operand before operation has completed, the thread stalls.

If this register is read before being written, the result is undefined

For V3M Ver.2.0, V3H and V3U, this register can be used as a general purpose register. Refer to Section 6.2 for detail.

### Inverse of Square Root Operation Register (R10)

This is a register to calculate the inverse of square root.

When a value (floating point) is written to this register, square root and inverse operation are executed with the value used as an operand. Upon completion of operation, the result is written back to this register. If a negative value is specified as an input, NaN (exponent=0xff, fraction!=0) is returned.

If this register is read from an instruction as a source operand before operation has completed, the thread stalls.

If this register is read before being written, the result is undefined

For V3M Ver.2.0, V3H and V3U, this register can be used as a general purpose register. Refer to Section 6.2 for detail.

### Exponent Operation Register (R11)

This is a register to calculate the exponent with 2 as the base.

When a value (floating point) is written to this register, exponent operation with base 2 is executed with the value used as an operand. Upon completion of operation, the result is written back to this register.

If this register is read from an instruction as a source operand before operation has completed, the thread stalls.

If this register is read before being written, the result is undefined

For V3M Ver.2.0, V3H and V3U, this register can be used as a general purpose register. Refer to Section 6.2 for detail.

### Logarithm Operation Register (R12)

This is a register to calculate the logarithm with 2 as the base.

When a value (floating point) is written to this register, logarithm operation with base 2 is executed with the value used as an operand. Upon completion of operation, the result is written back to this register.

If this register is read from an instruction as a source operand before operation has completed, the thread stalls.

If this register is read before being written, the result is undefined

For V3M Ver.2.0, V3H and V3U, this register can be used as a general purpose register. Refer to Section 6.2 for detail.

### Square Root Register (R13)

This is a register to calculate the square root.

When a value (floating point) is written to this register, square root operation is executed with the value used as an operand. Upon completion of operation, the result is written back to this register. If a negative value is specified as an input, NaN (exponent=0xff, fraction≠0) is returned.

If this register is read from an instruction as a source operand before operation has completed, the thread stalls.

If this register is read before being written, the result is undefined

For V3M Ver.2.0, V3H and V3U, this register can be used as a general purpose register. Refer to Section 6.2 for detail.

### Reciprocal Operation Registers (R14 and R15)

These are registers to calculate the reciprocal.

When a value (floating point) is written to either of these registers, reciprocal operation is executed with the value used as an operand. Upon completion of operation, the result is written back to that register.

If the register is read from an instruction as a source operand before operation has completed, the thread stalls.

If these registers are read before being written, the result is undefined

RCP0 (R14) and RCP1 (R15) are both registers to calculate reciprocal in the same manner. Since parallel processing of these registers is possible, the required number of cycles can be hidden by operating them in parallel when two reciprocals are needed.

Rounding mode of the registers is fixed to “Round to the nearest, ties to even”.

For V3M Ver.2.0, V3H and V3U, these registers can be used as a general purpose register. Refer to Section 6.2 for detail.

Note: RCP0 and RCP1 share the same reciprocal arithmetic unit. Therefore, if it is used frequently by multiple threads, contention will occur frequently and it may not be possible to successfully hide the number of cycles.

### R4, R5, R7, R16 to R47 and 64-bit Pair Register

These are general-purpose and temporary registers (readable and writable) with 32-bit width. Initial value of these registers is undefined. R4 and R5 are available only for V3M Ver.2.0, V3H and V3U.

In addition, some instructions handling 64-bit width data utilize even of two consequent general purpose registers with little endian. For example, "MULLS R16, R30, R31" means lower 32-bit of 64-bit result is stored in R16, and upper 32-bit is stored in R17. And when specify an odd register as 64-bit register, result of execution is not guaranteed. Only R16 to R47 can be used for the 64-bit pair registers.

Further, the new small register block load/store instructions to local work memory LDSRB, STSRB, LDSRBI, STSRBI, LDSRBC and STSRBC have the possibility to access the registers R16 to R23 by specifying R16, R24 to R31 by specifying R24, R32 to R39 by specifying R32 and R40 to R47 by specifying R40, while the new large register block load/store instructions to local work memory LDLRB, STLRB, LDLRBI, STLRBI, LDLRBC and STLRBC have the possibility to access the registers R16 to R31 by specifying R16 and the registers R32 to R47 by specifying R32.

### Status Register

This register has bits that are set to 0 or 1 by the results of compare instructions.

This register is used to determine whether to branch or conditionally execute instructions. Refer to Section 0.

This register cannot be read as an operand by instructions.

## Control Registers of CVengine Thread

The CVengine thread has control registers (CR) to control operation.

The control registers can be changed by the STC instruction or STCI instruction and read by the LDC instruction.

An integer value has to be set to the control registers.

These registers are initialized only upon a software reset. Refer to section 12.1.2.

CR0 When read, the core number (ID) is read. This register cannot be rewritten.

CR1 When read, the thread number (ID) is read. This register cannot be rewritten.

CR2 to CR7 Control registers that have not been implemented. Writing these registers is prohibited.

CR8 This register specifies the data plane number to be read via the D$.

The initial value is undefined.

CR9 This register specifies the data plane number to be written via the D$.

The initial value is undefined.

CR10 This register specifies the output mask for each element of SBO.

Bit 2, bit 1 and bit 0 correspond to R0 (X coordinate), R1 (Y coordinate) and D0 (data) respectively. When each bit is set to 1, the corresponding element is not output.

For example, when bit 2 = 0, bit 1 = 1, and bit 0 = 1, only R0 (X coordinate) is output from the SBO port.

CR11 Control register that have not been implemented. Rewriting this register is prohibited.

CR12 Control register that have not been implemented. Rewriting this register is prohibited.

For V3U, this register holds offset number from thread’s own TGDMAC channel. When this register holds value m, thread #n accesses TGDMAC channel #((n+m)%8). Refer to section 8.3. User can change the value of this register freely, but avoiding resource conflict is user responsibility.

CR13 to CR15 Control registers that have not been implemented. Rewriting these registers is prohibited.

CR16 This register specifies the rounding mode of floating-point operations. The initial value is 0.

0: Round to the nearest, ties to even

1: Round toward zero

This setting specifies the behavior of addition or multiplication performed by an instruction and also the behavior of conversion performed by the I2F instruction.

When changing value of this register, 4 or more cycles must have passed from the last floating-point operation. Or insert 4 NOPs to clear pipeline before changing its value.

CR17 This register specifies the rounding mode of conversion between floating-point and integer. The initial value is 1.

0: Round to the nearest, ties to even

1: Round toward zero

This setting specifies the behavior of conversion performed by the F2I instruction.

When changing value of this register, 4 or more cycles must have passed from the last F2I instruction. Or insert 4 NOPs to clear pipeline before changing its value.

CR18 This register specifies the rounding mode to be used in casting by the FFRC and FFLR instructions. The initial value is 1.

Changing the value from the initial value leads to the instruction not operating in the manner of the operating model. Accordingly, do not change the value unless this is essential.

0: Round to the nearest, ties to even

1: Round toward zero

When changing value of this register, 4 or more cycles must have passed from the last FFRC or FFLR instruction. Or insert 4 NOPs to clear pipeline before changing its value.

CR19 Writing 1 to bit-n of this register turns Rn to the general purpose register (n=8-15). This register is available only for V3M Ver.2.0, V3H and V3U.

Reading this register shows current mode of each of R8 to R15. The default mode is SOU for all the registers.

0: SOU

1: General purpose register

When changing each bit of this register, 4 or more cycles must have passed from the last corresponding SOU operation to the bit. Or the result of SOU operation must be used as an operand of an instruction before changing the corresponding bit. (Because pipeline stalls until result of SOU operation becomes ready.) Or insert 4 NOPs to clear pipeline before changing a bit.

CR20 Writing 1 to bit-n of this register turns Rn to the SOU register (n=8-15). This register is available only for V3M Ver.2.0, V3H and V3U.

This register is write-only. Behavior of reading this register is undefined.

When value of this register is changed, corresponding SOU operation must be executed 4 or more cycles after the changing instruction. Or insert 4 NOPs to clear pipeline after changing its value.

# Floating-Point

## Floating-Point Data Format

The CVengine thread handles 32-bit floating-point values. The floating-point data format is shown below.



Figure 7‑1 32-bit Floating-Point Data Format

* S (sign): Indicates the sign.   
  0: positive value or 0/1: negative value or 0
* Exp (exponent): Indicates the N-th power of 2 as a bias.   
  H'7F is the 0th power of 2, H'80 is the 1st power of 2, H'7E is the −1st power of 2,  
  H'FF is infinite and H'00 is 0.

Fraction: Indicates the fraction (a total of 24 bits with an implicit one bit added is regarded as the fraction part).

## IEEE754 Conformance

The CVengine is partially conformant to IEEE754 in floating-point number calculation accuracy.

Table 7‑1 IEEE754 Conformance

|  |  |  |
| --- | --- | --- |
| Operation | Bit Accuracy Compliancy to IEEE754, i.e. < 0.5ulp | IEEE754 Compliancy except  accuracy, e.g. handling NaN,  denormal, exception, etc. |
| Float to Integer (F2I) | Comply | Comply |
| Integer to Float (I2F) | Comply | Comply |
| Addition, Subtraction (FADD, etc.) | Comply | Comply |
| Multiplication (FMUL, etc.) | Comply | Comply |
| Sine (R8) | maximum 2^(-23) | Comply |
| Cosine (R9) | maximum 2^(-23) | Comply |
| Inverse Square Root (R10) | 1ulp | Comply |
| Exponent (R11) | 1ulp | Comply |
| Logarithm (R12) | maximum 2^(-23) | Comply |
| Square Root (R13) | 1ulp | Comply |
| Reciprocal (R14, R15) | 1ulp | Comply |
| Comparison (FCMP, etc.) | N/A | Comply |

# Peripheral Functions

## Streaming Buffer Output (SBO)

Executing SBOCRD instructions leads to the writing of an X coordinate, Y coordinate, and value of a specified register to consecutive areas of the main memory via the streaming buffer output (SBO) port. Masking can be specified individually for the output of the X coordinate, Y coordinate, and specified register value. When masking is specified, only unmasked data is written to consecutive areas. The X and Y coordinates to be output are the X and Y coordinates specified in R0 and R1 with the #IMM1 value added to X and the #IMM2 value added to Y and the data format converted into the integer format. The register value is output as the 32-bit bit string without any change.

The CVengine configuration register SBOBAR0 is used to specify the start address of each SBO. The SBOMMSR0 register is used to specify the maximum size of the sum of data output from each SBO port. When the sum of the output data exceeds this setting, data output is stopped and an interrupt is output.

Control register CR10 is used to specify whether to mask the output of the X coordinate, Y coordinate, and specified register value. Setting examples and the output results after the SBOCRD instruction is executed for n times are shown in the figure below. The CVengine has multiple threads, and thus data output via the SBO port are in the same order as that in which the SBOCRD instructions are issued. Note that the output of data through the SBO is not in the order of activation of the threads.

Note:

SBO buffer size is 128 bytes. So SBO transmits data in 128-byte long each time. User must maintain the data to be transmitted via SBO in 128-byte aligned.



Figure 8‑1 Examples of Data Format Output from the SBO Port (XYPCK=0)

The X coordinate and Y coordinate output can be controlled by the CVengine configuration registers SBOCR0. When the XYPCK bit is set to 1, the lower 16 bits of the X coordinate and Y coordinate which are used when the XYPCK bit is 0 are packed to form 32 bits, then output as the X and Y coordinates. The order of the X coordinate and Y coordinate can be swapped by changing the PCKMODE bit to 0 or 1. When the XYPCK bit is 1 and no mask is specified, the 32 bits output as the X coordinate and the 32 bits output as the Y coordinate are the same value. Setting examples and the output results with those settings are shown in the figure below.

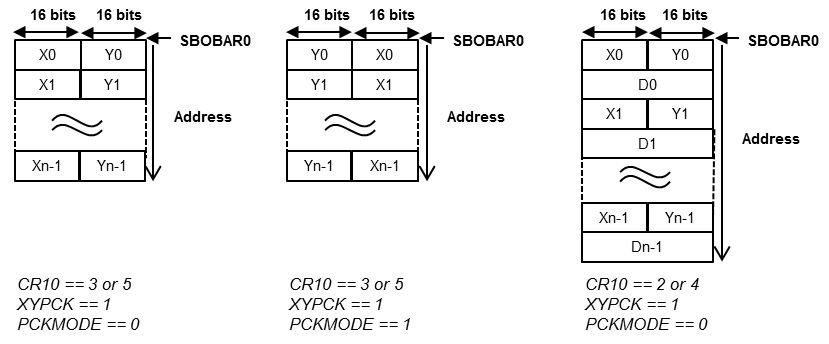


Figure 8‑2 Examples of X and Y Coordinate Packs Output from the SBO Port

## SOU and SAM

CVengine has some shared logic among threads or cores to consider trade-off between chip size and calculation efficiency. The Shared ArithMetic logic unit (SAM) is implemented for each core and it is shared by threads belonging to a core. This SAM is further divided in SAM4, SAM8. The SAM8 unit is shared among all 8 threads of a core. The SAM4 unit consists of SAM4E unit and SAM4O unit. The SAM4E is shared among even threads of a core and the SAM4O is shared odd threads. The following figure shows how these units are shared by the threads.

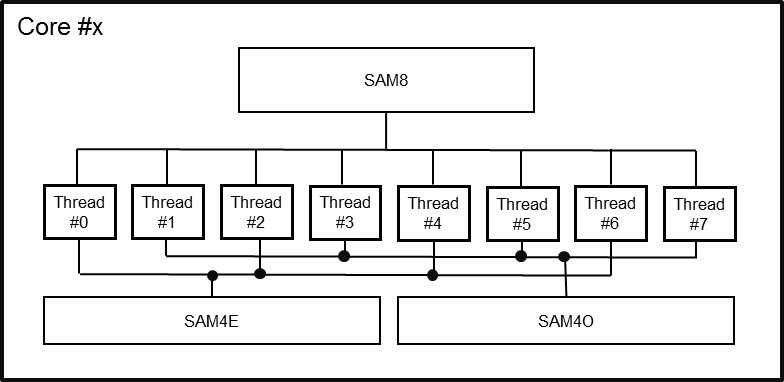


Figure 8‑3 SAM Sharing among Threads (V3M)

For V3M Ver.2.0, V3H and V3U, SAM2 is added. The SAM2 unit consists of SAM2A to SAM2D units. Each SAM2 unit is shared by two threads. The following figure shows how these units are shared by the threads.

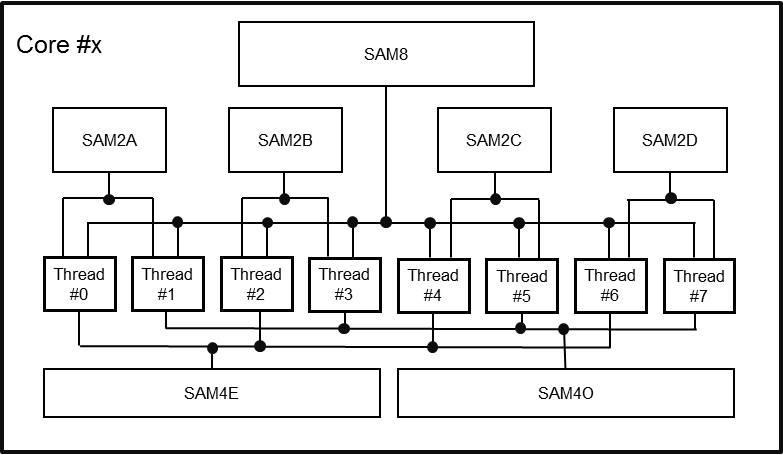


Figure 8‑4 SAM Sharing among Threads (V3M Ver.2.0 / V3H / V3U)

The instructions using the SAM unit are clarified in the Table 10‑2.

On the other hand, the shared operation unit (SOU) is implemented for each cluster and it is shared by all threads in the cluster. The SOU is used through special registers, namely EXP2(R11), LOG2(R12), SQRT(R13), RCP(R14 and R15), 1/SQRT(R10), SIN(R8), COS(R9).

For both SAM and SOU case, the sharing is done by hardware automatically. A thread waits for availability of a SAM/SOU operation, when the SAM/SOU operation is already used by different thread.

## DMA

CVengine supports DMA transfer from/to the common memory regions of LWM/GWM to/from the data plane on the external DDR memory and Scratchpad memory by utilizing TGDMAC “Thread Group DMAC” which can be initiated in CVengine program of any master thread. DMA transfer between LWM and GWM is not supported.

TGDMAC to/from LWM is LWM-TGDMAC which is implemented per 8 threads. And GWM-DMAC is for transfer from/to GWM which is implemented per cluster. Both transfer data with 32 byte unit.

There is one LWM DMA channel per core so there are 4 LWM DMA channels per cluster. Only one DMA channel can transfer data at a time.

Each DMA transfer from/to LWM is configured by TGDMAC configuration registers. This bunch of registers is implemented for each thread and mapped on the queue arbiter register area, so a thread can transfer any data with certain configuration. TGDMCR0 specifies transfer direction, data plane and has control bits to start/stop DMA transfer. Offset or X/Y coordinates of transferred data on the data plane is decided by TGDMEXTMEMOFSADDR Region transferred on LWM is specified by TGDMLWMBADDR as a base address and TGDMLWMSTRR as an image stride. TGLNGR is used to indicate lengths for X and Y direction. See chapter 12.1 for detail description of each register.

Only master thread can read/write LWM-TGDMAC configuration registers and manage DMA transfer. This means only master thread initiate DMA transfer and issue WAITDMA instruction to wait for completion of transfer. When some master threads issue DMA transfer, requests are prioritized by an arbiter which has a simple arbitration policy, thread 0 > thread 1 > … > thread 7.

For V3M Ver.2.0, V3H and V3U, the arbitration policy is changed to the fair arbitration.

Due to the fact that only master threads can initiate a DMA transfer, V3U has the function that enables master thread to control slave thread’s LWM-TGDMAC. The master thread can also access to the slave threads’ LWM-TGDMAC configuration registers to be able to control up to 8 different DMA transfers. To achieve this, CR12 register and DMAWAITS instruction were added to specify DMA transfer for which slave thread to control. Refer to section 6.2 and section 9.

To transfer any data from/to GWM, the GWM-DMAC must be used. GWM-DMAC has the same bunch of registers as LWM-TGDMAC, but these registers are implemented CVengine configuration register area which can be accessed by a command list only, i.e. ARM or thread cannot access these registers.

The following is a typical DMA procedure for LWM-TGDMAC and the Figure 8‑5 is for better understanding of this transfer.

1. Master thread configures DMA through his LWM-TGDMAC configuration registers.
2. Master thread starts DMA transfer by writing 1 to DMACEX bit of TGDMCR0.
3. After starting DMA, the master thread issues WAITDMA instruction, then waits for completion of DMA.
4. After DMA completion, the master thread resumes from WAITDMA, then proceeds next steps like invoking slave threads.

Considering use cases, DMA transfer and thread accessing to LWM should be executed via LWM common area to simplify splitting image area to each thread.

A typical procedure for GWM-DMAC is as follows.

1. Configure GWM-DMAC registers by a command list.
2. A command list starts DMA transfer by writing 1 to DMACEX bit of DMCR0.
3. After starting DMA, the command list issues SYNCS command, then waits for completion of DMA.
4. After DMA completion, the command list resumes from SYNCS, then proceeds next steps like invoking RECT command.

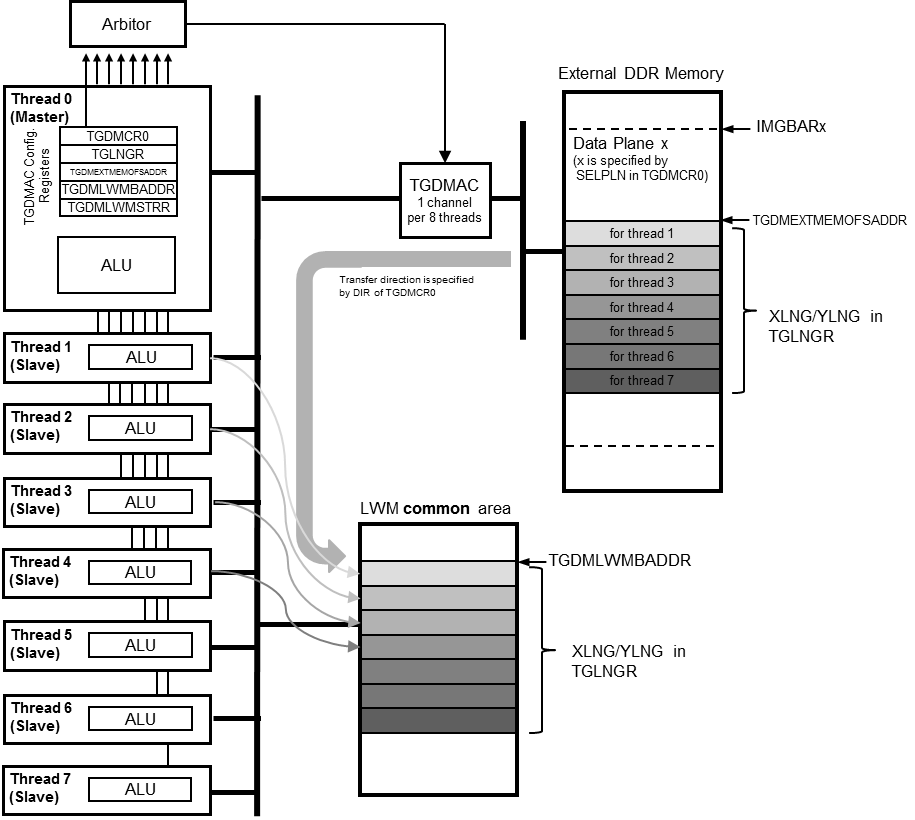


Figure 8‑5 Data Transfer by TGDMAC between LWM and External DDR Memory

## Debug Support

The debug resource is ready on the CVengine with some limitations described below. This debug resource is only available to the dedicated debugger.

1. All clusters have the same debug resources.
2. Only core 0 of each cluster has the debug resource. Other cores do not have.
3. Only thread 0 and thread 1 have the debug resource. Other threads do not have.
4. The two threads of the point 3 aims debugging of master/slave behavior. Thread 0 can be debugged only as master and thread 1 can be debugged only as slave.

## Interrupt

Each thread is able to issue an interrupt to the host processor by the INT instruction. After issuing interrupt, the corresponding bit of status register SR2 configuration register is set to 1. The thread stops to fetch the following instructions and the configuration registers VSPC for master or PSPC for slave show an address of next instruction (PC+1 or an address branched) to be executed. If the corresponding bit of interrupt control register ICR2 is unmasked, the bit13 of SR1 is set to 1. Finally the host processor is interrupted if it is not masked by interrupt mask register IMR1.

After the host processor received an interrupt, the host can check which thread is interrupted via two status registers SR1 and SR2. And the host resumes the disturbed thread by writing 0 to two status clear register SCR1 and SCR2.

See the Figure 8‑6 for better understanding of this interrupt routing. And some remarks should be described as follows:

A corresponding bit of SR2 is set to 1 after previous instruction including load/store and SOU calculation is completed.

* If INT instruction and status clearing take place at the same time, the clearing is prioritized.
* Clearing SR1 does not affect SR2, i.e. the interrupt is retained.
* DMA operation is not affected by the INT instruction, name it works continuously. Software must take care data synchronization transferred by DMA.

**This interrupt and its handshake with the host processor seriously disturb continuous thread processing. Therefore this INT should be used for debugging purpose only.**

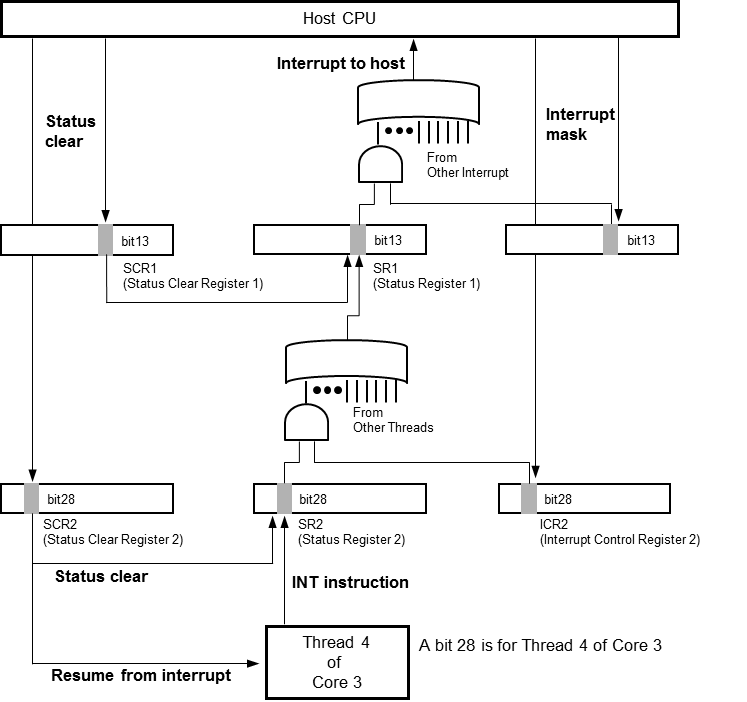


Figure 8‑6 Interrupt from Thread to Host

## Performance Counter

There are two kinds of hardware counters ready to measure performance.

One kind is a hardware counter ready in a cluster to measure performance of the cluster. It counts the clock cycles from the invocation of the threads (POINT/POINTS/RECT command) to the last TRAP instruction. This counter can be enabled or disabled by the settings of configuration registers, SPCR and SPBSYCNTR. Refer to sections 12.1.19 and 12.1.20. This counter does not take care for overflow.

The other kind is to measure performance of each thread. This counter can count stalled cycles of a thread. To use this counter, corresponding configuration registers must be set before invocation of CVengine. The configuration registers TGCR0, USPFCCTLR and PFCTOTALR0 are ready for this counter. Refer to sections 12.1.14, 12.1.50 and 12.1.51. This counter can detect whether overflow happened or not by checking the overflow bit. This counter can count only one thread in a cluster per execution.

**Note:**

Counter values are just for your information. These counters could be useful to improve software performance in some cases, but Renesas does not warrant any relationship between counter values and actual processing time. And Renesas will not open detailed condition of counting.

## Image Clipping

CVengine supports the image clipping feature. This feature allows user to clip the entire image into some necessary part of the image to process. Refer to Figure 8‑7.

To clip the image CLPMINR and CLPMAXR configuration registers of the CVengine are used. Refer to section 12.1.17 and section 12.1.18.

To enable this image clipping feature, set 1 to the clip bit in POINT/POINTS/RECT command. Refer to section 13.1, section 13.2 and section 13.3.

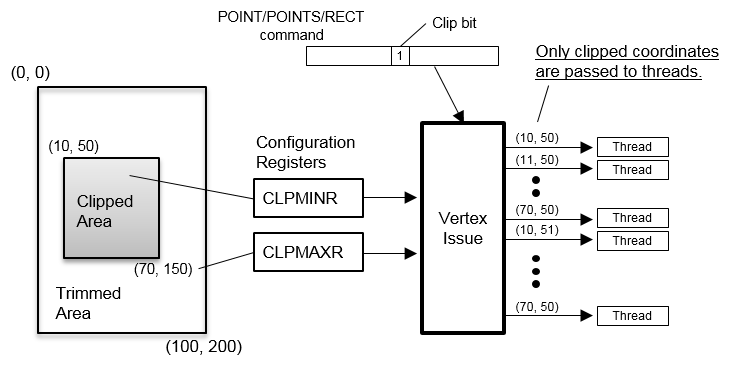


Figure 8‑7 Image Clipping Feature

## Border Padding

**This function is for V3M only. This function is NOT supported for V3M Ver.2.0, V3H and V3U.**

When the thread acquires data from outside of the clipped image area, the data to be acquired is generated as padding data based on the border padding configuration. Refer to Figure 8‑7.

IMGC0Rn (n = 0 to 7), IMGSIZER0 and IMGCNSTR0 configuration registers of the CVengine are used for the border padding configuration. Refer to section 12.1.42 section 12.1.43 and section12.1.44.

The border area is the area which is filled with padding data. The border area is kept around the clipped area, plus and minus of width and height of the clipped area to X and Y axes. The value of the data acquired from outside of the border area is undefined. Refer to Figure 8‑9.

Figure 8‑10 shows how the padding data is generated with the value set to CLPMODE of IMGC0Rn.

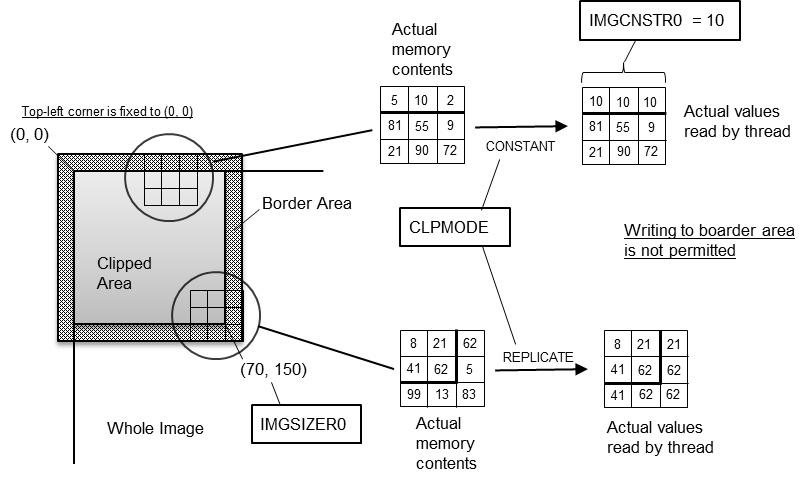


Figure 8‑8 Border Padding Feature

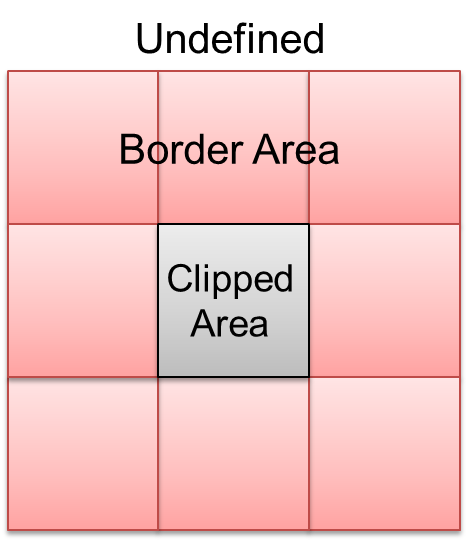


Figure 8‑9 Border Area

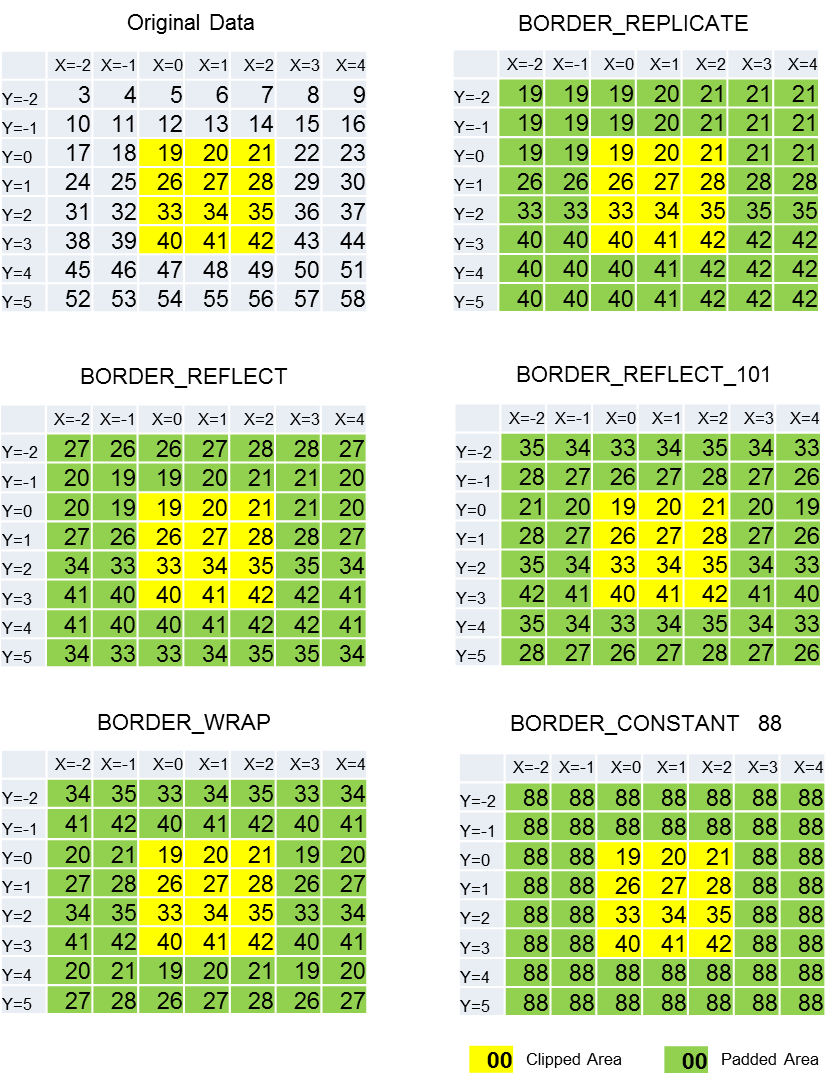


Figure 8‑10 Clipping Mode (CLPMODE)

## Guard Timer

This function is available only for V3M Ver.2.0, V3H and V3U.

The guard timer function is to check if CVengine processing time exceeds expected time. The guard timer is an additional function of the performance counter. So refer to section 8.6 first.

This function can set two types of flags, SR1.PBCOVF and SR1.INT, to show the processing time has exceeded the threshold. SR1.PBCOVF flag is automatically set at the moment when SPBSYCNTR exceeds SPBCTHRR while threads are running. To utilize SR1.INT flag, an INT command with INT\_PBCOVF bit enabled is needed after a CVengine operation command (e.g. RECT) in the Command List. When the INT command in Command List is executed, and if INT.INT\_PBCOVF and SR1.PBCOVF are both 1, SR1.INT flag is set so that the execution of command list is halted and the host CPU can recognize an overflow interrupt has happened. Thus the host CPU can take some action for the processing time overflow like rewriting commands in the Command List. When SCR1.PBCOVFCLR is set to 1, SR1.PBCOVF and SPBSYCNTR are both cleared. Figure 8‑11shows flowchart of this function. Also refer to section 12.1.5, 12.1.6, 12.1.21 and 13.7.

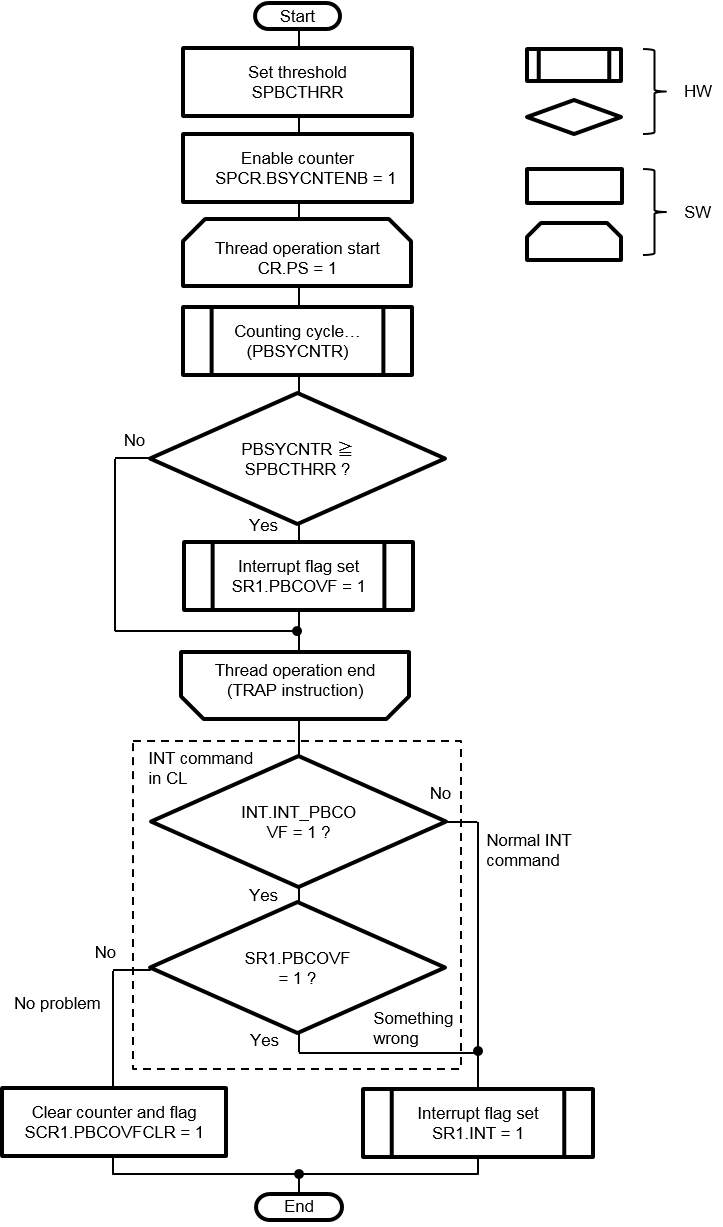


Figure 8‑11 Guard Timer Flow Chart

Figure 8‑12 shows a use case of the Guard Timer. In this case IMR1.PBCOVFMSK is set to 1 so the interrupt during RECT command, SR1.PBCOVF is masked. When counter value exceeds threshold value at the end of the 1st RECT command, the INT command in CL let the host CPU to know an overflow on processing time has happened. So the host CPU alters the threshold value set by WPR command in CL to the new value that is in case for overflow.

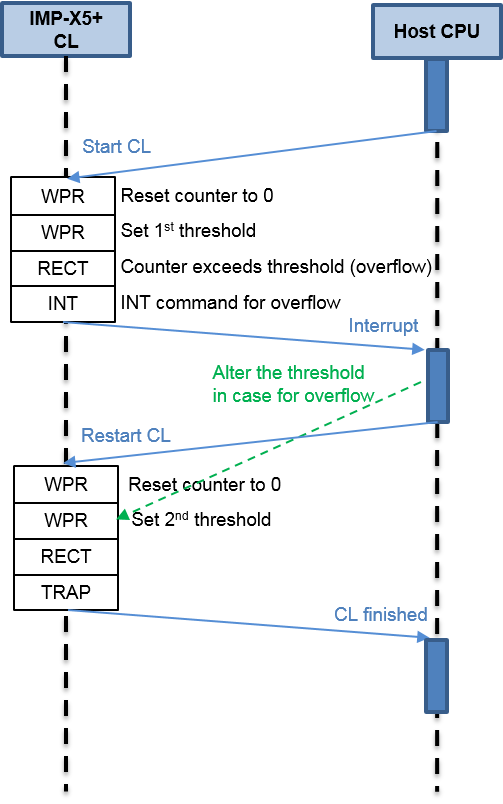


Figure 8‑12 Guard Timer Use Case

## Data Cache Bypassing

Data cache bypassing function is implemented on V3U.

Data cache bypassing area can be set with configuration registers DCBYPA0MIN and DCBYPA0MAX. The default values of the registers are 0xFFFFFFFF and 0x00000000 respectively so no bypassing area is set. Only 4-byte access is supported for D$ bypassing. This feature can be enabled and disabled for each thread with DCBYPEN register. Refer to sections 12.1.22, 12.1.23, and 12.1.24.

Additionally store data replacement feature is implemented. When both DCBYPEN and DCBYPCTRL are enabled for a thread, upper 16 bits of writing data with bypassing D$ by the thread are replaced to Core# and Thread#. Refer to section 12.1.25.

The D$ bypassing function is designed for Renesas debugging solution. Accessing to D$ bypass area is very slow so using this function other than debugging purpose is out of concern.

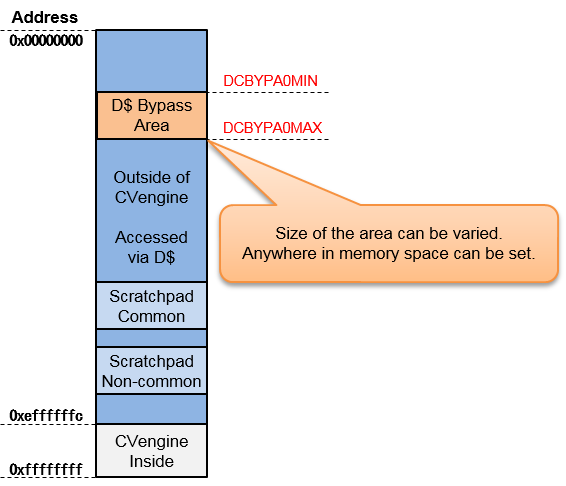


Figure 8‑13 Data Cache Bypassing Area

# Instruction Set

The CVengine thread instructions are classified into the following three types of instructions.

(1) Flow control instructions such as branch instructions

There are branch instructions, compare instructions to generate the branch conditions, and instructions to control the stop of program execution.

(2) Data transfer instructions

(3) Operation instructions

All of the instructions can be executed conditionally with the condition of the status register value being 0 or 1.

**Note1:**

Division instructions are not implemented in the CVengine threads of the CVengine.

**Note 2:**

#ASMIMM is the immediate description for the CVengine Assembler. This #ASMIMM is interpreted by the assembler and is translated to #IMM which is the immediate used in an actual instruction code defined in Section 10.1.

The translation from #ASMIMM to #IMM is defined instruction-by-instruction in the following description.

**Note3:**

For 8-bit and 16-bit group instructions such as GADDxx, GMADxx, GSADxx or GSUBxx, s0 and s1 are treated as 64-bit pair registers. Refer to Section 6.1.12. A pair register contains 4 16-bit data or 8 8-bit data as a data array. For 16-bit group instructions, S0[0] means S0(n)[15:0] and S0[3] means S0(n+1)[31:16]. For 8-bit group instructions, S0[0] means S0(n)[7:0] and S0[7] means S0(n+1)[31:24].

**Note 4:**

For V3M Ver.2.0, V3H and V3U, R6 register can be used either the offset base of immediate value indexing a Uniform storage or a general purpose register. Refer to section 10.1 about the R6 bit of instruction format. To use R6 as the offset base with the CVengine Assembler, add ‘$’ mark to the 3rd operand for the RU type instructions. See the examples below.

ADDU R16, R17, 1 → R16 = R17 + Uniform[1];

ADDU R16, R17, $1 →　 R16 = R17 + Uniform[R6+1];

#R6IMM is used at behavior model in the description of the RU type instructions. If R6 is used as the offset base, the value of #R6IMM is R6 + #IMM. If R6 is used as a general purpose register, the value of #R6IMM is equal to #IMM.

#IMM: 7-bit signed integer

R6: 8-bit signed integer

If the value of R6 + #IMM exceeds the area of uniform storage, result is undefined.

## Conditional instruction execution

For all instructions, it is possible to control the execution/non-execution of the instruction by referencing the status register SR. By writing "SR" at the end of an operand, the instruction is executed if the status register is true, and the instruction is regarded as the NOP instruction if the status register is false. By writing "!SR" at the end of an operand, the instruction is executed if the status register is false, and the instruction is regarded as the NOP instruction if the status register is true.

**Examples:**

CMP.EQ R3, R16, R17

ADD R20, R21, R22, SR # When R16 == R17, execute ADD.

SUB R20, R21, R22, !SR # When R16 != R17, execute SUB.

ABS

**ABSolute**

| Usage | ABS D0, S0 |  | |
| --- | --- | --- | --- |
| Function | Stores the absolute value of S0 in D0.  The value of S0 is handled as an integer in the operation. | | |
| Behavior model | D0 = |(int)S0|; | |  |

ABS8

**ABSolute 8-bit values**

| Usage | ABS8 D0, S0 |
| --- | --- |
| Function | This is an instruction to get the absolute value from 8 signed 8-bit input values from source S0. |
| Behavior model | D0[n] = ABS(S0[n]), n = 0..7; |
| Operand type | D0: 8 times 8-bit unsigned integer  S0: 8 times 8-bit signed integer |

ABS16

**ABSolute 16-bit values**

| Usage | ABS16 D0, S0 |
| --- | --- |
| Function | This is an instruction to get the absolute value from 4 signed 16-bit input values from source S0. |
| Behavior model | D0[n] = ABS(S0[n]), n = 0..3; |
| Operand type | D0: 4 times 16-bit unsigned integer  S0: 4 times 16-bit signed integer |

ACTST

**ACTivate Shader Thread**

| Usage | ACTST S0 |  | |
| --- | --- | --- | --- |
| Function | Master thread activates slave thread. Which slave is going to be activated is determined by queue arbiter. The values of R0 and R1 are sent as the X and Y coordinates. In addition, the register value specified by S0 is sent to R2 of the slave thread.  This instruction must not be executed by the slave thread. | | |
| Behavior model | - | |  |
| Operand type | S0: -(32-bit array) | |  |

ADD

**ADD**

| Usage | ADD D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Adds S0 and S1, then stores the result in D0.  S0 and S1 are handled as integers in the operation. | | |
| Behavior model | D0 = (int)S0 + (int)S1; | |  |

ADDS

**ADD with Saturation**

| Usage | ADDS D0, S0, S1 |
| --- | --- |
| Function | This is an instruction to add 2 signed 32-bit values from source S0 and source S1 with saturation. Mapped on Thread. |
| Behavior model | D0 = SAT(S0 + S1, 32); |
| Operand type | D0: 32-bit signed integer  S0: 32-bit signed integer  S1: 32-bit signed integer |

ADDSS8

**ADD Signed with Saturation 8-bit values**

| Usage | ADDSS8 D0, S0, S1 |
| --- | --- |
| Function | This is an instruction to add 8 times 2 signed 8-bit values from source S0 and source S1 with saturation. |
| Behavior model | D0[n] = SAT(S0[n] + S1[n], 8), n = 0..7; |
| Operand type | D0: 8 times 8-bit signed integer  S0: 8 times 8-bit signed integer  S1: 8 times 8-bit signed integer |

ADDSS16

**ADD Signed with Saturation 16-bit values**

| Usage | ADDSS16 D0, S0, S1 |
| --- | --- |
| Function | This is an instruction to add 4 times 2 signed 16-bit values from source S0 and source S1 with saturation. |
| Behavior model | D0[n] = SAT(S0[n] + S1[n], 16), n = 0..3; |
| Operand type | D0: 4 times 16-bit signed integer  S0: 4 times 16-bit signed integer  S1: 4 times 16-bit signed integer |

ADDSU

**ADD with Saturation signed Uniform value**

| Usage | ADDSU D0, S0, #IMM |
| --- | --- |
| Function | This is an instruction to add 2 signed 32-bit values from source S0 and Uniform value indexed by #IMM with saturation. Mapped on Thread. |
| Behavior model | D0 = SAT(S0 + Uniform[#R6IMM], 32); |
| Operand type | D0: 32-bit signed integer  S0: 32-bit signed integer |

ADDSI

**ADD with Saturation signed Immediate value**

| Usage | ADDSI D0, S0, #IMM |
| --- | --- |
| Function | This is an instruction to add a signed 32-bit values from source S0 and the immediate 8-bit value #IMM with saturation. Mapped on Thread. |
| Behavior model | D0 = SAT(S0 + #IMM, 32); |
| Operand type | D0: 32-bit signed integer  S0: 32-bit signed integer  #IMM: 8-bit signed integer |

ADDDBL

**ADD DouBL**

| Usage | ADDDBL D0, S0, S1 |
| --- | --- |
| Function | This instruction is available only for V3M Ver.2.0, V3H and V3U.  This instruction treats S0 as two 32-bit integers, upper and lower, and S1 as two 15bit integers, upper and lower. And adds those integers respectively. Then stores the result to D0.  Typical use case of this instruction is to increment two pointers in a loop with one instruction.  D0 and S0 are 64-bit pair registers, always the lower even register is specified, which holds the lower 32-bit. S1 is a 32-bit register. |
| Behavior model | D0[63:32] = {S0[63:48], (16-bit unsign\_extended(S0[46:32])+16-bit unsign\_extended(S1[30:16]))}  D0[31:0] = {S0[31:16], (16-bit unsign\_extended(S0[14:0])+16-bit unsign\_extended(S1[14:0]))} |
| Operand type | D0: Concatenated two 32-bit unsigned integers  S0: Concatenated two 32-bit unsigned integers.  S1: Concatenated two 15-bit unsigned integers (16-bit aligned) |

ADDDBLU

**ADD DouBL with Uniform**

| Usage | ADDDBLU D0, S0, #IMM |
| --- | --- |
| Function | This instruction is available only for V3M Ver.2.0, V3H and V3U.  This instruction treats S0 as two 32-bit integers, upper and lower, and the uniform value indexed by #IMM as two 15bit integers, upper and lower. And adds those integers respectively. Then stores the result to D0.  Typical use case of this instruction is to increment two pointers in a loop with one instruction.  D0 and S0 are 64-bit pair registers, always the lower even register is specified, which holds the lower 32-bit. S1 is a 32-bit register. |
| Behavior model | D0[63:32] = {S0[63:48], (16-bit unsign\_extended(S0[46:32]) + 16-bit unsign\_extended(Uniform[#R6IMM][30:16]))};  D0[31:0] = {S0[31:16], (16-bit unsign\_extended(S0[14:0]) + 16-bit unsign\_extended(Uniform[#R6IMM][14:0]))}; |
| Operand type | D0: Concatenated two 32-bit unsigned integers  S0: Concatenated two 32-bit unsigned integers  Uniform: Concatenated two 15-bit unsigned integers (16-bit aligned) |

ADDI

**ADD to Immediate value**

| Usage | ADDI D0, S0, #IMM |  | |
| --- | --- | --- | --- |
| Function | Adds S0 and #IMM, then stores the result in D0.  S0 and #IMM are handled as integers in the operation. | | |
| Behavior model | D0 = (int)S0 + (char)#IMM; | |  |

ADDU

**ADD to Uniform**

| Usage | ADDU D0, S0, #IMM |
| --- | --- |
| Function | Adds S0 and the uniform value indexed by #IMM, then stores the result in D0.  S0 is handled as an integer in the operation. |
| Behavior model | D0 = S0 + (int)Uniform[#R6IMM]; |

ADDUS8

**ADD Unsigned with Saturation 8-bit values**

| Usage | ADDUS8 D0, S0, S1 |
| --- | --- |
| Function | This is an instruction to add 8 times 2 unsigned 8-bit values from source S0 and source S1 with saturation. |
| Behavior model | D0[n] = SAT(S0[n] + S1[n], 8), n = 0..7; |
| Operand type | D0: 8 times 8-bit unsigned integer  S0: 8 times 8-bit unsigned integer  S1: 8 times 8-bit unsigned integer |

ADDUS16

**ADD Unsigned with Saturation 16-bit values**

| Usage | ADDUS16 D0, S0, S1 |
| --- | --- |
| Function | This is an instruction to add 4 times 2 unsigned 16-bit values from source S0 and source S1 with saturation. |
| Behavior model | D0[n] = SAT(S0[n] + S1[n], 16), n = 0..3; |
| Operand type | D0: 4 times 16-bit unsigned integer  S0: 4 times 16-bit unsigned integer  S1: 4 times 16-bit unsigned integer |

AND

**AND**

| Usage | AND D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Performs the logical AND operation between S0 and S1, then stores the result in D0. | | |
| Behavior model | D0 = S0 & S1; | |  |

AND64

**AND 64-bit values**

| Usage | AND64 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to AND 2 64-bit (8 times 8-bit or 4 times 16-bit) values. | | |
| Behavior model | D0[n] = S0[n] AND S1[n], n = 0..63; | |  |
| Operand type | D0: 64-bit integer  S0: 64-bit integer  S1: 64-bit integer | | |

ANDU

**AND with Uniform**

| Usage | ANDU D0, S0, #IMM | |
| --- | --- | --- |
| Function | Performs the logical AND operation between S0 and the uniform value indexed by #IMM, then stores the result in D0. | |
| Behavior model | D0 = S0 & Uniform[#R6IMM]; |

ASM

**Add and Sub, then each Multiplied**

| Usage | ASM D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This instruction is assumed to be used in Haar Wavelet or FFT butterfly. D0, S0, S1 has 64-bit width which is concatenated by two 32-bit signed integers. Calculate summation and subtraction between higher 32-bit and lower 32-bit of S0. Then each summation and subtraction are multiplied by higher 32-bit and lower 32-bit of S1 respectively. Finally there two values are concatenated into 64-bit width D0.  Hardware does not take care any overflow handlings, so software must take care of it.. | | |
| Behavior model | D0[63:32] = (S0[63:32] + S0[31:0]) \* S1[63:32]; // Each 32-bit is “signed”.  D0[31:0] = (S0[63:32] - S0[31:0]) \* S1[31:0]; // Each 32-bit is “signed”. | |
| Operand type | D0: Concatenated two 32-bit signed integers  S0: Concatenated two 32-bit signed integers  S1: Concatenated two 32-bit signed integers | |

ASMU

**Add and Sub, then each Multiplied by Uniform**

| Usage | ASMU D0, S0, #IMM |
| --- | --- |
| Function | This instruction is assumed to be used in Haar Wavelet or FFT butterfly. D0, S0 has 64-bit width which is concatenated by two 32-bit signed integers. Calculate summation and subtraction between higher 32-bit and lower 32-bit of S0. Then each summation and subtraction are multiplied by a constant in uniform value indexed by #IMM. Finally there two values are concatenated into 64-bit width D0.  Hardware does not take care any overflow handlings, so software must take care of it. |
| Behavior model | D0[63:32] = (S0[63:32] + S0[31:0]) \* Uniform[#R6IMM];  D0[31:0] = (S0[63:32] - S0[31:0]) \* Uniform[#R6IMM]; |
| Operand type | D0: Concatenated two 32-bit signed integers  S0: Concatenated two 32-bit signed integers  Uniform: 32-bit signed integer |

BRA

**BRAnch**

| Usage | BRA #LABEL |  | |
| --- | --- | --- | --- |
| Function | Branches to #LABEL. | | |
| Behavior model | PC=#LABEL; | |  |
| Operand type | #LABEL: 16-bit unsigned integer | |  |

BRAR

**BRAnch (Register)**

| Usage | BRAR S0 |  | |
| --- | --- | --- | --- |
| Function | Branches to the PC of the register value specified by S0.  The 16 lower-order bits of S0 are handled as an unsigned integer. | | |
| Behavior model | PC= (ushort)(S0&0xffff); | |  |
| Operand type | S0: 16-bit unsigned integer | |  |

BRCID

**BRanch based on Core ID**

| Usage | BRCID #IMM |  | |
| --- | --- | --- | --- |
| Function | Jump to new PC which is defined as the sum of by 1 incremented old PC address and the product of the own core ID times the specified 8-bit signed immediate value. The address is a word address. | | |
| Behavior model | PC=PC+(core ID \* #IMM)+1; | |  |

BRTID

**BRanch based on Thread ID**

| Usage | BRTID #IMM |  | |
| --- | --- | --- | --- |
| Function | Jump to new PC which is defined as the sum of by 1 incremented old PC address and the product of the own thread ID times the specified 8-bit signed immediate value. The address is a word address. | | |
| Behavior model | PC=PC+(thread ID \* #IMM)+1; | |  |

BTST

**Bit TeST**

| Usage | BTST D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This instruction copies the S1th bit from S0 to D0, in case D0 is R3, SR is set with this bit. Mapped on each Thread. | | |
| Behavior model | D0 = (S0 >> S1) & 1 | |  |
| Operand type | D0: 1bit integer  S0: 32-bit integer  S1: 5bit unsigned integer | |  |

**Bit TeST with S0th bit from UNI[#IMM]**

BTSTU

| Usage | BTSTU D0, S0, #IMM |  | |
| --- | --- | --- | --- |
| Function | This instruction copies the S0th bit from UNI[#IMM] to D0, in case D0 is R3, SR is set with this bit. Mapped on each Thread. | | |
| Behavior model | D0 = (UNI[#IMM] >> S0) & 1 | |  |
| Operand type | D0: 1bit integer  S0: 5bit unsigned integer  #IMM: 6bit unsigned integer | |  |

BTSTI

**Bit TeST with Immediate bit selection**

| Usage | BTSTI D0, S0, #IMM |  | |
| --- | --- | --- | --- |
| Function | This instruction copies the #IMMth bit from S0 to D0, in case D0 is R3, SR is set with this bit. Mapped on each Thread. | | |
| Behavior model | D0 = (S0 >> #IMM) & 1 | |  |
| Operand type | D0: 1bit integer  S0: 32-bit integer  #IMM: 5bit unsigned integer | |  |

CALL

**sub routine CALL**

| Usage | CALL D0, #LABEL |  |
| --- | --- | --- |
| Function | PC+1 is saved to D0, and branch to #LABEL. | |
| Behavior model | D0 = (PC+1); PC=#LABEL; | |

CALLA

**sub routine CALL Alternative**

| Usage | CALLA D0, #LABEL |  | |
| --- | --- | --- | --- |
| Function | Saves PC+1 to the temporary-storage PC register specified by D0 and branches to #LABEL. Although this temporary-storage PC register is not described in the list of registers, but it is implicitly implemented.  The register to be specified by D0 is not the general-purpose register number, but the CALLA/RETA dedicated temporary-storage PC register number. There are eight registers for this purpose and 0 to 7 can be specified.  The 3 lower-order bits of D0 are handled as an unsigned integer.  For example, to save the return address to the 3rd PC register, let D0 to have the value 3 before CALLA instruction. | | |
| Behavior model | PCR[(D0&0x7)]=PC+1; PC=#LABEL; | |  |
| Operand type | D0: 3-bit unsigned integer  #LABEL: 16-bit unsigned integer | |  |

CALLR

**sub routine CALL(Register)**

| Usage | CALLR D0, S0 |  |
| --- | --- | --- |
| Function | PC+1 is saved to D0, and branch to the PC specified by S0. | |
| Behavior model | D0 = (PC+1); PC=S0; | |

CMP.EQ

**EQual**

| Usage | CMP.EQ D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 is equal to S1, sets the SR.F bit to 1, otherwise sets it to 0.  D0 != R3 (zero register): If S0 is equal to S1, D0 is set to 1, else it is set to 0. In this case, SR.F is not updated.  Both operands S0 and S1 are handled as 32-bit signed integers in the comparison. | | |
| Behavior model | SR.F = (S0 == S1 ); (D0 == R3)  D0 = (S0 == S1 ); (D0 != R3) | |  |
| Operand type | S0: 32-bit signed integer  S1: 32-bit signed integer | |  |

CMP.EQ8

**CoMPare EQual 8-bit values**

| Usage | CMP.EQ8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to compare (equal) 8 8-bit values with 8 8-bit value. The result of the nth compare is stored in the nth byte of D0. | | |
| Behavior model | D0[n] = S0[n] == S1[n], n = 0..7; | |  |
| Operand type | D0: 8 times 8-bit unsigned integer, only lowest bit in each byte used  S0: 8 times 8-bit integer  S1: 8 times 8-bit integer | |  |

CMP.EQ16

**CoMPare EQual 16-bit values**

| Usage | CMP.EQ16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to compare (equal) 4 16-bit values with 4 16-bit value. The result of the nth compare is stored in the nth 16-bit word of D0. | | |
| Behavior model | D0[n] = S0[n] == S1[n], n = 0..3; | |  |
| Operand type | D0: 4 times 16-bit unsigned integer, only lowest bit in each 16-bit word used  S0: 4 times 16-bit integer  S1: 4 times 16-bit integer | |  |

CMP.NEQ

**Not EQual**

| Usage | CMP.NEQ D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 is not equal to S1, sets the SR.F bit to 1, otherwise sets it to 0.  D0 != R3 (zero register): If S0 is not equal to S1, D0 is set to 1, else it is set to 0. In this case, SR.F is not updated.  Both operands S0 and S1 are handled as 32-bit signed integers in the comparison. | | |
| Behavior model | SR.F = (S0 != S1 ); (D0 == R3)  D0 = (S0 != S1 ); (D0 != R3) | |  |
| Operand type | S0: 32-bit signed integer  S1: 32-bit signed integer | |  |

CMP.NEQ8

**CoMPare Not EQual 8-bit values**

| Usage | CMP.NEQ8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to compare (not equal) 8 8-bit values with 8 8-bit value. The result of the nth compare is stored in the nth byte of D0. | | |
| Behavior model | D0[n] = S0[n] != S1[n] , n = 0..7; | |  |
| Operand type | D0: 8 times 8-bit unsigned integer, only lowest bit in each byte used  S0: 8 times 8-bit integer  S1: 8 times 8-bit integer | |  |

CMP.NEQ16

**CoMPare Not EQual 16-bit values**

| Usage | CMP.NEQ16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to compare (not equal) 4 16-bit values with 4 16-bit value. The result of the nth compare is stored in the nth 16-bit word of D0. | | |
| Behavior model | D0[n] = S0[n] != S1[n], n = 0..3; | |  |
| Operand type | D0: 4 times 16-bit unsigned integer, only lowest bit in each 16-bit word used  S0: 4 times 16-bit integer  S1: 4 times 16-bit integer | |  |

CMP.SGE

**Greater than or Equal**

| Usage | CMP.SGE D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 is greater than or equal to S1, sets the SR.F bit to 1, otherwise sets it to 0.  D0 != R3 (zero register): If S0 is greater than or equal to S1, D0 is set to 1, else it is set to 0. In this case, SR.F is not updated.  Both operands S0 and S1 are handled as 32-bit signed integers in the comparison. | | |
| Behavior model | SR.F = (S0 >= S1 ); (D0 == R3)  D0 = (S0 >= S1 ); (D0 != R3) | |  |
| Operand type | S0: 32-bit signed integer  S1: 32-bit signed integer | |  |

CMP.SGT

**Greater Than**

| Usage | CMP.SGT D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 is greater than S1, sets the SR.F bit to 1, otherwise sets it to 0.  D0 != R3 (zero register): If S0 is greater than S1, D0 is set to 1, else it is set to 0. In this case, SR.F is not updated.  Both operands S0 and S1 are handled as 32-bit signed integers in the comparison. | | |
| Behavior model | SR.F = (S0 > S1 ); (D0 == R3)  D0 = (S0 > S1 ); (D0 != R3) | |  |
| Operand type | S0: 32-bit signed integer  S1: 32-bit signed integer | |  |

CMP.SLE

**Less than or Equal**

| Usage | CMP.SLE D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 is smaller than or equal to S1, sets the SR.F bit to 1, otherwise sets it to 0.  D0 != R3 (zero register): If S0 is smaller than or equal to S1, D0 is set to 1, else it is set to 0. In this case, SR.F is not updated.  Both operands are handled as 32-bit signed integers in the comparison. | | |
| Behavior model | SR.F = (S0 <= S1 ); (D0 == R3)  D0 = (S0 <= S1 ); (D0 != R3) | |  |
| Operand type | S0: 32-bit signed integer  S1: 32-bit signed integer | |  |

CMP.SLE8

**CoMPare Signed Less than or Equal 8-bit values**

| Usage | CMP.SLE8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to compare (lower or equal than) 8 signed 8-bit values with 8 signed 8-bit value. The result of the nth compare is stored in the nth byte of D0. Implemented in SAM4. | | |
| Behavior model | D0[n] = S0[n] <= S1[n] , n = 0..7; | |  |
| Operand type | D0: 8 times 8-bit unsigned integer, only lowest bit in each byte used  S0: 8 times 8-bit signed integer  S1: 8 times 8-bit signed integer | |  |

CMP.SLE16

**CoMPare Signed Less than or Equal 16-bit values**

| Usage | CMP.SLE16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to compare (lower or equal than) 4 signed 16-bit values with 4 signed 16-bit value. The result of the nth compare is stored in the nth 16-bit word of D0. | | |
| Behavior model | D0[n] = S0[n] <= S1[n] , n = 0..3; | |  |
| Operand type | D0: 4 times 16-bit unsigned integer, only lowest bit in each 16-bit word used  S0: 4 times 16-bit signed integer  S1: 4 times 16-bit signed integer | |  |

CMP.SLT

**Less than**

| Usage | CMP.SLT D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 is smaller than S1, sets the SR.F bit to 1, otherwise sets it to 0.  D0 != R3 (zero register): If S0 is smaller than S1, D0 is set to 1, else it is set to 0. In this case, SR.F is not updated.  Both operands are handled as 32-bit signed integers in the comparison. | | |
| Behavior model | SR.F = (S0 < S1 ); (D0 == R3)  D0 = (S0 < S1 ); (D0 != R3) | |  |
| Operand type | S0: 32-bit signed integer  S1: 32-bit signed integer | |  |

CMP.SLT8

**CoMPare Signed Less than 8-bit values**

| Usage | CMP.SLT8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to compare (lower than) 8 signed 8-bit values with 8 signed 8-bit value. The result of the nth compare is stored in the nth byte of D0. | | |
| Behavior model | D0[n] = S0[n] < S1[n] , n = 0..7; | |  |
| Operand type | D0: 8 times 8-bit unsigned integer, only lowest bit in each byte used  S0: 8 times 8-bit signed integer  S1: 8 times 8-bit signed integer | |  |

CMP.SLT16

**CoMPare Signed Less than 16-bit values**

| Usage | CMP.SLT16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to compare (lower than) 4 signed 16-bit values with 4 signed 16-bit value. The result of the nth compare is stored in the nth 16-bit word of D0. | | |
| Behavior model | D0[n] = S0[n] < S1[n] , n = 0..3; | |  |
| Operand type | D0: 4 times 16-bit unsigned integer, only lowest bit in each 16-bit word used  S0: 4 times 16-bit signed integer  S1: 4 times 16-bit signed integer | |  |

CMP.ULE8

**CoMPare Unsigned Less than or Equal 8-bit values**

| Usage | CMP.ULE8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to compare (lower or equal than) 8 unsigned 8-bit values with 8 unsigned 8-bit value. The result of the nth compare is stored in the nth byte of D0. | | |
| Behavior model | D0[n] = S0[n] <= S1[n] , n = 0..7; | |  |
| Operand type | D0: 8 times 8-bit unsigned integer, only lowest bit in each byte used  S0: 8 times 8-bit unsigned integer  S1: 8 times 8-bit unsigned integer | |  |

CMP.ULE16

**CoMPare Unsigned Less than or Equal 16-bit values**

| Usage | CMP.ULE16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to compare (lower or equal than) 4 unsigned 16-bit values with 4 unsigned 16-bit value. The result of the nth compare is stored in the nth 16-bit word of D0. | | |
| Behavior model | D0[n] = S0[n] <= S1[n] , n = 0..3; | |  |
| Operand type | D0: 4 times 16-bit unsigned integer, only lowest bit in each 16-bit word used  S0: 4 times 16-bit unsigned integer  S1: 4 times 16-bit unsigned integer | |  |

CMP.ULT8

**CoMPare Unsigned Less than 8-bit values**

| Usage | CMP.ULT8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to compare (lower than) 8 unsigned 8-bit values with 8 unsigned 8-bit value. The result of the nth compare is stored in the nth byte of D0. | | |
| Behavior model | D0[n] = S0[n] < S1[n] , n = 0..7; | |  |
| Operand type | D0: 8 times 8-bit unsigned integer, only lowest bit in each byte used  S0: 8 times 8-bit unsigned integer  S1: 8 times 8-bit unsigned integer | |  |

CMP.ULT16

**CoMPare Unsigned Less than 16-bit values**

| Usage | CMP.ULT16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to compare (lower than) 4 unsigned 16-bit values with 4 unsigned 16-bit value. The result of the nth compare is stored in the nth 16-bit word of D0. | | |
| Behavior model | D0[n] = S0[n] < S1[n] , n = 0..3; | |  |
| Operand type | D0: 4 times 16-bit unsigned integer, only lowest bit in each 16-bit word used  S0: 4 times 16-bit unsigned integer  S1: 4 times 16-bit unsigned integer | |  |

CMPU.EQ

**EQual to Uniform**

| Usage | CMPU.EQ D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 is equal to the uniform value indexed by #IMM, sets the SR.F bit to 1, otherwise sets it to 0.  D0 != R3 (zero register): If S0 is equal to the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. In this case, SR.F is not updated.  Both operands are handled as signed integers in the comparison. |
| Behavior model | SR.F = (S0 == Uniform[#R6IMM] ); (D0 == R3)  D0 = (S0 == Uniform[#R6IMM] ); (D0 != R3) |
| Operand type | S0: 32-bit signed integer  R6: 8-bit unsigned integer  Uniform: 32-bit signed integer |

CMPU.NEQ

**Not EQual to Uniform**

| Usage | CMPU.NEQ D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 is not equal to the uniform value indexed by #IMM, sets the SR.F bit to 1, otherwise sets it to 0.  D0 != R3 (zero register): If S0 is not equal to the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. In this case, SR.F is not updated.  Both operands are handled as signed integers in the comparison. |
| Behavior model | SR.F = (S0 != Uniform [#R6IMM] ); (D0 == R3)  D0 = (S0 != Uniform [#R6IMM]); (D0 != R3) |
| Operand type | S0: 32-bit signed integer  R6: 8-bit unsigned integer  Uniform: 32-bit signed integer |

CMPU.SGE

**Greater than or Equal to Uniform**

| Usage | CMPU.SGE D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 is greater than or equal to the uniform value indexed by #IMM, sets the SR.F bit to 1, otherwise sets it to 0.  D0 != R3 (zero register): If S0 is greater than or equal to the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. In this case, SR.F is not updated.  Both operands are handled as signed integers in the comparison. |
| Behavior model | SR.F = (S0 >= Uniform[#R6IMM] ); (D0 == R3)  D0 = (S0 >= Uniform[#R6IMM] ); (D0 != R3) |
| Operand type | S0: 32-bit signed integer  R6: 8-bit unsigned integer  Uniform: 32-bit signed integer |

CMPU.SGT

**Greater Than to Uniform**

| Usage | CMPU.SGT D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 is greater than the uniform value indexed by #IMM, sets the SR.F bit to 1, otherwise sets it to 0.  D0 != R3 (zero register): If S0 is greater than the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. In this case, SR.F is not updated.  Both operands are handled as signed integers in the comparison. |
| Behavior model | SR.F = (S0 > Uniform[#R6IMM] ); (D0 == R3)  D0 = (S0 > Uniform[#R6IMM] ); (D0 != R3) |
| Operand type | S0: 32-bit signed integer  R6: 8-bit unsigned integer  Uniform: 32-bit signed integer |

CMPU.SLE

**Less than or Equal to Uniform**

| Usage | CMPU.SLE D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 is smaller than or equal to the uniform value indexed by #IMM, sets the SR.F bit to 1, otherwise sets it to 0.  D0 != R3 (zero register): If S0 is smaller than or equal to the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. In this case, SR.F is not updated.  Both operands are handled as signed integers in the comparison. |
| Behavior model | SR.F = (S0 <= Uniform[#R6IMM] ); (D0 == R3)  D0 = (S0 <= Uniform[#R6IMM] ); (D0 != R3) |
| Operand type | S0: 32-bit signed integer  R6: 8-bit unsigned integer  Uniform: 32-bit signed integer |

CMPU.SLT

**Less Than Uniform**

| Usage | CMPU.SLT D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 is smaller than the uniform value indexed by #IMM, sets the SR.F bit to 1, otherwise sets it to 0.  D0 != R3 (zero register): If S0 is smaller than the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. In this case, SR.F is not updated.  Both operands are handled as signed integers in the comparison. |
| Behavior model | SR.F = (S0 < Uniform[#R6IMM]); (D0 == R3)  D0 = (S0 < Uniform[#R6IMM); (D0 != R3) |
| Operand type | S0: 32-bit signed integer  R6: 8-bit unsigned integer  Uniform: 32-bit signed integer |

CMV8

**Conditional MoVe of 8-bit values**

| Usage | CMV8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to copy dependent on 2 condition bits stored in S0[n] and S0[32+n] the value from S1[7:0] to D0[(n+1)\*8-1:n\*8]. The values of other bits are retained. | | |
| Behavior model | if (S0[n] && (S0[32+n]) then D0[(n+1)\*8-1:n\*8] = S1[7:0], n=0..7 | |  |
| Operand type | D0: 8 times 8-bit value (=64-bit)  S0: 64-bit value (only bits 0..7 and 32..39 used)  S1: 8-bit value in lower 8-bit | | |

CMV16

**Conditional MoVe of 16-bit values**

| Usage | CMV16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to copy dependent on 2 condition bits stored in S0[n] and S0[32+n] the value from S1[15:0] to D0[(n+1)\*16-1:n\*16]. The values of other bits are retained. | | |
| Behavior model | if (S0[n] && (S0[32+n]) then D0[(n+1)\*16-1:n\*16] = S1[15:0], n=0..3 | |  |
| Operand type | D0: 4 times 16-bit value (=64-bit)  S0: 64-bit value (only bits 0..3 and 32..35 used)  S1: 16-bit value in lower 16-bit | | |

CNT0

**Count 0**

| Usage | CNT0 D0, S0 |  | |
| --- | --- | --- | --- |
| Function | Treats S0 as a 32-bit array, counts the number of zero-valued bits in this array, and stores the result in D0. | | |
| Behavior model | D0 = {number of zero-valued bits in S0} | |  |

CNT1

**Count 1**

| Usage | CNT1 D0, S0 |  | |
| --- | --- | --- | --- |
| Function | Treats S0 as a 32-bit array, counts the number of one-valued bits in this array, and stores the result in D0. | | |
| Behavior model | D0 = {number of one-valued bits in S0} | |  |

CSUSS

**Signed census**

| Usage | CSUSS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = (S0[23:16] < D0[15:8]) << 31 |  (D0[23:16] < D0[15:8]) << 30 |  (S1[23:16] < D0[15:8]) << 29 |  (S0[15:8] < D0[15:8]) << 28 |  (S1[15:8] < D0[15:8]) << 27 |  (S0[7:0] < D0[15:8]) << 26 |  (D0[7:0] < D0[15:8]) << 25 |  (S1[7:0] < D0[15:8]) << 24 |  D0[23:0];  S0, S1 and D0 are 3 24bit vectors with each holding 3 8-bit signed elements. The instruction is calculating the census operation by comparing the center byte stored in D0[15:8] with each of the other 8 bytes. Each comparison delivers a 0 or 1 depending on the result of the comparison. These 8 comparison results are stored into D0[31:24] before D0 is stored back to the register file. | | |
| Behavior model | D0 = (S0[23:16] < D0[15:8]) << 31 |  (D0[23:16] < D0[15:8]) << 30 |  (S1[23:16] < D0[15:8]) << 29 |  (S0[15:8] < D0[15:8]) << 28 |  (S1[15:8] < D0[15:8]) << 27 |  (S0[7:0] < D0[15:8]) << 26 |  (D0[7:0] < D0[15:8]) << 25 |  (S1[7:0] < D0[15:8]) << 24 |  D0[23:0]; | |  |

CSUSU

**Unsigned census**

| Usage | CSUSU D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = (S0[23:16] < D0[15:8]) << 31 |  (D0[23:16] < D0[15:8]) << 30 |  (S1[23:16] < D0[15:8]) << 29 |  (S0[15:8] < D0[15:8]) << 28 |  (S1[15:8] < D0[15:8]) << 27 |  (S0[7:0] < D0[15:8]) << 26 |  (D0[7:0] < D0[15:8]) << 25 |  (S1[7:0] < D0[15:8]) << 24 |  D0[23:0];  S0, S1 and D0 are 3 24bit vectors with each holding 3 8-bit unsigned elements. The instruction is calculating the census operation by comparing the center byte stored in D0[15:8] with each of the other 8 bytes. Each comparison delivers a 0 or 1 depending on the result of the comparison. These 8 comparison results are stored into D0[31:24] before D0 is stored back to the register file. | | |
| Behavior model | D0 = (S0[23:16] < D0[15:8]) << 31 |  (D0[23:16] < D0[15:8]) << 30 |  (S1[23:16] < D0[15:8]) << 29 |  (S0[15:8] < D0[15:8]) << 28 |  (S1[15:8] < D0[15:8]) << 27 |  (S0[7:0] < D0[15:8]) << 26 |  (D0[7:0] < D0[15:8]) << 25 |  (S1[7:0] < D0[15:8]) << 24 |  D0[23:0]; | |  |

DMAWAITS

**Wait for the specified DMA**

| Usage | DMAWAITS #IMM |  | | |
| --- | --- | --- | --- | --- |
| Function | This is a command to wait for the finish of DMA transfers. With #IMM you are specifying the thread of the core. For thread #n, each k-th bit of #IMM corresponds to TGDMAC#((n+k)%8), which the master thread should wait for. Refer to section 8.3. | | | |
| Behavior model | #IMM[k] == 1? DMAWAITS = 1 ; check for each specified bit (n+k)%8(== slave thread) and wait till DMAWAITS is zero. | | |  |
| Operand type | #IMM: 8-bit unsigned integer | |

EOR

**Exclusive OR**

| Usage | EOR D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Treats S0 and S1 as 32-bit arrays, takes the exclusive OR (XOR) of S0 and S1, and stores the  result in D0. | | |
| Behavior model | D0 =S0 ^ S1; | |  |

EOR64

**Exclusive OR 64-bit values**

| Usage | EOR64 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to XOR 2 64-bit (8 times 8-bit or 4 times 16-bit) values. | | |
| Behavior model | D0[n] = S0[n] XOR S1[n], n = 0..63; | |  |
| Operand type | D0: 64-bit integer  S0: 64-bit integer  S1: 64-bit integer | |  |

EORU

**Exclusive OR with Uniform**

| Usage | EORU D0, S0, #IMM |
| --- | --- |
| Function | Treats S0 and the uniform value indexed by #IMM as 32-bit arrays, takes the exclusive OR (XOR) of S0 and the uniform value, and stores the result in D0. | |
| Behavior model | D0 =S0 ^ Uniform[#R6IMM]; | |

F2I

**Float to Integer**

| Usage | F2I D0, S0 |  | |
| --- | --- | --- | --- |
| Function | Assumes the S0 value is an IEEE754 single-precision floating-point number, converts it into a 32-bit signed integer, then stores the result in D0.  The rounding method can be changed by the control register. | | |
| Behavior model | D0 =int((float)S0 ); | |  |

FABS

**Float ABSolute**

| Usage | FABS D0, S0 |  | |
| --- | --- | --- | --- |
| Function | Stores the absolute value of S0 in D0.  S0 is handled as a floating-point number in the operation. | | |
| Behavior model | D0 = |(float)S0|; | |  |

FADD

**Float ADD**

| Usage | FADD D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Adds S0 and S1, then stores the result in D0.  S0 and S1 are handled as floating-point numbers in the operation. | | |
| Behavior model | D0 = (float)S0 + (float)S1; | |  |

FADDU

**Float ADD to Uniform**

| Usage | FADDU D0, S0, #IMM |
| --- | --- |
| Function | Adds S0 and the uniform value indexed by #IMM, then stores the result in D0.  S0 and the uniform value are handled as floating-point numbers in the operation. |
| Behavior model | D0 = (float)S0 + (float)Uniform[#R6IMM]; |

FCMP.O

**Ordered Floating point**

| Usage | FCMP.O D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 and S1 are not NaN, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 and S1 are not NaN, D0 is set to 1, else it is set to 0. | | |
| Behavior model | SR.F = S0 != NaN and S1 != NaN; (D0 == R3)  D0 = S0 != NaN and S1 != NaN; (D0 != R3) | |  |

FCMP.OEQ

**Ordered Floating point equal**

| Usage | FCMP.OEQ D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 and S1 are not NaN and S0 is equal to S1, status register flag (SR.F) is set to 1, else it is set to 0. This instruction is equal to the earlier instruction FCMP.EQ.  D0 != R3 (zero register): If S0 and S1 are not NaN and S0 is equal to S1, D0 is set to 1, else it is set to 0. | | |
| Behavior model | SR.F = (S0 == S1 ) and S0 != NaN and S1 != NaN; (D0 == R3)  D0 = (S0 == S1 ) and S0 != NaN and S1 != NaN; (D0 != R3) | |  |

FCMP.OGE

**Ordered Floating point greater than or equal**

| Usage | FCMP.OGE D0, S0, S1 |  |
| --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 and S1 are not NaN and S0 is greater than or equal S1, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 and S1 are not NaN and S0 is greater than or equal S1, D0 is set to 1, else it is set to 0. | |
| Behavior model | SR.F = (S0 >= S1 ) and S0 != NaN and S1 != NaN; (D0 == R3)  D0 = (S0 >= S1 ) and S0 != NaN and S1 != NaN; (D0 != R3) |  |

FCMP.OGT

**Ordered Floating point greater than**

| Usage | FCMP.OGT D0, S0, S1 |  |
| --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 and S1 are not NaN and S0 is greater than S1, status register flag (SR.F) is set to 1, else it is set to 0. This instruction is equal to the earlier instruction FCMP.GT.  D0 != R3 (zero register): If S0 and S1 are not NaN and S0 is greater than S1, D0 is set to 1, else it is set to 0. | |
| Behavior model | SR.F = (S0 > S1 ) and S0 != NaN and S1 != NaN; (D0 == R3)  D0 = (S0 > S1 ) and S0 != NaN and S1 != NaN; (D0 != R3) |  |

FCMP.OLE

**Ordered Floating point less than or equal**

| Usage | FCMP.OLE D0, S0, S1 |  |
| --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 and S1 are not NaN and S0 is less than or equal S1, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 and S1 are not NaN and S0 is less than or equal S1, D0 is set to 1, else it is set to 0. | |
| Behavior model | SR.F = (S0 <= S1 ) and S0 != NaN and S1 != NaN; (D0 == R3)  D0 = (S0 <= S1 ) and S0 != NaN and S1 != NaN; (D0 != R3) |  |

FCMP.OLT

**Ordered Floating point less than**

| Usage | FCMP.OLT D0, S0, S1 |  |
| --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 and S1 are not NaN and S0 is less than S1, status register flag (SR.F) is set to 1, else it is set to 0. This instruction is equal to the earlier instruction FCMP.LT.  D0 != R3 (zero register): If S0 and S1 are not NaN and S0 is less than S1, D0 is set to 1, else it is set to 0. | |
| Behavior model | SR.F = (S0 < S1 ) and S0 != NaN and S1 != NaN; (D0 == R3)  D0 = (S0 < S1 ) and S0 != NaN and S1 != NaN; (D0 != R3) |  |

FCMP.ONE

**Ordered Floating point not equal**

| Usage | FCMP.ONE D0, S0, S1 |  |
| --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 and S1 are not NaN and S0 is not equal S1, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 and S1 are not NaN and S0 is not equal S1, D0 is set to 1, else it is set to 0. | |
| Behavior model | SR.F = (S0 != S1 ) and S0 != NaN and S1 != NaN; (D0 == R3)  D0 = (S0 != S1 ) and S0 != NaN and S1 != NaN; (D0 != R3) |  |

FCMP.UEQ

**Unordered Floating point equal**

| Usage | FCMP.UEQ D0, S0, S1 |  |
| --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 is NaN or S1 is NaN or S0 is equal to S1, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 is NaN or S1 is NaN or S0 is equal to S1, D0 is set to 1, else it is set to 0. | |
| Behavior model | SR.F = (S0 == S1 ) or S0 == NaN or S1 == NaN; (D0 == R3)  D0 = (S0 == S1 ) or S0 == NaN or S1 == NaN; (D0 != R3) |  |

FCMP.UGE

**Unordered Floating point greater than or equal**

| Usage | FCMP.UGE D0, S0, S1 |  |
| --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 is NaN or S1 is NaN or S0 is greater than or equal S1, status register flag (SR.F) is set to 1, else it is set to 0. This instruction is equal to the earlier instruction FCMP.GE.  D0 != R3 (zero register): If S0 is NaN or S1 is NaN or S0 is greater than or equal S1, D0 is set to 1, else it is set to 0. | |
| Behavior model | SR.F = (S0 >= S1 ) or S0 == NaN or S1 == NaN; (D0 == R3)  D0 = (S0 >= S1 ) or S0 == NaN or S1 == NaN; (D0 != R3) |  |

FCMP.UGT

**Unordered Floating point greater than**

| Usage | FCMP.UGT D0, S0, S1 |  |
| --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 is NaN or S1 is NaN or S0 is greater than S1, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 is NaN or S1 is NaN or S0 is greater than S1, D0 is set to 1, else it is set to 0. | |
| Behavior model | SR.F = (S0 > S1 ) or S0 == NaN or S1 == NaN; (D0 == R3)  D0 = (S0 > S1 ) or S0 == NaN or S1 == NaN; (D0 != R3) |  |

FCMP.ULE

**Unordered Floating point less than or equal**

| Usage | FCMP.ULE D0, S0, S1 |  |
| --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 is NaN or S1 is NaN or S0 is less than or equal S1, status register flag (SR.F) is set to 1, else it is set to 0.. This instruction is equal to the earlier instruction FCMP.LE.  D0 != R3 (zero register): If S0 is NaN or S1 is NaN or S0 is less than or equal S1, D0 is set to 1, else it is set to 0. | |
| Behavior model | SR.F = (S0 <= S1 ) or S0 == NaN or S1 == NaN; (D0 == R3)  D0 = (S0 <= S1 ) or S0 == NaN or S1 == NaN; (D0 != R3) |  |

FCMP.ULT

**Unordered Floating point less than**

| Usage | FCMP.ULT D0, S0, S1 |  |
| --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 is NaN or S1 is NaN or S0 is less than S1, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 is NaN or S1 is NaN or S0 is less than S1, D0 is set to 1, else it is set to 0. | |
| Behavior model | SR.F = (S0 < S1 ) or S0 == NaN or S1 == NaN; (D0 == R3)  D0 = (S0 < S1 ) or S0 == NaN or S1 == NaN; (D0 != R3) |  |

FCMP.UNE

**Unordered Floating not equal**

| Usage | FCMP.UNE D0, S0, S1 |  |
| --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 is NaN or S1 is NaN or S0 is not equal S1, status register flag (SR.F) is set to 1, else it is set to 0.This instruction is equal to the earlier instruction FCMP.NEQ.  D0 != R3 (zero register): If S0 is NaN or S1 is NaN or S0 is not equal S1, D0 is set to 1, else it is set to 0. | |
| Behavior model | SR.F = (S0 != S1 ) or S0 == NaN or S1 == NaN; (D0 == R3)  D0 = (S0 != S1 ) or S0 == NaN or S1 == NaN; (D0 != R3) |  |

FCMP.UO

**Unordered Floating point**

| Usage | FCMP.UO D0, S0, S1 |  |
| --- | --- | --- |
| Function | D0 == R3 (zero register): If S0 is NaN or S1 is NaN, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 is NaN or S1 is NaN, D0 is set to 1, else it is set to 0. | |
| Behavior model | SR.F = S0 == NaN or S1 == NaN; (D0 == R3)  D0 = S0 == NaN or S1 == NaN; (D0 != R3) |  |

FCMPU.O

**Ordered Floating point with Uniform**

| Usage | FCMPU.O D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 and the uniform value indexed by #IMM are not NaN, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 and the uniform value indexed by #IMM, are not NaN, D0 is set to 1, else it is set to 0. |
| Behavior model | SR.F = S0 != NaN and Uniform[#R6IMM] != NaN; (D0 == R3)  D0 = S0 != NaN and Uniform[#R6IMM] != NaN; (D0 != R3) |

FCMPU.OEQ

**Ordered Floating point equal to Uniform**

| Usage | FCMPU.OEQ D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 and the uniform value indexed by #IMM are not NaN and S0 is equal to the uniform value indexed by #IMM, status register flag (SR.F) is set to 1, else it is set to 0. This instruction is equal to the earlier instruction FCMPU.EQ.  D0 != R3 (zero register): If S0 and Uniform[R6+#IMM] are not NaN and S0 is equal to the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. |
| Behavior model | SR.F = (S0 == Uniform[#R6IMM]) and S0 != NaN and Uniform[#R6IMM] != NaN; (D0 == R3)  D0 = (S0 == Uniform[#R6IMM]) and S0 != NaN and Uniform[#R6IMM] != NaN; (D0 != R3) |

FCMPU.OGE

**Ordered Floating point greater than or equal to Uniform**

| Usage | FCMPU.OGE D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 and the uniform value indexed by #IMM are not NaN and S0 is greater than or equal to the uniform value indexed by #IMM, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 and the uniform value indexed by #IMM are not NaN and S0 is greater than or equal to the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. |
| Behavior model | SR.F = (S0 >= Uniform[#R6IMM]) and S0 != NaN and Uniform[#R6IMM] != NaN; (D0 == R3)  D0 = (S0 >= Uniform[#R6IMM]) and S0 != NaN and Uniform[#R6IMM] != NaN; (D0 != R3) |

FCMPU.OGT

**Ordered Floating point greater than Uniform**

| Usage | FCMPU.OGT D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 and the uniform value indexed by #IMM are not NaN and S0 is greater than the uniform value indexed by #IMM, status register flag (SR.F) is set to 1, else it is set to 0. This instruction is equal to the earlier instruction FCMPU.GT.  D0 != R3 (zero register): If S0 and the uniform value indexed by #IMM are not NaN and S0 is greater than the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. |
| Behavior model | SR.F = (S0 > Uniform[#R6IMM]) and S0 != NaN and Uniform[#R6IMM] != NaN; (D0 == R3)  D0 = (S0 > Uniform[#R6IMM]) and S0 != NaN and Uniform[#R6IMM] != NaN; (D0 != R3) |

FCMPU.OLE

**Ordered Floating point less than or equal to Uniform**

| Usage | FCMPU.OLE D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 and the uniform value indexed by #IMM are not NaN and S0 is less than or equal to the uniform value indexed by #IMM, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 and the uniform value indexed by #IMM are not NaN and S0 is less than or equal to the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. |
| Behavior model | SR.F = (S0 <= Uniform[#R6IMM]) and S0 != NaN and Uniform[#R6IMM] != NaN; (D0 == R3)  D0 = (S0 <= Uniform[#R6IMM]) and S0 != NaN and Uniform[#R6IMM] != NaN; (D0 != R3) |

FCMPU.OLT

**Ordered Floating point less than Uniform**

| Usage | FCMPU.OLT D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 and the uniform value indexed by #IMM are not NaN and S0 is less than the uniform value indexed by #IMM, status register flag (SR.F) is set to 1, else it is set to 0. This instruction is equal to the earlier instruction FCMPU.LT.  D0 != R3 (zero register): If S0 and the uniform value indexed by #IMM are not NaN and S0 is less than the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. |
| Behavior model | SR.F = (S0 < Uniform[#R6IMM]) and S0 != NaN and Uniform[#R6IMM] != NaN; (D0 == R3)  D0 = (S0 < Uniform[#R6IMM]) and S0 != NaN and Uniform[#R6IMM] != NaN; (D0 != R3) |

FCMPU.ONE

**Ordered Floating point not equal to Uniform**

| Usage | FCMPU.ONE D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 and the uniform value indexed by #IMM are not NaN and S0 is not equal the uniform value indexed by #IMM, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 and the uniform value indexed by #IMM are not NaN and S0 is not equal the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. |
| Behavior model | SR.F = (S0 != Uniform[#R6IMM]) and S0 != NaN and Uniform[#R6IMM] != NaN; (D0 == R3)  D0 = (S0 != Uniform[#R6IMM]) and S0 != NaN and Uniform[#R6IMM] != NaN; (D0 != R3) |

FCMPU.UEQ

**Unordered Floating point equal to Uniform**

| Usage | FCMPU.UEQ D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 is NaN or the uniform value indexed by #IMM is NaN or S0 is equal to the uniform value indexed by #IMM, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 is NaN or the uniform value indexed by #IMM is NaN or S0 is equal to the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. |
| Behavior model | SR.F = (S0 == Uniform[#R6IMM]) or S0 == NaN or Uniform[#R6IMM] == NaN; (D0 == R3)  D0 = (S0 == Uniform[#R6IMM]) or S0 == NaN or Uniform[#R6IMM] == NaN; (D0 != R3) |

FCMPU.UGE

**Unordered Floating point greater than or equal to Uniform**

| Usage | FCMPU.UGE D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 is NaN or the uniform value indexed by #IMM is NaN or S0 is greater than or equal the uniform value indexed by #IMM, status register flag (SR.F) is set to 1, else it is set to 0. This instruction is equal to the earlier instruction FCMPU.GE.  D0 != R3 (zero register): If S0 is NaN or the uniform value indexed by #IMM is NaN or S0 is greater than or equal the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. |
| Behavior model | SR.F = (S0 >= Uniform[#R6IMM]) or S0 == NaN or Uniform[#R6IMM] == NaN; (D0 == R3)  D0 = (S0 >= Uniform[#R6IMM]) or S0 == NaN or Uniform[#R6IMM] == NaN; (D0 != R3) |

FCMPU.UGT

**Unordered Floating point greater than Uniform**

| Usage | FCMPU.UGT D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 is NaN or the uniform value indexed by #IMM is NaN or S0 is greater than the uniform value indexed by #IMM, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 is NaN or the uniform value indexed by #IMM is NaN or S0 is greater than the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. |
| Behavior model | SR.F = (S0 > Uniform[#R6IMM]) or S0 == NaN or Uniform[#R6IMM] == NaN; (D0 == R3)  D0 = (S0 > Uniform[#R6IMM]) or S0 == NaN or Uniform[#R6IMM] == NaN; (D0 != R3) |

FCMPU.ULE

**Unordered Floating point less than or equal to Uniform**

| Usage | FCMPU.ULE D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 is NaN or the uniform value indexed by #IMM is NaN or S0 is less than or equal to the uniform value indexed by #IMM, status register flag (SR.F) is set to 1, else it is set to 0. This instruction is equal to the earlier instruction FCMPU.LE.  D0 != R3 (zero register): If S0 is NaN or the uniform value indexed by #IMM is NaN or S0 is less than or equal to the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. |
| Behavior model | SR.F = (S0 <= Uniform[#R6IMM]) or S0 == NaN or Uniform[#R6IMM] == NaN; (D0 == R3)  D0 = (S0 <= Uniform[#R6IMM]) or S0 == NaN or Uniform[#R6IMM] == NaN; (D0 != R3) |

FCMPU.ULT

**Unordered Floating point less than Uniform**

| Usage | FCMPU.ULT D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 is NaN or the uniform value indexed by #IMM is NaN or S0 is less than the uniform value indexed by #IMM, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 is NaN or the uniform value indexed by #IMM is NaN or S0 is less than the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. |
| Behavior model | SR.F = (S0 < Uniform[#R6IMM]) or S0 == NaN or Uniform[#R6IMM] == NaN; (D0 == R3)  D0 = (S0 < Uniform[#R6IMM]) or S0 == NaN or Uniform[#R6IMM] == NaN; (D0 != R3) |

FCMPU.UNE

**Unordered Floating not equal to Uniform**

| Usage | FCMPU.UNE Do, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 is NaN or the uniform value indexed by #IMM is NaN or S0 is not equal the uniform value indexed by #IMM, status register flag (SR.F) is set to 1, else it is set to 0. This instruction is equal to the earlier instruction FCMPU.NEQ.  D0 != R3 (zero register): If S0 is NaN or the uniform value indexed by #IMM is NaN or S0 is not equal the uniform value indexed by #IMM, D0 is set to 1, else it is set to 0. |
| Behavior model | SR.F = (S0 != Uniform[#R6IMM]) or S0 == NaN or Uniform[#R6IMM] == NaN; (D0 == R3)  D0 = (S0 != Uniform[#R6IMM]) or S0 == NaN or Uniform[#R6IMM] == NaN; (D0 != R3) |

FCMPU.UO

**Unordered Floating point with Uniform**

| Usage | FCMPU.UO D0, S0, #IMM |
| --- | --- |
| Function | D0 == R3 (zero register): If S0 is NaN or the uniform value indexed by #IMM is NaN, status register flag (SR.F) is set to 1, else it is set to 0.  D0 != R3 (zero register): If S0 is NaN or the uniform value indexed by #IMM is NaN, D0 is set to 1, else it is set to 0. |
| Behavior model | SR.F = S0 == NaN or Uniform[#R6IMM] == NaN; (D0 == R3)  D0 = S0 == NaN or Uniform[#R6IMM] == NaN; (D0 != R3) |

FFLR

**Float FLooR**

| Usage | FFLR D0, S0 |  | |
| --- | --- | --- | --- |
| Function | Stores the integer part of S0 in D0. This integer part is calculated according to a configuration of CR18. The default value of CR 18 is 1 and the logic is integer part of floating value (round towards zero). When CR18 is 0 floating value is rounded to "nearest" integer. The result is stored to D0 and S0 is handled as a floating-point number in the operation. | | |
| Behavior model | D0=floor ((float)S0); | |  |

FFRC

**Float FRaCtion**

| Usage | FFRC D0, S0 |  | |
| --- | --- | --- | --- |
| Function | Subtracts the integer part of S0 from S0. This integer part is calculated according to a configuration of CR18. value of CR 18 is 1 and the logic is integer part of floating value (round towards zero). When CR18 is 0 floating value is rounded to "nearest" integer. The result is stored to D0 and S0 is handled as a floating-point number in the operation. | | |
| Behavior model | D0=S0 - floor ((float)S0); | |  |

FLDCI

**Float Load from Control Register**

| Usage | FLDCI D0, #IMM |  | |
| --- | --- | --- | --- |
| Function | This is an instruction for loading data from a control register.  Writes the data read from the control register at the address specified by #IMM to D0.  The value written to D0 is floating-point.  Specifying #IMM as the number of a control register that does not exist is prohibited. | | |
| Behavior model | D0 = (float)ControlRegister[#IMM]; | |  |

FLDRCI

**Float Load from Relative Coordinate XY with Immediate offset**

| Usage | FLDRCI D0, #IMM1, #IMM2 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer from memory to a register.  X and Y coordinates are specified in R0 and R1 respectively. Reads the value from memory at the coordinates obtained by adding the #IMM1 value to X and the #IMM2 value to Y and stores the value in D0.  The full 32 bits of R0 and the 16 lower-order bits of R1 are handled as signed integers. #IMM1 and #IMM2 need to be specified with 8-bit signed integers.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine.  The value stored in D0 is floating-point. | | |
| Behavior model | D0 = (float) \*(R0+(char)#IMM1, (short)(R1&0xffff)+(char)#IMM2); | | |
| Operand type | D0: 32-bit signed floating-point  #IMM1: 8-bit signed integer  #IMM2: 8-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer | |  |

FLDRCIU

**Float Load from Relative Coordinate XY with Immediate offset, then source Updated**

| Usage | FLDRCIU D0, #IMM1, #IMM2 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer from memory to a register.  X and Y coordinates are specified in R0 and R1 respectively. Reads the value from memory at the coordinates obtained by adding the #IMM1 value to X and the #IMM2 value to Y and stores the value in D0.  The full 32 bits of R0 and the 16 lower-order bits of R1 are handled as signed integers. #IMM1 and #IMM2 need to be specified with 8-bit signed integers. After operation, X(R0) and Y(R1) are updated by adding #IMM1 and #IMM2 respectively.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine.  The value stored in D0 is floating-point. | | |
| Behavior model | D0 = (float) \*(R0+(char)#IMM1, (short)(R1&0xffff)+(char)#IMM2);  R0 = R0+(char)#IMM1; R1 = R1+ (char)#IMM2; | |  |
| Operand type | D0: 32-bit signed floating-point  #IMM1: 8-bit signed integer  #IMM2: 8-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer | |  |

FLDRXY

**Float Load from Relative Coordinate XY with register offset**

| Usage | FLDRXY D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer from memory to a register.  X and Y coordinates are specified in R0 and R1 respectively. Reads the value from memory at the coordinates obtained by adding the S0 value to X and the S1 value to Y and stores the value in D0.  The full 32 bits of R0 and S0 and the 16 lower-order bits of R1 and S1 are handled as signed integers.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine.  The value stored in D0 is floating-point. | | |
| Behavior model | D0 = (float) \*(R0+S0, (short)(R1&0xffff)+(short)(S1&0xffff)); | |  |
| Operand type | D0: 32-bit signed floating-point  S0: 32-bit signed integer  S1: 16-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer | |  |

FLDRXYU

**Float Load from Relative Coordinate XY with register offset, then source Updated**

| Usage | FLDRXYU D0, S0, S1 | |  |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer from memory to a register.  X and Y coordinates are specified in R0 and R1 respectively. Reads the value from memory at the coordinates obtained by adding the S0 value to X and the S1 value to Y and stores the value in D0.  The full 32 bits of R0 and S0 and the 16 lower-order bits of R1 and S1 are handled as signed integers. After operation, X(R0) and Y(R1) are updated by adding S0 and S1 respectively.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine.  The value stored in D0 is floating-point. | | |
| Behavior model | D0 = (float) \*(R0+S0, (short)(R1&0xffff)+(short)(S1&0xffff));  R0 = R0+S0; R1 = (short)(R1&0xffff)+(short)(S1&0xffff); |  | |
| Operand type | D0: 32-bit signed floating-point  S0: 32-bit signed integer  S1: 16-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer |  | |

FMAD

**Float Multiply and ADd**

| Usage | FMAD D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Adds D0 and the product of S0 and S1, then stores the result in D0.  D0, S0 and S1 are handled as floating-point numbers in the operation.  The result of this instruction matches that of the similar operation obtained by combining the FMUL and FADD instructions. | | |
| Behavior model | D0 = S0\*S1 + D0; | |  |

FMADU

**Float Multiply and Add Uniform**

| Usage | FMADU D0, S0, #IMM |
| --- | --- |
| Function | Adds D0 and the product of S0 and the uniform value indexed by #IMM, then stores the result in D0.  D0, S0 and S1 are handled as floating-point numbers in the operation.  The result of this instruction matches that of the similar operation obtained by combining the FMULU and FADD instructions. |
| Behavior model | D0 = S0\*Uniform[#R6IMM] + D0; |

FMAX

**Float MAXimum**

| Usage | FMAX D0, S0, S1 |
| --- | --- |
| Function | Selects the larger value between S0 and S1, then stores the result in D0.  S0 and S1 are handled as floating-point numbers in the operation. |
| Behavior model | D0 = (S0 > S1) ? S0 : S1; |

FMAXU

**Float MAXimum compared to Uniform**

| Usage | FMAXU D0, S0, #IMM |
| --- | --- |
| Function | Selects the larger value between S0 and the uniform value indexed by #IMM, then stores the result in D0.  S0 and the uniform value are handled as floating-point numbers in the operation. |
| Behavior model | D0 = (S0 > Uniform[#R6IMM]) ? S0 : Uniform[#R6IMM]); |

FMIN

**Float MINimum**

| Usage | FMIN D0, S0, S1 |
| --- | --- |
| Function | Selects the smaller value between S0 and S1, then stores the result in D0.  S0 and S1 are handled as floating-point numbers in the operation. |
| Behavior model | D0 = (S0 > S1) ? S1 : S0; |

FMINU

**Float MINimum compared to Uniform**

| Usage | FMINU D0, S0, #IMM |
| --- | --- |
| Function | Selects the smaller value between S0 and the uniform value indexed by #IMM, then stores the result in D0.  S0 and the uniform value are handled as floating-point numbers in the operation. |
| Behavior model | D0 = (S0 > Uniform[#R6IMM]) ? Uniform[#R6IMM]) : S0; |

FMP

**Float Middle Point**

| Usage | FMP D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Divides the sum of S0 and S1 by 2.0, then stores the result in D0.  S0 and S1 are handled as floating-point numbers in the operation. | | |
| Behavior model | D0 = (S0 + S1) / 2.0; | |  |

FMSU

**Float Multiply and Sub**

| Usage | FMSU D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Subtracts the product of S0 and S1 from D0, then stores the result in D0.  D0, S0 and S1 are handled as floating-point numbers in the operation.  The result of this instruction matches that of the similar operation obtained by combining the FMUL and FSUB instructions. | | |
| Behavior model | D0 = -S0\*S1 + D0; | |  |

FMSUU

**Float Multiply and Sub Uniform**

| Usage | FMSUU D0, S0, #IMM |
| --- | --- |
| Function | Subtracts the product of S0 and the uniform value indexed by #IMM from D0, then stores the result in D0.  D0, S0 and S1 are handled as floating-point numbers in the operation.  The result of this instruction matches that of the similar operation obtained by combining the FMULU and FSUB instructions. |
| Behavior model | D0 = -S0\*Uniform[#R6IMM] + D0; |

FMUL

**Float MULTiply**

| Usage | FMUL D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Multiplies S0 and S1, then stores the result in D0.  S0 and S1 are handled as floating-point numbers in the operation. | | |
| Behavior model | D0 = S0 \* S1; | |  |

FMULU

**Float MULTiply by Uniform**

| Usage | FMULU D0, S0, #IMM |
| --- | --- |
| Function | Multiplies S0 and the uniform value indexed by #IMM, then stores the result in D0.  S0 and the uniform value are handled as floating-point numbers in the operation. |
| Behavior model | D0 = S0 \* Uniform[#R6IMM]; |

FSTC

**Float STore to Control Register**

| Usage | FSTC D0, S0 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction for storing data in a control register.  The S0 value is converted to integer and written to the control register D0. Specifying D0 as a control register that does not exist is prohibited. | | |
| Behavior model | ControlRegisterD0 = (int)S0; | |  |

FSTRCI

**Float STore to Relative Coordinate with Immediate offset**

| Usage | FSTRCI D0, #IMM1, #IMM2 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer from a register to memory.  Stores the D0 value in memory at X and Y coordinates which are specified in R0 and R1 respectively, with the #IMM1 value added to X and the #IMM2 value added to Y.  The full 32 bits of R0 and the 16 lower-order bits of R1 are handled as signed integers. #IMM1 and #IMM2 need to be specified with 8-bit signed integers.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine.  D0 is handled as a floating-point number. | | |
| Behavior model | @(R0+(char)#IMM1, (short)(R1&0xffff)+(char)#IMM2)= D0 | | |
| Operand type | D0: 32-bit signed floating-point  #IMM1: 8-bit signed integer  #IMM2: 8-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer | |  |

FSTRCIU

**Float STore to Relative Coordinate with Immediate offset, then source Updated**

| Usage | FSTRCIU D0, #IMM1, #IMM2 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer from a register to memory.  Stores the D0 value in memory at X and Y coordinates which are specified in R0 and R1 respectively, with the #IMM1 value added to X and the #IMM2 value added to Y.  The full 32 bits of R0 and the 16 lower-order bits of R1 are handled as signed integers. #IMM1 and #IMM2 need to be specified with 8-bit signed integers. After operation, X(R0) and Y(R1) are updated by adding #IMM1 and #IMM2 respectively.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine.  D0 is handled as a floating-point number. | | |
| Behavior model | @(R0+(char)#IMM1, (short)(R1&0xffff)+(char)#IMM2) = D0;  R0 = R0+(char)#IMM1; R1 = R1+ (char)#IMM2; | |  |
| Operand type | D0: 32-bit signed floating-point  #IMM1: 8-bit signed integer  #IMM2: 8-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer | |  |

FSTRXY

**Float STore to Relative Coordinate XY with register offset**

| Usage | FSTRXY D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer from a register to memory.  Stores the D0 value in memory at X and Y coordinates which are specified in R0 and R1 respectively, with the S0 value added to X and the S1 value added to Y.  The full 32 bits of R0 and S0 and the 16 lower-order bits of R1 and S1 are handled as signed integers.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine.  D0 is handled as a floating-point number. | | |
| Behavior model | @(R0+S0, (short)(R1&0xffff)+(short)(S1&0xffff)) = D0; | | |
| Operand type | D0: 32-bit signed floating-point  S0: 32-bit signed integer  S1: 16-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer | |  |

FSTRXYU

**Float STore to Relative Coordinate XY with register offset, then source Updated**

| Usage | FSTRXYU D0, S0, S1 |  |
| --- | --- | --- |
| Function | This is an instruction for data transfer from a register to memory.  Stores the D0 value in memory at X and Y coordinates which are specified in R0 and R1 respectively, with the S0 value added to X and the S1 value added to Y.  The full 32 bits of R0 and S0 and the 16 lower-order bits of R1 and S1 are handled as signed integers. After operation, X(R0) and Y(R1) are updated by adding S0 and S1 respectively.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine.  D0 is handled as a floating-point number. | |
| Behavior model | @(R0+S0, (short)(R1&0xffff)+(short)(S1&0xffff)) = D0;  R0 = R0+S0; R1 = (short)(R1&0xffff)+(short)(S1&0xffff); |  |
| Operand type | D0: 32-bit signed floating-point  S0: 32-bit signed integer  S1: 16-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer |  |

FSUB

**Float SUB**

| Usage | FSUB D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Subtracts S1 from S0, then stores the result in D0.  S0 and S1 are handled as floating-point numbers in the operation. | | |
| Behavior model | D0 = S0 - S1; | |  |

FSUBU

**Float SUB Uniform from**

| Usage | FSUBU D0, S0, #IMM |
| --- | --- |
| Function | Subtracts the uniform value indexed by #IMM from S0, then stores the result in D0.  S0 and the uniform value are handled as floating-point numbers in the operation. |
| Behavior model | D0 = S0 - Uniform[#R6IMM]; |

GADDBS

**Group add and sum up for 8-bit signed input data**

| Usage | GADDBS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+S0[0]+S1[0]+S0[1]+S1[1]+S0[2]+S1[2]+S0[3]+S1[3]+S0[4]+S1[4]+S0[5]+S1[5]+S0[6]+S1[6]+S0[7]+S1[7]  S0 and S1 are signed 8-bit vectors with each 8 elements. The instruction is performing an addition of each element between the vectors S0 and S1 and adding all results to the input D0 before storing back the result to D0. D0 is a 32-bit signed value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers.  Hardware does not take care any overflow handlings, so software must take care of it. | | |
| Behavior model | D0 = D0+S0[0]+S1[0]+S0[1]+S1[1]+S0[2]+S1[2]+S0[3]+S1[3]+S0[4]+S1[4]+S0[5]+S1[5]+S0[6]+S1[6]+S0[7]+S1[7] | |  |
| Operand type | D0: 32-bit signed integer  S0: 8 \* 8-bit signed integer  S1: 8 \* 8-bit signed integer | |  |

GADDBU

**Group add and sum up for 8-bit unsigned input data**

| Usage | GADDBU D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+S0[0]+S1[0]+S0[1]+S1[1]+S0[2]+S1[2]+S0[3]+S1[3]+S0[4]+S1[4]+S0[5]+S1[5]+S0[6]+S1[6]+S0[7]+S1[7]  S0 and S1 are unsigned 8-bit vectors with each 8 elements. The instruction is performing an addition of each element between the vectors S0 and S1 and adding all results to the input D0 before storing back the result to D0. D0 is a 32-bit signed value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers.  Hardware does not take care any overflow handlings, so software must take care of it. | | |
| Behavior model | D0 = D0+S0[0]+S1[0]+S0[1]+S1[1]+S0[2]+S1[2]+S0[3]+S1[3]+S0[4]+S1[4]+S0[5]+S1[5]+S0[6]+S1[6]+S0[7]+S1[7] | |  |
| Operand type | D0: 32-bit signed integer  S0: 8 \* 8-bit unsigned integer  S1: 8 \* 8-bit unsigned integer | |  |

GADDHS

**Group add and sum up for 16-bit signed input data**

| Usage | GADDHS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+S0[0]+S1[0]+S0[1]+S1[1]+S0[2]+S1[2]+S0[3]+S1[3]  S0 and S1 are signed 16-bit vectors with each 4 elements. The instruction is performing an addition of each element between the vectors S0 and S1 and adding all results to the input D0 before storing back the result to D0.D0 is a 32-bit signed value.S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers.  Hardware does not take care any overflow handlings, so software must take care of it. | | |
| Behavior model | D0 = D0+S0[0]+S1[0]+S0[1]+S1[1]+S0[2]+S1[2]+S0[3]+S1[3] | |  |
| Operand type | D0: 32-bit signed integer  S0: 4 \* 16-bit signed integer  S1: 4 \* 16-bit signed integer | |  |

GADDHU

**Group add and sum up for 16-bit unsigned input data**

| Usage | GADDHU D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+S0[0]+S1[0]+S0[1]+S1[1]+S0[2]+S1[2]+S0[3]+S1[3]  S0 and S1 are unsigned 16-bit vectors with each 4 elements. The instruction is performing an addition of each element between the vectors S0 and S1 and adding all results to the input D0 before storing back the result to D0.D0 is a 32-bit signed value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers.  Hardware does not take care any overflow handlings, so software must take care of it. | | |
| Behavior model | D0 = D0+S0[0]+S1[0]+S0[1]+S1[1]+S0[2]+S1[2]+S0[3]+S1[3] | |  |
| Operand type | D0: 32-bit signed integer  S0: 4 \* 16-bit unsigned integer  S1: 4 \* 16-bit unsigned integer | |  |

GMADBS

**Group multiply and add for 8-bit signed input data**

| Usage | GMADBS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3]+S0[4]\*S1[4]+S0[5]\*S1[5]+S0[6]\*S1[6]+S0[7]\*S1[7]  S0 and S1 are signed 8-bit vectors with each 8 elements. The instruction is performing a multiplication of each element between the vectors S0 and S1 and adding all results to the input D0 before storing back the result to D0. D0 is a 32-bit signed value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers.  Hardware does not take care any overflow handlings, so software must take care of it. | | |
| Behavior model | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3]+S0[4]\*S1[4]+S0[5]\*S1[5]+S0[6]\*S1[6]+S0[7]\*S1[7] | |  |
| Operand type | D0: 32-bit signed integer  S0: 8 \* 8-bit signed integer  S1: 8 \* 8-bit signed integer | |  |

GMADBSL

**Group multiply and add for 8-bit signed input data long**

| Usage | GMADBSL D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3]+S0[4]\*S1[4]+S0[5]\*S1[5]+S0[6]\*S1[6]+S0[7]\*S1[7]  S0 and S1 are signed 8-bit vectors with each 8 elements. The instruction is performing a multiplication of each element between the vectors S0 and S1 and adding all results to the input D0 before storing back the result to D0. D0 is a 64-bit signed value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers.  Hardware does not take care any overflow handlings, so software must take care of it. | | |
| Behavior model | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3]+S0[4]\*S1[4]+S0[5]\*S1[5]+S0[6]\*S1[6]+S0[7]\*S1[7] | |  |
| Operand type | D0: 64-bit signed integer  S0: 8 \* 8-bit signed integer  S1: 8 \* 8-bit signed integer | |  |

GMADBU

**Group multiply and add for 8-bit unsigned input data**

| Usage | GMADBU D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3]+S0[4]\*S1[4]+S0[5]\*S1[5]+S0[6]\*S1[6]+S0[7]\*S1[7]  S0 and S1 are unsigned 8-bit vectors with each 8 elements. The instruction is performing a multiplication of each element between the vectors S0 and S1 and adding all results to the input D0 before storing back the result to D0. D0 is a 32-bit signed value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers.  Hardware does not take care any overflow handlings, so software must take care of it. | | |
| Behavior model | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3]+S0[4]\*S1[4]+S0[5]\*S1[5]+S0[6]\*S1[6]+S0[7]\*S1[7] | |  |
| Operand type | D0: 32-bit signed integer  S0: 8 \* 8-bit unsigned integer  S1: 8 \* 8-bit unsigned integer | |  |

GMADBUL

**Group multiply and add for 8-bit unsigned input data long**

| Usage | GMADBUL D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3]+S0[4]\*S1[4]+S0[5]\*S1[5]+S0[6]\*S1[6]+S0[7]\*S1[7]  S0 and S1 are unsigned 8-bit vectors with each 8 elements. The instruction is performing a multiplication of each element between the vectors S0 and S1 and adding all results to the input D0 before storing back the result to D0. D0 is a 64-bit signed value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers.  Hardware does not take care any overflow handlings, so software must take care of it. | | |
| Behavior model | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3]+S0[4]\*S1[4]+S0[5]\*S1[5]+S0[6]\*S1[6]+S0[7]\*S1[7] | |  |
| Operand type | D0: 64-bit unsigned integer  S0: 8 \* 8-bit unsigned integer  S1: 8 \* 8-bit unsigned integer | |  |

GMADHS

**Group multiply and add for 16-bit signed input data**

| Usage | GMADHS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3]  S0 and S1 are signed 16-bit vectors with each 4 elements. The instruction is performing a multiplication of each element between the vectors S0 and S1 and adding all results to the input D0 before storing back the result to D0. D0 is a 32-bit signed value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers.  Hardware does not take care any overflow handlings, so software must take care of it. | | |
| Behavior model | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3] | |  |
| Operand type | D0: 32-bit signed integer  S0: 4 \* 16-bit signed integer  S1: 4 \* 16-bit signed integer | |  |

GMADHSL

**Group multiply and add for 16-bit signed input data long**

| Usage | GMADHSL D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3]  S0 and S1 are signed 16-bit vectors with each 4 elements. The instruction is performing a multiplication of each element between the vectors S0 and S1 and adding all results to the input D0 before storing back the result to D0. D0 is a 64-bit signed value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers.  Hardware does not take care any overflow handlings, so software must take care of it. | | |
| Behavior model | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3] | |  |
| Operand type | D0: 64-bit signed integer  S0: 4 \* 16-bit signed integer  S1: 4 \* 16-bit signed integer | |  |

GMADHU

**Group multiply and add for 16-bit unsigned input data**

| Usage | GMADHU D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3]  S0 and S1 are unsigned 16-bit vectors with each 4 elements. The instruction is performing a multiplication of each element between the vectors S0 and S1 and adding all results to the input D0 before storing back the result to D0. D0 is a 32-bit unsigned value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers.  Hardware does not take care any overflow handlings, so software must take care of it. | | |
| Behavior model | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3] | |  |
| Operand type | D0: 32-bit unsigned integer  S0: 4 \* 16-bit unsigned integer  S1: 4 \* 16-bit unsigned integer | |  |

GMADHUL

**Group multiply and add for 16-bit unsigned input data long**

| Usage | GMADHUL D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3]  S0 and S1 are unsigned 16-bit vectors with each 4 elements. The instruction is performing a multiplication of each element between the vectors S0 and S1 and adding all results to the input D0 before storing back the result to D0. D0 is a 64-bit unsigned value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers.  Hardware does not take care any overflow handlings, so software must take care of it. | | |
| Behavior model | D0 = D0+S0[0]\*S1[0]+S0[1]\*S1[1]+S0[2]\*S1[2]+S0[3]\*S1[3] | |  |
| Operand type | D0: 64-bit unsigned integer  S0: 4 \* 16-bit unsigned integer  S1: 4 \* 16-bit unsigned integer | |  |

GSADBS

**Group SAD and sum up for 8-bit signed input data**

| Usage | GSADBS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+|S0[0]-S1[0]|+|S0[1]-S1[1]|+|S0[2]-S1[2]|+|S0[3]-S1[3]|+|S0[4]-S1[4]|+|S0[5]-S1[5]|+|S0[6]-S1[6]|+|S0[7]-S1[7]|  S0 and S1 are signed 8-bit vectors with each 8 elements. The instruction is performing difference calculation of each element of vector S1 from vector S0 and adding all results to the input D0 before storing back the result to D0. D0 is a 32-bit signed value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers. | | |
| Behavior model | D0 = D0+|S0[0]-S1[0]|+|S0[1]-S1[1]|+|S0[2]-S1[2]|+|S0[3]-S1[3]|+|S0[4]-S1[4]|+|S0[5]-S1[5]|+|S0[6]-S1[6]|+|S0[7]-S1[7]| | |  |
| Operand type | D0: 32-bit signed integer  S0: 8 \* 8-bit signed integer  S1: 8 \* 8-bit signed integer | |  |

GSADBU

**Group SAD and sum up for 8-bit unsigned input data**

| Usage | GSADBU D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+|S0[0]-S1[0]|+|S0[1]-S1[1]|+|S0[2]-S1[2]|+|S0[3]-S1[3]|+|S0[4]-S1[4]|+|S0[5]-S1[5]|+|S0[6]-S1[6]|+|S0[7]-S1[7]|  S0 and S1 are unsigned 8-bit vectors with each 8 elements. The instruction is performing a difference calculation of each element of vector S1 from vector S0 and adding all results to the input D0 before storing back the result to D0. D0 is a 32-bit unsigned value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers. | | |
| Behavior model | D0 = D0+|S0[0]-S1[0]|+|S0[1]-S1[1]|+|S0[2]-S1[2]|+|S0[3]-S1[3]|+|S0[4]-S1[4]|+|S0[5]-S1[5]|+|S0[6]-S1[6]|+|S0[7]-S1[7]| | |  |
| Operand type | D0: 32-bit unsigned integer  S0: 8 \* 8-bit unsigned integer  S1: 8 \* 8-bit unsigned integer | |  |

GSADHS

**Group SAD and sum up for 16-bit signed input data**

| Usage | GSADHS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+|S0[0]-S1[0]|+|S0[1]-S1[1]|+|S0[2]-S1[2]|+|S0[3]-S1[3]|  S0 and S1 are signed 16-bit vectors with each 4 elements. The instruction is performing a difference calculation of each element of vector S1 from vector S0 and adding all results to the input D0 before storing back the result to D0.D0 is a 32-bit signed value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers. | | |
| Behavior model | D0 = D0+|S0[0]-S1[0]|+|S0[1]-S1[1]|+|S0[2]-S1[2]|+|S0[3]-S1[3]| | |  |
| Operand type | D0: 32-bit signed integer  S0: 4 \* 16-bit signed integer  S1: 4 \* 16-bit signed integer | |  |

GSADHU

**Group SAD and sum up for 16-bit unsigned input data**

| Usage | GSADHU D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+|S0[0]-S1[0]|+|S0[1]-S1[1]|+|S0[2]-S1[2]|+|S0[3]-S1[3]|  S0 and S1 are unsigned 16-bit vectors with each 4 elements. The instruction is performing difference calculation of each element of vector S1 from vector S0 and adding all results to the input D0 before storing back the result to D0.D0 is a 32-bit signed value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers. | | |
| Behavior model | D0 = D0+|S0[0]-S1[0]|+|S0[1]-S1[1]|+|S0[2]-S1[2]|+|S0[3]-S1[3]| | |  |
| Operand type | D0: 32-bit signed integer  S0: 4 \* 16-bit unsigned integer  S1: 4 \* 16-bit unsigned integer | |  |
|  |  | |  |

GSUBBS

**Group sub and sum up for 8-bit signed input data**

| Usage | GSUBBS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+(S0[0]-S1[0])+(S0[1]-S1[1])+(S0[2]-S1[2])+(S0[3]-S1[3])+(S0[4]-S1[4])+(S0[5]-S1[5])+(S0[6]-S1[6])+(S0[7]-S1[7])  S0 and S1 are signed 8-bit vectors with each 8 elements. The instruction is performing a subtraction of each element of vector S1 from vector S0 and adding all results to the input D0 before storing back the result to D0. D0 is a 32-bit signed value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers. | | |
| Behavior model | D0 = D0+(S0[0]-S1[0])+(S0[1]-S1[1])+(S0[2]-S1[2])+(S0[3]-S1[3])+(S0[4]-S1[4])+(S0[5]-S1[5])+(S0[6]-S1[6])+(S0[7]-S1[7]) | |  |
| Operand type | D0: 32-bit signed integer  S0: 8 \* 8-bit signed integer  S1: 8 \* 8-bit signed integer | |  |

GSUBBU

**Group sub and sum up for 8-bit unsigned input data**

| Usage | GSUBBU D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+(S0[0]-S1[0])+(S0[1]-S1[1])+(S0[2]-S1[2])+(S0[3]-S1[3])+(S0[4]-S1[4])+(S0[5]-S1[5])+(S0[6]-S1[6])+(S0[7]-S1[7])  S0 and S1 are unsigned 8-bit vectors with each 8 elements. The instruction is performing a subtraction of each element of vector S1 from vector S0 and adding all results to the input D0 before storing back the result to D0. D0 is a 32-bit signed value. S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers. | | |
| Behavior model | D0 = D0+(S0[0]-S1[0])+(S0[1]-S1[1])+(S0[2]-S1[2])+(S0[3]-S1[3])+(S0[4]-S1[4])+(S0[5]-S1[5])+(S0[6]-S1[6])+(S0[7]-S1[7]) | |  |
| Operand type | D0: 32-bit signed integer  S0: 8 \* 8-bit unsigned integer  S1: 8 \* 8-bit unsigned integer | |  |

GSUBHS

**Group sub and sum up for 16-bit signed input data**

| Usage | GSUBHS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+(S0[0]-S1[0])+(S0[1]-S1[1])+(S0[2]-S1[2])+(S0[3]-S1[3])  S0 and S1 are signed 16-bit vectors with each 4 elements. The instruction is performing a subtraction of each element of vector S1 from vector S0 and adding all results to the input D0 before storing back the result to D0.D0 is a 32-bit signed value.S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers. | | |
| Behavior model | D0 = D0+(S0[0]-S1[0])+(S0[1]-S1[1])+(S0[2]-S1[2])+(S0[3]-S1[3]) | |  |
| Operand type | D0: 32-bit signed integer  S0: 4 \* 16-bit signed integer  S1: 4 \* 16-bit signed integer | |  |

GSUBHU

**Group sub and sum up for 16-bit unsigned input data**

| Usage | GSUBHU D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0+(S0[0]-S1[0])+(S0[1]-S1[1])+(S0[2]-S1[2])+(S0[3]-S1[3])  S0 and S1 are unsigned 16-bit vectors with each 4 elements. The instruction is performing a subtraction of each element of vector S1 from vector S0 and adding all results to the input D0 before storing back the result to D0.D0 is a 32-bit signed value.S0 and S1 are 64-bit vectors, which are specified by the even of 2 consecutive 32-bit registers. | | |
| Behavior model | D0 = D0+(S0[0]-S1[0])+(S0[1]-S1[1])+(S0[2]-S1[2])+(S0[3]-S1[3]) | |  |
| Operand type | D0: 32-bit signed integer  S0: 4 \* 16-bit unsigned integer  S1: 4 \* 16-bit unsigned integer | |  |

I2F

**Integer to Float**

| Usage | I2F D0, S0 |  | |
| --- | --- | --- | --- |
| Function | Assumes the S0 value is an integer, converts it into an IEEE754 single-precision floating-point number, then stores the result in D0. The rounding method can be changed by the control register. | | |
| Behavior model | D0 = float((int)S0); | |  |

INT

**Interrupt**

| Usage | INT |  | |
| --- | --- | --- | --- |
| Function | Interrupts current processing in a thread and waits for resuming by host. | | |
| Behavior model | - | |  |

INV

**INVerse**

| Usage | INV D0, S0 |  | |
| --- | --- | --- | --- |
| Function | Treats S0 as a 32-bit array, inverts all bits of this array, and stores the result in D0. | | |
| Behavior model | D0 = ~S0; | |  |

INV64

**INVerse 64-bit values**

| Usage | INV64 D0, S0 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to invert a 64-bit (8 times 8-bit or 4 times 16-bit) value. | | |
| Behavior model | D0[n] = !S0[n], n = 0..63; | |  |
| Operand type | D0: 64-bit integer  S0: 64-bit integer | |  |

L2F

**Long to Float**

| Usage | L2F D0, S0 |  | |
| --- | --- | --- | --- |
| Function | Convert 64-bit value stored in register pair specified by even register S0 to IEEE754 single floating point format, then store the result to register D0. The rounding method is specified by the control register. | | |
| Behavior model | D0 = float(S0); | |  |

LD

**Load word**

| Usage | LD D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | It is a data transfer instruction from memory to the register.  The 32-bit data is loaded to register D0 from the 32-bit aligned address specified by the sum of the values S0 and S1\*4. S1 is specifying a 32-bit word offset. | | |
| Behavior model | D0 = \*(S0+S1\*4); | |  |

LDBS

**Load byte with signed extension**

| Usage | LDBS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | It is a data transfer instruction from memory to the register.  The 8-bit data is loaded and signed 32-bit extended to register D0 from the 8-bit aligned address specified by the sum of the values S0 and S1. S1 is specifying a signed byte offset. | | |
| Behavior model | D0 = (signed int)\*(S0+S1); | |  |

LDBSI

**Load byte with signed extension and immediate offset**

| Usage | LDBSI D0, S0, #IMM |  |
| --- | --- | --- |
| Function | It is a data transfer instruction from memory to the register.  The 8-bit data is loaded and signed 32-bit extended to register D0 from the 8-bit aligned address specified by the sum of the values S0 and #IMM. The 8-bit signed immediate value is specifying a byte offset. | |
| Behavior model | D0 = (signed int)\*(S0+#IMM); |  |

LDBU

**Load byte with unsigned extension**

| Usage | LDBU D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | It is a data transfer instruction from memory to the register.  The 8-bit data is loaded and unsigned 32-bit extended to register D0 from the 8-bit aligned address specified by the sum of the values S0 and S1. S1 is specifying a signed byte offset. | | |
| Behavior model | D0 = (unsigned int)\*(S0+S1); | |  |

LDBUI

**Load byte with unsigned extension and immediate offset**

| Usage | LDBUI D0, S0, #IMM | |  |
| --- | --- | --- | --- |
| Function | It is a data transfer instruction from memory to the register.  The 8-bit data is loaded and unsigned 32-bit extended to register D0 from the 8-bit aligned address specified by the sum of the values S0 and #IMM. The 8-bit signed immediate value is specifying a byte offset. | | |
| Behavior model | D0 = (unsigned int)\*(S0+#IMM); |  | |

LDCI

**Load from Control Register**

| Usage | LDCI D0, #IMM |  | |
| --- | --- | --- | --- |
| Function | This is an instruction for loading data from a control register.  Writes the data read from the control register specified by #IMM to the general purpose register D0. The value written to D0 is an integer.  Specifying #IMM as the number of a control register that does not exist is prohibited. | | |
| Behavior model | D0 = ControlRegister[#IMM]; | |  |
| Operand type | D0: General purpose register  #IMM: 8-bit unsigned integer | |  |

LDHS

**Load half word with signed extension**

| Usage | LDHS D0, S0, S1 |  |
| --- | --- | --- |
| Function | It is a data transfer instruction from memory to the register.  The 16-bit data is loaded and signed 32-bit extended to register D0 from the 16-bit aligned address specified by the sum of the values S0 and S1\*2. S1 is specifying a 16-bit short offset. | |
| Behavior model | D0 = (signed int)\*(S0+S1\*2); |  |

LDHSI

**Load half word with signed extension and immediate offset**

| Usage | LDHSI D0, S0, #ASMIMM |  |
| --- | --- | --- |
| Function | It is a data transfer instruction from memory to the register.  The 16-bit data is loaded and signed 32-bit extended to register D0 from the 16-bit aligned address specified by the sum of the values S0 and #ASMIMM. The 8-bit signed immediate value is specifying a byte offset. | |
| Behavior model | #IMM = #ASMIMM>>1  D0 = (signed int)\*(S0+#IMM\*2); |  |

LDHU

**Load half word with unsigned extension**

| Usage | LDHU D0, S0, S1 |  |
| --- | --- | --- |
| Function | It is a data transfer instruction from memory to the register.  The 16-bit data is loaded and unsigned 32-bit extended to register D0 from the 16-bit aligned address specified by the sum of the values S0 and S1\*2. S1 is specifying a 16-bit short offset. | |
| Behavior model | D0 = (unsigned int)\*(S0+S1\*2); |  |

LDHUI

**Load half word with unsigned extension and immediate offset**

| Usage | LDHUI D0, S0, #ASMIMM |  |
| --- | --- | --- |
| Function | It is a data transfer instruction from memory to the register.  The 16-bit data is loaded and unsigned 32-bit extended to register D0 from the 16-bit aligned address specified by the sum of the values S0 and #ASMIMM. The 8-bit signed immediate value is specifying a byte offset. | |
| Behavior model | #IMM = #ASMIMM>>1  D0 = (unsigned int)\*(S0+#IMM\*2); |  |

LDI

**Load word with immediate offset**

| Usage | LDI D0, S0, #ASMIMM |  |
| --- | --- | --- |
| Function | It is a data transfer instruction from memory to the register.  The 32-bit data is loaded to register D0 from the 32-bit aligned address specified by the sum of the values S0 and #ASMIMM. The 8-bit signed immediate value is specifying a byte offset. | |
| Behavior model | #IMM = #ASMIMM>>2  D0 = \*(S0+#IMM\*4); |  |

LDLL

**Load Long from Local working memory**

| Usage | LDLL D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This instruction is available only for V3M Ver.2.0, V3H and V3U.  This is an instruction to transfer 64-bit data from local working memory to a 64-bit pair register. This instruction supports the common and non-common address space.  Writes the data read from the address specified by adding the S1×8 value to address S0 of local working memory to D0.  When this instruction is used for other memory accesses, this instruction transfer 32-bit from the specified memory address into the even register and set the odd register to 0.  The 64-bit access to LWM must be aligned, when unaligned, function is prohibited.  D0 is a 64-bit pair register, always the lower even register is specified, which holds the lower 32-bit. S0 and S1 are 32-bit registers. | | |
| Behavior model | D0 = \*(S0 + S1\*8); | |  |
| Operand type | D0: 64-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  S1: 32-bit signed integer | |  |

LDLLI

**Load Long from Local working memory with Immediate offset**

| Usage | LDLLI D0, S0, #ASMIMM |  | |
| --- | --- | --- | --- |
| Function | This instruction is available only for V3M Ver.2.0, V3H and V3U.  This is an instruction to transfer 64-bit data from local working memory to a 64-bit pair register. This instruction supports the common and non-common address space.  Writes the data read from the address specified by adding the #IMM×8 value to address S0 of local working memory to D0.  When this instruction is used for other memory accesses, this instruction transfer 32-bit from the specified memory address into the even register and set the odd register to 0.  The 64-bit access to LWM must be aligned, when unaligned, function is prohibited.  D0 is a 64-bit pair register, always the lower even register is specified, which holds the lower 32-bit. S0 is a 32-bit register. | | |
| Behavior model | #IMM = #ASMIMM>>3  D0 = \*(S0 + (char)#IMM\*8); | |  |
| Operand type | D0: 64-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  #IMM: 8-bit signed integer | |  |

LDLRB

**LoaD Large Register Block**

| Usage | LDLRB D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to transfer 512-bit data from local working memory to a 512-bit large register block. This instruction supports only the common address space.  Writes the data read from the address specified by adding S1\*64 to address S0 of local working memory to D0.  The access to LWM must be 64-bit aligned, when unaligned, function is prohibited.  D0 is specifying the lowest register of the register block. Allowed values are 16 (for register block R16-R31) and 32 (for register block R32-R47) | | |
| Behavior model | D0 = \*(S0 + S1\*64); | |  |
| Operand type | D0: 512-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  S1: 16-bit signed integer | |  |

LDLRBI

**LoaD Large Register Block with Immediate offset**

| Usage | LDLRBI D0, S0, #IMM |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to transfer 512-bit data from local working memory to a 512-bit large register block. This instruction supports only the common address space.  Writes the data read from the address specified by adding #IMM\*64 to address S0 of local working memory to D0.  The access to LWM must be 64-bit aligned, when unaligned, function is prohibited.  D0 is specifying the lowest register of the register block. Allowed values are 16 (for register block R16-R31) and 32 (for register block R32-R47) | | |
| Behavior model | D0 = \*(S0 + #IMM\*64); | |  |
| Operand type | D0: 512-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  #IMM: 8-bit signed integer | |  |

LDLRBC

**LoaD Large Register Block with LPC offset**

| Usage | LDLRBC D0, S0 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to transfer 512-bit data from local working memory to a 512-bit large register block. This instruction supports only the common address space.  Writes the data read from the address specified by adding LPC (loop counter)\*64 to address S0 of local working memory to D0.  The access to LWM must be 64-bit aligned, when unaligned, function is prohibited.  D0 is specifying the lowest register of the register block. Allowed values are 16 (for register block R16-R31) and 32 (for register block R32-R47) | | |
| Behavior model | D0 = \*(S0 + LPC\*64); | |  |
| Operand type | D0: 512-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  LPC: 9-bit signed integer | |  |

LDRCI

**Load from Relative Coordinate XY with Immediate offset**

| Usage | LDRCI D0, #IMM1, #IMM2 | |  |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer from memory to a register.  X and Y coordinates are specified in R0 and R1 respectively. Reads the value from memory at the coordinates obtained by adding the #IMM1 value to X and the #IMM2 value to Y and stores the value in D0.  The full 32 bits of R0 and the 16 lower-order bits of R1 are handled as signed integers. #IMM1 and #IMM2 need to be specified with 8-bit signed integers.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine.  The value stored in D0 is an integer. | | |
| Behavior model | D0 = (int)\* (R0+(char)#IMM1, (short)(R1&0xffff)+(char)#IMM2); | | |
| Operand type | D0: 32-bit signed integer  #IMM1: 8-bit signed integer  #IMM2: 8-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer |  | |

LDRCIU

**Load from Relative Coordinate XY with Immediate offset, then source Updated**

| Usage | LDRCIU D0, #IMM1, #IMM2 | |  |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer from memory to a register.  X and Y coordinates are specified in R0 and R1 respectively. Reads the value from memory at the coordinates obtained by adding the #IMM1 value to X and the #IMM2 value to Y and stores the value in D0.  The full 32 bits of R0 and the 16 lower-order bits of R1 are handled as signed integers. #IMM1 and #IMM2 need to be specified with 8-bit signed integers. After operation, X(R0) and Y(R1) are updated by adding #IMM1 and #IMM2 respectively.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine.  The value stored in D0 is an integer. | | |
| Behavior model | D0 = (int)\* (R0+(char)#IMM1, (short)(R1&0xffff)+(char)#IMM2);  R0 = R0+(char)#IMM1; R1 = R1+(char)#IMM2; | | |
| Operand type | D0: 32-bit signed integer  #IMM1: 8-bit signed integer  #IMM2: 8-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer |  | |

LDRXY

**Load from Relative Coordinate XY with register offset**

| Usage | LDRXY D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer from memory to a register.  X and Y coordinates are specified in R0 and R1 respectively. Reads the value from memory at the coordinates obtained by adding the S0 value to X and the S1 value to Y and stores the value in D0.  The full 32 bits of R0 and S0 and the 16 lower-order bits of R1 and S1 are handled as signed integers.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine.  The value stored in D0 is an integer. | | |
| Behavior model | D0 = (int) \*(R0+S0, (short)(R1&0xffff)+(short)(S1&0xffff)); | |  |
| Operand type | D0: 32-bit signed integer  S0: 32-bit signed integer  S1: 16-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer | |  |

LDRXYU

**Load from Relative Coordinate XY with register offset, then source Updated**

| Usage | LDRXYU D0, S0, S1 | |  |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer from memory to a register.  X and Y coordinates are specified in R0 and R1 respectively. Reads the value from memory at the coordinates obtained by adding the S0 value to X and the S1 value to Y and stores the value in D0.  The full 32 bits of R0 and S0 and the 16 lower-order bits of R1 and S1 are handled as signed integers. After operation, X(R0) and Y(R1) are updated by adding S0 and S1 respectively.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine.  The value stored in D0 is an integer. | | |
| Behavior model | D0 = (int) \*(R0+S0, (short)(R1&0xffff)+(short)(S1&0xffff));  R0 = R0+S0; R1 = (short)(R1&0xffff)+(short)(S1&0xffff); | | |
| Operand type | D0: 32-bit signed integer  S0: 32-bit signed integer  S1: 16-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer |  | |

LDSRB

**LoaD Small Register Block**

| Usage | LDSRB D0, S0, S1 | |  |
| --- | --- | --- | --- |
| Function | This is an instruction to transfer 256-bit data from local working memory to a 256-bit large register block. This instruction supports only the common address space.  Writes the data read from the address specified by adding S1\*32 to address S0 of local working memory to D0.  The access to LWM must be 64-bit aligned, when unaligned, function is prohibited.  D0 is specifying the lowest register of the register block. Allowed values are 16 (for register block R16-R23), 24 (for register block R24-R31), 32 (for register block R32-R39) and 40 (for register block R40-R47) | | |
| Behavior model | D0 = \*(S0 + S1\*32); | | |
| Operand type | D0: 256-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  S1: 16-bit signed integer |  | |

LDSRBI

**LoaD Small Register Block with Immediate offset**

| Usage | LDSRBI D0, S0, #IMM | |  |
| --- | --- | --- | --- |
| Function | This is an instruction to transfer 256-bit data from local working memory to a 256-bit large register block. This instruction supports only the common address space.  Writes the data read from the address specified by adding #IMM\*32 to address S0 of local working memory to D0.  The access to LWM must be 64-bit aligned, when unaligned, function is prohibited.  D0 is specifying the lowest register of the register block. Allowed values are 16 (for register block R16-R23), 24 (for register block R24-R31), 32 (for register block R32-R39) and 40 (for register block R40-R47) | | |
| Behavior model | D0 = \*(S0 + #IMM\*32); | | |
| Operand type | D0: 256-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  #IMM: 8-bit signed integer |  | |

LDSRBC

**LoaD Small Register Block with LPC offset**

| Usage | LDSRBC D0, S0 | |  |
| --- | --- | --- | --- |
| Function | This is an instruction to transfer 256-bit data from local working memory to a 256-bit large register block. This instruction supports only the common address space.  Writes the data read from the address specified by adding LPC\*32 to address S0 of local working memory to D0.  The access to LWM must be 64-bit aligned, when unaligned, function is prohibited.  D0 is specifying the lowest register of the register block. Allowed values are 16 (for register block R16-R23), 24 (for register block R24-R31), 32 (for register block R32-R39) and 40 (for register block R40-R47) | | |
| Behavior model | D0 = \*(S0 + LPC\*32); | | |
| Operand type | D0: 256-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  LPC: 9-bit signed integer |  | |

LLROT

**Long left rotate**

| Usage | LLROT D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = (S0 << S1) | (S0 >> (64-(S1&0x3f)))  S0 is a 64-bit vector and S1 is a 32-bit vector. The instruction is left rotating the 64-bit source S0 by an unsigned value specified in the lower 6bits of source S1. D0 and S0 are 64-bit values stored in 2 consecutive 32-bit registers, which are specified by the even of 2 consecutive 32-bit registers. | | |
| Behavior model | D0 = (S0 << S1) | (S0 >> (64-S1)) | |  |

LMBD

**Left Most Bit Detection**

| Usage | LMBD D0, S0 | |  |
| --- | --- | --- | --- |
| Function | This is an instruction which searches the first "1" from left side inside S0 and gives back the position in D0. Position counting from left side. Output == 32 if "1" is not available. Implemented in SAM4. | | |
| Behavior model | S0[31] == "1"?D0=0: (S0[30] == "1"?D0=1: (S0[29] == "1"?D0=2:( .... S0[1] == "1"?D0=30: (S0[0] == "1"?D0=31:D0=32))) | | |
| Operand type | D0: 32bit unsigned integer (6bit used)  S0: 32bit unsigned integer |  | |

LPE

**LooP End**

| Usage | LPE |  | |
| --- | --- | --- | --- |
| Function | LPC = LPC - LPC\_DEC  (LPC >=0)? PC = LPC\_ADD : PC = PC+1  This is an instruction to subtract the content from the internal register LPC\_DEC from LPC and depending on the result, jump back to the address stored in the internal register LPC\_ADD (in case LPC is positive) without 1 cycle delay or continue with the instruction flow (1 cycle delay will appear). | | |
| Behavior model | LPC = LPC - LPC\_DEC  (LPC >=0)? PC = LPC\_ADD : PC = PC+1 | |  |

LPS

**LooP Start**

| Usage | LPS S0, #IMM |  | | |
| --- | --- | --- | --- | --- |
| Function | This is an instruction to fast set up a most inner loop with loop counter and loop decrement value. The value specified in S0 is written to the internal loop counter register LPC that can have a 9-bit signed integer. The loop counter decrement LPC\_DEC is initialized with the immediate value #IMM. The word address following the LPS instruction is saved into an internal register LPC\_ADD. | | | |
| Behavior model | LPC = S0;  LPC\_DEC = #IMM;  LPC\_ADD = PC+1; | | |  |
| Operand type | S0: 8-bit unsigned integer  #IMM: 5-bit unsigned integer | |

MACS

**Multiply and ACcumulate**

| Usage | MACS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = D0 + S0 \* S1  S0 and S1 are 2 32-bit signed values and D0 is a 64-bit signed element. The instruction is multiplying the 2 32-bit elements and the result is added to the 64-bit signed element D0. After, the 64-bit are stored back to the register file. D0 is a register pair specified by its even register number. | | |
| Behavior model | D0 = D0 + S0 \* S1 | |  |

MAXS

**MAXimum**

| Usage | MAXS D0, S0,S1 |  | |
| --- | --- | --- | --- |
| Function | Selects the larger value between S0 and S1, then stores the result in D0.  S0 and S1 are handled as integers in the operation. | | |
| Behavior model | D0 = ((int)S0 > (int)S1) ? S0 : S1; | |  |

MAXS8

**MAXimum Signed 8-bit value**

| Usage | MAXS8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to get the maximum value from 8 times 2 signed 8-bit input values from source S0 and S1. Implemented in SAM4. | | |
| Behavior model | D0[n] = MAX(S0[n], S1[n]), n = 0..7; | |  |
| Operand type | D0: 8 times 8-bit signed integer  S0: 8 times 8-bit signed integer  S1: 8 times 8-bit signed integer | |  |

MAXS16

**MAXimum Signed 16-bit value**

| Usage | MAXS16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to get the maximum value from 4 times 2 signed 16-bit input values from source S0 and S1. Implemented in SAM4. | | |
| Behavior model | D0[n] = MAX(S0[n], S1[n]), n = 0..3; | |  |
| Operand type | D0: 4 times 16-bit signed integer  S0: 4 times 16-bit signed integer  S1: 4 times 16-bit signed integer | |  |

MAXSU

**MAXimum compared to Uniform**

| Usage | MAXSU D0, S0,#IMM |
| --- | --- |
| Function | Selects the larger value between S0 and the uniform value indexed by #IMM, then stores the result in D0.  S0 and the uniform value are handled as integers in the operation. |
| Behavior model | D0 = ((int)S0 > (int)Uniform[#R6IMM] ? S0 : (int)Uniform[#R6IMM]; |

MAXU8

**MAXimum Unsigned 8-bit value**

| Usage | MAXU8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to get the maximum value from 8 times 2 unsigned 8-bit input values from source S0 and S1. Implemented in SAM4. | | |
| Behavior model | D0[n] = MAX(S0[n], S1[n]), n = 0..7; | |  |
| Operand type | D0: 8 times 8-bit unsigned integer  S0: 8 times 8-bit unsigned integer  S1: 8 times 8-bit unsigned integer | |  |

MAXU16

**MAXimum Unsigned 16-bit value**

| Usage | MAXU16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to get the maximum value from 4 times 2 unsigned 16-bit input values from source S0 and S1. Implemented in SAM4. | | |
| Behavior model | D0[n] = MAX(S0[n], S1[n]), n = 0..3; | |  |
| Operand type | D0: 4 times 16-bit unsigned integer  S0: 4 times 16-bit unsigned integer  S1: 4 times 16-bit unsigned integer | |  |

MINS

**MINimum**

| Usage | MINS D0, S0,S1 |  | |
| --- | --- | --- | --- |
| Function | Selects the smaller value between S0 and S1, then stores the result in D0.  S0 and S1 are handled as integers in the operation. | | |
| Behavior model | D0 = ((int)S0 > (int)S1) ? S1 : S0; | |  |

MINS8

**MINimum Signed 8-bit value**

| Usage | MINS8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to get the minimum value from 8 times 2 signed 8-bit input values from source S0 and S1. Implemented in SAM4. | | |
| Behavior model | D0[n] = MIN(S0[n], S1[n]), n = 0..7; | |  |
| Operand type | D0: 8 times 8-bit signed integer  S0: 8 times 8-bit signed integer  S1: 8 times 8-bit signed integer | |  |

MINS16

**MINimum Signed 16-bit value**

| Usage | MINS16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to get the minimum value from 4 times 2 signed 16-bit input values from source S0 and S1. Implemented in SAM4. | | |
| Behavior model | D0[n] = MIN(S0[n], S1[n]), n = 0..3; | |  |
| Operand type | D0: 4 times 16-bit signed integer  S0: 4 times 16-bit signed integer  S1: 4 times 16-bit signed integer | |  |

MINSU

**MINimum compared to Uniform**

| Usage | MINSU D0, S0,#IMM |
| --- | --- |
| Function | Selects the smaller value between S0 and the uniform value indexed by #IMM, then stores the result in D0.  S0 and the uniform value are handled as integers in the operation. |
| Behavior model | D0 = ((int)S0 > (int)Uniform[#R6IMM] ? (int)Uniform[#R6IMM] : S0; | |

MINU8

**MINimum Unsigned 8-bit value**

| Usage | MINU8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to get the minimum value from 8 times 2 unsigned 8-bit input values from source S0 and S1. Implemented in SAM4. | | |
| Behavior model | D0[n] = MIN(S0[n], S1[n]), n = 0..7; | |  |
| Operand type | D0: 8 times 8-bit unsigned integer  S0: 8 times 8-bit unsigned integer  S1: 8 times 8-bit unsigned integer | |  |

MINU16

**MINimum Unsigned 16-bit value**

| Usage | MINU16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to get the minimum value from 4 times 2 unsigned 16-bit input values from source S0 and S1. Implemented in SAM4. | | |
| Behavior model | D0[n] = MIN(S0[n], S1[n]), n = 0..3; | |  |
| Operand type | D0: 4 times 16-bit unsigned integer  S0: 4 times 16-bit unsigned integer  S1: 4 times 16-bit unsigned integer | |  |

MOV

**MOVe**

| Usage | MOV D0, S0 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer between registers. Stores the S0 value in D0. | | |
| Behavior model | D0 = S0; | |  |
| Operand type | D0: -(32-bit array)  S0: -(32-bit array) | |  |

MOVI

**MOVe Immediate value**

| Usage | MOVI D0, #IMM |  | |
| --- | --- | --- | --- |
| Function | The #IMM value is handled as a 16-bit signed integer, then sign-extended into a 32-bit signed integer, and the result is stored in D0. | | |
| Behavior model | D0 = (int)(#IMM & 0x0000ffff); | |  |
| Operand type | D0: Signed integer  #IMM: 16-bit Signed integer | |  |

MOVU

**MOVe Uniform**

| Usage | MOVU D0, #IMM |
| --- | --- |
| Function | Stores the index value in Uniform specified by #IMM in D0. |
| Behavior model | D0 = Uniform[#R6IMM]; |
| Operand type | D0: -(32-bit array)  Uniform value: -(32-bit array) |

MUL

**MULtiply**

| Usage | MUL D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Multiply the 32-bit signed input values S0 and S1 and store the lower 32-bit of the result to D0. | | |
| Behavior model | D0 = S0 \* S1; | |  |

MULS

**MULtiply with Saturation**

| Usage | MULS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to multiply 2 signed 32-bit values from source S0 and source S1 with saturation. | | |
| Behavior model | D0 = SAT(S0 \* S1, 32); | |  |
| Operand type | D0: 32-bit signed integer  S0: 32-bit signed integer  S1: 32-bit signed integer | | |

MULSS8

**MULtiply with Saturation Signed 8-bit values**

| Usage | MULSS8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to multiply 8 times 2 signed 8-bit values from source S0 and source S1 with saturation. | | |
| Behavior model | D0[n] = SAT(S0[n] \* S1[n], 8), n = 0..7; | |  |
| Operand type | D0: 8 times 8-bit signed integer  S0: 8 times 8-bit signed integer  S1: 8 times 8-bit signed integer | | |

MULSS16

**MULtiply with Saturation Signed 16-bit values**

| Usage | MULSS16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to multiply 4 times 2 signed 16-bit values from source S0 and source S1 with saturation. | | |
| Behavior model | D0[n] = SAT(S0[n] \* S1[n], 16), n = 0..3; | |  |
| Operand type | D0: 4 times 16-bit signed integer  S0: 4 times 16-bit signed integer  S1: 4 times 16-bit signed integer | | |

MULSU

**MULtiply Saturation with Uniform**

| Usage | MULSU D0, S0, #IMM |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to multiply 2 signed 32-bit values from source S0 and Uniform storage indexed by #IMM with saturation. | | |
| Behavior model | D0 = SAT(S0 \* Uniform[#R6IMM], 32); | |  |
| Operand type | D0: 32-bit signed integer  S0: 32-bit signed integer | | |

MULLS

**MULtiply long signed**

| Usage | MULLS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Multiply the 32-bit signed input values S0 and S1 and store the 64-bit result to D0. | | |
| Behavior model | D0 = S0 \* S1; | |  |

MULLS8

**MULtiply Long Signed 8-bit values**

| Usage | MULLS8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to multiply 4 times 2 signed 8-bit values from source S0 and source S1 with storage of 16-bit result. | | |
| Behavior model | D0[n] = S0[n] \* S1[n], n = 0..3; | |  |
| Operand type | D0: 4 times 16-bit signed integer  S0: 4 times 8-bit signed integer  S1: 4 times 8-bit signed integer | | | |

MULLS16

**MULtiply Long Signed 16-bit values**

| Usage | MULLS16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to multiply 2 times 2 signed 16-bit values from source S0 and source S1 with storage of 32-bit result. | | |
| Behavior model | D0[n] = S0[n] \* S1[n], n = 0..1; | |  |
| Operand type | D0: 2 times 32-bit signed integer  S0: 2 times 16-bit signed integer  S1: 2 times 16-bit signed integer | | | |

MULLSU

**MULtiply long signed by Uniform**

| Usage | MULLSU D0, S0, #IMM |
| --- | --- |
| Function | Multiply the 32-bit signed input values S0 with the 32-bit signed input value read from uniform storage indexed by #IMM and store the 64-bit result to D0. |
| Behavior model | D0 = S0 \* Uniform[#R6IMM]; |

MULLU8

**MULtiply Long Unsigned 8-bit values**

| Usage | MULLU8 D0, S0, S1 |
| --- | --- |
| Function | This is an instruction to multiply 4 times 2 unsigned 8-bit values from source S0 and source S1 with storage of 16-bit result. |
| Behavior model | D0[n] = S0[n] \* S1[n], n = 0..3; |
| Operand type | D0: 4 times 16-bit unsigned integer  S0: 4 times 8-bit unsigned integer  S1: 4 times 8-bit unsigned integer |

MULLU16

**MULtiply Long Unsigned 16-bit values**

| Usage | MULLU16 D0, S0, S1 |
| --- | --- |
| Function | This is an instruction to multiply 2 times 2 unsigned 16-bit values from source S0 and source S1 with storage of 32-bit result. |
| Behavior model | D0[n] = S0[n] \* S1[n], n = 0..1; |
| Operand type | D0: 2 times 32-bit unsigned integer  S0: 2 times 16-bit unsigned integer  S1: 2 times 16-bit unsigned integer |

MULU

**MULtiply by Uniform**

| Usage | MULU D0, S0, #IMM |
| --- | --- |
| Function | Multiply the 32-bit signed input value S0 with the 32-bit signed input value read from uniform storage indexed by #IMM and store the lower 32-bit of the result to D0. |
| Behavior model | D0 = S0 \* Uniform[#R6IMM]; |

MULUS8

**MULtiply Unsigned with Saturation 8-bit values**

| Usage | MULUS8 D0, S0, S1 |
| --- | --- |
| Function | This is an instruction to multiply 8 times 2 unsigned 8-bit values from source S0 and source S1 with saturation. |
| Behavior model | D0[n] = SAT(S0[n] \* S1[n], 8), n = 0..7; |
| Operand type | D0: 8 times 8-bit unsigned integer  S0: 8 times 8-bit unsigned integer  S1: 8 times 8-bit unsigned integer |

MULUS16

**MULtiply Unsigned with Saturation 16-bit values**

| Usage | MULUS16 D0, S0, S1 |
| --- | --- |
| Function | This is an instruction to multiply 4 times 2 unsigned 16-bit values from source S0 and source S1 with saturation. |
| Behavior model | D0[n] = SAT(S0[n] \* S1[n], 16), n = 0..3; |
| Operand type | D0: 4 times 16-bit unsigned integer  S0: 4 times 16-bit unsigned integer  S1: 4 times 16-bit unsigned integer |

NCSCS

**Non-symmetric Census Signature Calculation Step**

| Usage | NCSCS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | S0 is compared against S1 and -S1. Each comparison is adding a 1 or 0 depending whether the result is true (==1) or false (==0) to the earlier by 2 bits to the left shifted D0. The result is again stored to D0. | | |
| Behavior model | D0 = (D0 << 2) + (S0 > S1) + (S0 > -S1) | |  |

NOP

**No OPeration**

| Usage | NOP |  | |
| --- | --- | --- | --- |
| Function | No operation is executed. One cycle of bubble is inserted in the thread operation pipeline. | | |
| Behavior model | ; // empty statement | |  |

OR

**OR**

| Usage | OR D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Treats S0 and S1 as 32-bit arrays, takes the logical OR of S0 and S1, and stores the  result in D0. | | |
| Behavior model | D0 = S0 | S1; | |  |

OR64

**OR 64-bit values**

| Usage | OR64 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to OR 2 64-bit (8 times 8-bit or 4 times 16-bit) values. | | |
| Behavior model | D0[n] = S0[n] OR S1[n], n = 0..63; | |  |
| Operand type | D0: 64-bit integer  S0: 64-bit integer  S1: 64-bit integer | | |

ORU

**OR with Uniform**

| Usage | ORU D0, S0, #IMM |
| --- | --- |
| Function | Treats S0 and the uniform value indexed by #IMM as 32-bit arrays, takes the logical OR of S0 and the uniform value, and stores the result in D0. |
| Behavior model | D0 = S0 | Uniform[#R6IMM]; |

PACKB0

**PACK byte 0**

| Usage | PACKB0 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = (S0[39:32] << 24) | (S0[7:0] << 16) | (S1[39:32] << 8) | (S1[7:0])  S0 and S1 are 64-bit vectors with each 2 32-bit elements, which are specified by the even of 2 consecutive 32-bit registers. The instruction is taking from each of the four 32-bit elements the lowest 8-bit part and packing them together in D0. D0 is handled as a 32-bit signed value. | | |
| Behavior model | D0 = (S0[39:32] << 24) | (S0[7:0] << 16) | (S1[39:32] << 8) | (S1[7:0]) | |  |

PACKB1

**PACK byte 1**

| Usage | PACKB1 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = (S0[47:40] << 24) | (S0[15:8] << 16) | (S1[47:40] << 8) | (S1[15:8])  S0 and S1 are 64-bit vectors with each 2 32-bit elements, which are specified by the even of 2 consecutive 32-bit registers. The instruction is taking from each of the four 32-bit elements the second lowest 8-bit part and packing them together in D0. D0 is handled as a 32-bit signed value. | | |
| Behavior model | D0 = (S0[47:40] << 24) | (S0[15:8] << 16) | (S1[47:40] << 8) | (S1[15:8]) | |  |

PACKB2

**PACK byte 2**

| Usage | PACKB2 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = (S0[55:48] << 24) | (S0[23:16] << 16) | (S1[55:48] << 8) | (S1[23:16])  S0 and S1 are 64-bit vectors with each 2 32-bit elements, which are specified by the even of 2 consecutive 32-bit registers. The instruction is taking from each of the four 32-bit elements the second highest 8-bit part and packing them together in D0. D0 is handled as a 32-bit signed value. | | |
| Behavior model | D0 = (S0[55:48] << 24) | (S0[23:16] << 16) | (S1[55:48] << 8) | (S1[23:16]) | |  |

PACKB3

**PACK byte 3**

| Usage | PACKB3 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = (S0[63:56] << 24) | (S0[31:24] << 16) | (S1[63:56] << 8) | (S1[31:24])  S0 and S1 are 64-bit vectors with each 2 32-bit elements, which are specified by the even of 2 consecutive 32-bit registers. The instruction is taking from each of the four 32-bit elements the highest 8-bit part and packing them together in D0. D0 is handled as a 32-bit signed value. | | |
| Behavior model | D0 = (S0[63:56] << 24) | (S0[31:24] << 16) | (S1[63:56] << 8) | (S1[31:24]) | |  |

PACKH0

**PACK half word 0**

| Usage | PACKH0 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = (S0[15:0] << 16) | (S1[15:0])  S0 and S1 are 32-bit vectors with each 1 32-bit element. The instruction is taking from each 32-bit element the lowest 16-bit part and packing them together in D0. D0 is handled as a 32-bit signed value. | | |
| Behavior model | D0 = (S0[15:0] << 16) | (S1[15:0]) | |  |

PACKH1

**PACK half word 1**

| Usage | PACKH1 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | D0 = (S0[31:16] << 16) | (S1[31:16])  S0 and S1 are 32-bit vectors with each 1 32-bit element. The instruction is taking from each 32-bit element the highest 16-bit part and packing them together in D0. D0 is handled as a 32-bit signed value. | | |
| Behavior model | D0 = (S0[31:16] << 16) | (S1[31:16]) | |  |

PCMPAGT

**Parallel CoMPare Anded Greater Than**

| Usage | PCMPAGT D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This instruction compares (greater) in parallel the upper and lower halves of S0 and S1 and writes the ANDed comparison result to D0, in case D0 is R3, SR is set with this bit. Mapped on each Thread. | | |
| Behavior model | SR.F = (S0[31:0] > S1[31:0]) && (S0[63:32] > S1[63:32]); (D0 == R3)  D0 = (S0[31:0] > S1[31:0]) && (S0[63:32] > S1[63:32]); (D0 != R3) | |  |
| Operand type | D0: 1bit integer  S0: 2 32-bit signed integer  S1: 2 32-bit signed integer | |  |

PCMPALT

**Parallel CoMPare Anded Less Than**

| Usage | PCMPALT D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This instruction compares (smaller) in parallel the upper and lower halves of S0 and S1 and writes the ANDed comparison result to D0, in case D0 is R3, SR is set with this bit. Mapped on each Thread. | | |
| Behavior model | SR.F = (S0[31:0] < S1[31:0]) && (S0[63:32] < S1[63:32]) ; (D0 == R3)  D0 = (S0[31:0] < S1[31:0]) && (S0[63:32] < S1[63:32]) ; (D0 != R3) | |  |
| Operand type | D0: 1bit integer  S0: 2 32-bit signed integer  S1: 2 32-bit signed integer | |  |

PCMPAEQ

**Parallel CoMPare Anded EQual**

| Usage | PCMPAEQ D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This instruction compares (equal) in parallel the upper and lower halves of S0 and S1 and writes the ANDed comparison result to D0, in case D0 is R3, SR is set with this bit. Mapped on each Thread. | | |
| Behavior model | SR.F = (S0[31:0] == S1[31:0]) && (S0[63:32] == S1[63:32]) ; (D0 == R3)  D0 = (S0[31:0] == S1[31:0]) && (S0[63:32] == S1[63:32]) ; (D0 != R3) | |  |
| Operand type | D0: 1bit integer  S0: 2 32-bit signed integer  S1: 2 32-bit signed integer | |  |

PCMPOGT

**Parallel CoMPare Ored Greater Than**

| Usage | PCMPOGT D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This instruction compares (greater) in parallel the upper and lower halves of S0 and S1 and writes the ORed comparison result to D0, in case D0 is R3, SR is set with this bit. Mapped on each Thread. | | |
| Behavior model | SR.F = (S0[31:0] > S1[31:0]) || (S0[63:32] > S1[63:32]) ; (D0 == R3)  D0 = (S0[31:0] > S1[31:0]) || (S0[63:32] > S1[63:32]) ; (D0 != R3) | |  |
| Operand type | D0: 1bit integer  S0: 2 32-bit signed integer  S1: 2 32-bit signed integer | |  |

PCMPOLT

**Parallel CoMPare Ored Less Than**

| Usage | PCMPOLT D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This instruction compares (smaller) in parallel the upper and lower halves of S0 and S1 and writes the ORed comparsion result to D0, in case D0 is R3, SR is set with this bit. Mapped on each Thread. | | |
| Behavior model | SR.F = (S0[31:0] < S1[31:0]) || (S0[63:32] < S1[63:32]) ; (D0 == R3)  D0 = (S0[31:0] < S1[31:0]) || (S0[63:32] < S1[63:32]) ; (D0 != R3) | |  |
| Operand type | D0: 1bit integer  S0: 2 32-bit signed integer  S1: 2 32-bit signed integer | |  |

PCMPOEQ

**Parallel CoMPare Ored EQual**

| Usage | PCMPOEQ D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This instruction compares (equal) in parallel the upper and lower halves of S0 and S1 and writes the ORed comparison result to D0, in case D0 is R3, SR is set with this bit. Mapped on each Thread. | | |
| Behavior model | SR.F = (S0[31:0] == S1[31:0]) || (S0[63:32] == S1[63:32]); (D0 == R3)  D0 = (S0[31:0] == S1[31:0]) || (S0[63:32] == S1[63:32]); (D0 != R3) | |  |
| Operand type | D0: 1bit integer  S0: 2 32-bit signed integer  S1: 2 32-bit signed integer | |  |

PLSB8

**Pack the LSB's of 4 8-bit values into the 4 lowest bits of dst**

| Usage | PLSB8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to take the LSB's of 4 8-bit values in SRC0 and concat them to the 4-bit left shifted value of SRC1 and store the result to D0. Implemented in each thread. | | |
| Behavior model | D0[31:4] = S1[27:0]; D0[0] = S0[0]; D0[1] = S0[8]; D0[2] = S0[16]; D0[3] = S0[24]; | |  |
| Operand type | D0: 32-bit unsigned integer  S0: 32-bit unsigned integer  S1: 32-bit unsigned integer | |  |

PLSB16

**Pack the LSB's of 2 16-bit values into the 2 lowest bits of dst**

| Usage | PLSB16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to take the LSB's of 2 16-bit values in SRC0 and concat them to the 2-bit left shifted value of SRC1 and store the result to D0. Implemented in each thread. | | |
| Behavior model | D0[31:2] = S1[29:0]; D0[0] = S0[0]; D0[1] = S0[16]; | |  |
| Operand type | D0: 32-bit unsigned integer  S0: 32-bit unsigned integer  S1: 32-bit unsigned integer | |  |

RETA

**RETurn from subroutine Alternative**

| Usage | RETA D0 |  | |
| --- | --- | --- | --- |
| Function | Returns to the PC stored in the temporary-storage PC register specified by D0.  The register to be specified by D0 is not the general-purpose register number, but the CALLA/RETA dedicated temporary-storage PC register number.  There are eight registers for this purpose and 0 to 7 can be specified.  When a number not written by the CALLA instruction is specified, operation is undefined.  The 3 lower-order bits of D0 are handled as an unsigned integer. | | |
| Behavior model | PC=PCR[(D0&0x7)] | |  |
| Operand type | D0: 3-bit unsigned integer | |  |

RMBD

**Right Most Bit Detection**

| Usage | RMBD D0, S0 | |  |
| --- | --- | --- | --- |
| Function | This is an instruction which searches the first "1" from right side inside S0 and gives back the position in D0. Position counting from right side. Output == 32 if "1" is not available. Implemented in SAM4. | | |
| Behavior model | S0[0] == "1"?D0=0: (S0[1] == "1"?D0=1: (S0[2] == "1"?D0=2: (.... S0[30] == "1"?D0=30: (S0[31] == "1"?D0=31:D0=32))))) | | |
| Operand type | D0: 32bit unsigned integer (6bit used)  S0: 32bit unsigned integer |  | |

RXCH

**Register eXCHange**

| Usage | RXCH D0, S0 |  | |
| --- | --- | --- | --- |
| Function | This instruction exchanges the content of 2 registers of a register pair. Mapped on each Thread. | | |
| Behavior model | D0[31:0] = S0[63:32]; D0[63:32] = S0[31:0]; | |  |
| Operand type | D0: 2 32-bit signed integer  S0: 2 32-bit signed integer | |  |

SBOCRD

**SBO of CooRDinate**

| Usage | SBOCRD D0, #IMM1, #IMM2 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction for outputting the X coordinate, Y coordinate, and D0 (data) to memory via the SBO port.  The X and Y coordinates to be output are X and Y coordinates which are specified in R0 and R1 respectively, with the #IMM1 value added to X and the #IMM2 value added to Y.  The 16 lower-order bits of R0 and R1 are handled as integers. #IMM1 and #IMM2 need to be specified with 8-bit signed integers.  The output mask can be specified by control register CR10. | | |
| Behavior model | SBO((ushort((ushort)(R0&0xffff)+(char)#IMM1), (ushort)((ushort)(R1&0xffff)+#IMM2), D0); | | |
| Operand type | D0: -(32-bit array)  #IMM1: 8-bit signed integer  #IMM2: 8-bit signed integer  R0: 16-bit signed integer  R1: 16-bit signed integer | |  |

SCSCS

**Symmetric Census Signature Calculation Step**

| Usage | SCSCS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | S0 is compared against S1 and INV(S1) as signed values. Each comparison is adding a 1 or 0 depending whether the result is true (==1) or false (==0) to the earlier by 2 bits to the left shifted D0. The result is again stored to D0. | | |
| Behavior model | D0 = (D0 << 2) + (S0 > S1) + (S0 > INV(S1)) | |  |

SETHI

**Set HIgher bits**

| Usage | SETHI D0, #ASMIMM |  | |
| --- | --- | --- | --- |
| Function | Higher 16 bits of #ASMIMM replaces the 16 higher-order bits of D0, and the result is retained by D0. | | |
| Behavior model | #IMM = (#ASMIMM & 0xffff0000)>>16  D0 = (D0 & 0x0000ffff) | #IMM<<16; | |  |
| Operand type | D0: -(32-bit array)  #ASMIMM: -(32-bit array) | |  |
|  | #IMM: -(16-bit array) | |  |

SHA8

**SHift Arithmetic left of signed 8-bit values**

| Usage | SHA8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to arithmetic shift 8 signed 8-bit values by 8 signed 4-bit values stored in S1 as 8 signed 8-bit values. Implemented in SAM4. | | |
| Behavior model | D0[n] = S0[n] <<< S1[n], n = 0..7; (S1[n]>0)  D0[n] = S0[n] >>> S1[n], n = 0..7; (S1[n]<0)  D0[n] = S0[n], n = 0..7; (S1[n]==0) | |  |
| perand type | D0: 8 times 8-bit signed integer  S0: 8 times 8-bit signed integer  S1: 8 times 8-bit signed integer, only lower 4 bits are used | |  |

SHA16

**SHift Arithmetic of signed 16-bit values**

| Usage | SHA16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to arithmetic shift 4 signed 16-bit values by 4 signed 5-bit values stored in S1 as 4 signed 16-bit values. Implemented in SAM4. | | |
| Behavior model | D0[n] = S0[n] <<< S1[n], n = 0..3; (S1[n]>0)  D0[n] = S0[n] >>> S1[n], n = 0..3; (S1[n]<0)  D0[n] = S0[n], n = 0..3; (S1[n]==0) | |  |
| Operand type | D0: 4 times 16-bit signed integer  S0: 4 times 16-bit signed integer  S1: 4 times 16-bit signed integer, only lower 5 bits are used | |  |

SHAI8

**SHift Arithmetic by signed Immediate value signed 8-bit values**

| Usage | SHAI8 D0, S0, #IMM |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to shift 8 signed 8-bit values by 1 signed 4-bit value stored as 1 signed 8-bit immediate value. Implemented in SAM4. | | |
| Behavior model | D0[n] = S0[n] <<< #IMM, n = 0..7; (#IMM>0)  D0[n] = S0[n] >>> #IMM, n = 0..7; (#IMM<0)  D0[n] = S0[n], n = 0..7; (#IMM==0) | |  |
| Operand type | D0: 8 times 8-bit signed integer  S0: 8 times 8-bit signed integer  #IMM: 1 8-bit signed integer, only lower 4 bits are used | |  |

SHAI16

**SHift Arithmetic by signed Immediate value signed 16-bit values**

| Usage | SHAI16 D0, S0, #IMM |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to shift 4 signed 16-bit values by 1 signed 5-bit value stored as 1 signed 8-bit immediate value. Implemented in SAM4. | | |
| Behavior model | D0[n] = S0[n] <<< #IMM, n = 0..3; (#IMM>0)  D0[n] = S0[n] >>> #IMM, n = 0..3; (#IMM<0)  D0[n] = S0[n] , n = 0..3; (#IMM==0) | |  |
| Operand type | D0: 4 times 16-bit signed integer  S0: 4 times 16-bit signed integer  #IMM: 1 8-bit signed integer, only lower 5 bits are used | |  |

SHL

**SHift Left**

| Usage | SHL D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Shifts S0 to the left by the number of bits given by the value of S1, and stores the result in D0. The lower 5bits of S1 are used to decide shift size. | | |
| Behavior model | D0 = S0 << (S1 & 0x1f); | |  |

SHL8

**SHift Logical of unsigned 8-bit values**

| Usage | SHL8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to shift 8 unsigned 8-bit values by 8 signed 4-bit values stored in S1 as 8 signed 8-bit values. Implemented in SAM4. | | |
| Behavior model | D0[n] = S0[n] << S1[n] , n = 0..7; (S1[n]>0)  D0[n] = S0[n] >> S1[n] , n = 0..7; (S1[n]<0)  D0[n] = S0[n] , n = 0..7; (S1[n]==0) | |  |
| Operand type | D0: 8 times 8-bit unsigned integer  S0: 8 times 8-bit unsigned integer  S1: 8 times 8-bit signed integer, only lower 4 bits are used | |  |

SHL16

**SHift Logical of unsigned 16-bit values**

| Usage | SHL16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to shift 4 unsigned 16-bit values by 4 signed 5-bit values stored in S1 as 4 signed 16-bit values. Implemented in SAM4. | | |
| Behavior model | D0[n] = S0[n] << S1[n] , n = 0..3; (S1[n]>0)  D0[n] = S0[n] >> S1[n] , n = 0..3; (S1[n]<0)  D0[n] = S0[n] , n = 0..3; (S1[n]==0) | |  |
| Operand type | D0: 4 times 16-bit unsigned integer  S0: 4 times 16-bit unsigned integer  S1: 4 times 16-bit signed integer, only lower 5 bits are used | |  |

SHLI

**SHift Left with Immediate value**

| Usage | SHLI D0, S0, #IMM |  | |
| --- | --- | --- | --- |
| Function | Shifts S0 to the left by the number of bits given by the value of #IMM, and stores the result in D0. The lower 5bits of #IMM are used to decide shift size. | | |
| Behavior model | D0 = S0 << (#IMM & 0x1f); | |  |

SHLI8

**SHift Logical by signed Immediate value unsigned 8-bit values**

| Usage | SHLI8 D0, S0, #IMM |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to shift 8 unsigned 8-bit values by 1 signed 4-bit value stored as 1 signed 8-bit immediate value (sign bit expanded). Implemented in SAM4. | | |
| Behavior model | D0[n] = S0[n] << #IMM, n = 0..7; (#IMM>0)  D0[n] = S0[n] >> #IMM, n = 0..7; (#IMM<0)  D0[n] = S0[n], n = 0..7; (#IMM==0) | |  |
| Operand type | D0: 8 times 8-bit unsigned integer  S0: 8 times 8-bit unsigned integer  #IMM: 1 8-bit signed integer, only lower 4 bits are used | |  |

SHLI16

**SHift Logical by signed Immediate value unsigned 16-bit values**

| Usage | SHLI16 D0, S0, #IMM |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to shift 4 unsigned 16-bit values by 1 signed 5-bit value stored as 1 signed 8-bit immediate value (sign bit expanded). Implemented in SAM4. | | |
| Behavior model | D0[n] = S0[n] << #IMM, n = 0..3; (#IMM>0)  D0[n] = S0[n] >> #IMM, n = 0..3; (#IMM<0)  D0[n] = S0[n] , n = 0..3; (#IMM==0) | |  |
| Operand type | D0: 4 times 16-bit unsigned integer  S0: 4 times 16-bit unsigned integer  #IMM: 1 8-bit signed integer, only lower 5 bits are used | |  |

SHRA

**SHift Right Arithmetically**

| Usage | SHRA D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Treats S0 as a 32-bit signed integer, arithmetically shifts S0 to the right by the number of bits given by the value of S1, and stores the result in D0. The lower 5bits of S1 are used to decide sift size. | | |
| Behavior model | D0 = (signed int)S0 >> (S1 & 0x1f); | |  |

SHRAI

**SHift Right Arithmetically with Immediate value**

| Usage | SHRAI D0, S0, #IMM |  | |
| --- | --- | --- | --- |
| Function | Treats S0 as a 32-bit signed integer, arithmetically shifts S0 to the right by the number of bits given by the value of #IMM, and stores the result in D0. The lower 5bits of #IMM are used to decide shift size. | | |
| Behavior model | D0 = (signed int)S0 >> (#IMM & 0x1f); | |  |

SHRL

**SHift Right Logically**

| Usage | SHRL D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Treats S0 as a 32-bit unsigned integer, logically shifts S0 to the right by the number of bits given by the value of S1, and stores the result in D0. The lower 5bits of S1 are used to decide sift size. | | |
| Behavior model | D0 = (unsigned int)S0 >> (S1 & 0x1f); | |  |

SHRLI

**SHift Right Logically with Immediate value**

| Usage | SHRLI D0, S0, #IMM |  | |
| --- | --- | --- | --- |
| Function | Treats S0 as a 32-bit unsigned integer, logically shifts S0 to the right by the number of bits given by the value of #IMM, and stores the result in D0. The lower 5bits of #IMM are used to decide shift size. | | |
| Behavior model | D0 = (unsigned int)S0 >> (#IMM & 0x1f); | |  |

SHUF

**SHUFfle**

| Usage | SHUF D0, S0, S1 | |  |
| --- | --- | --- | --- |
| Function | This instruction is a shuffle instruction which is shuffling the data in S0 based on the given parameters in S1 and overwrites dependent on the parameter setting in S1 the whole or parts of the destination register. Implemented in SAM4. | | |
| Behavior model | if (S1[n\*8+3] == 1) then D0[(n+1)\*8-1:n\*8] = D0[(n+1)\*8-1:n\*8]  else if (S1[n\*8+4] == 1) then D0[(n+1)\*8-1:n\*8] = 8'b0  else if (S1[n\*8+5] == 1) then D0[(n+1)\*8-1:n\*8] = S0[S1[n\*8+2:n\*8]\*8 : (S1[n\*8+2:n\*8]+1)\*8-1]  else D0[(n+1)\*8-1:n\*8] = S0[ (S1[n\*8+2:n\*8]+1)\*8-1 : S1[n\*8+2:n\*8]\*8] | | |
| Operand type | D0: 64bit unsigned integer  S0: 64bit unsigned integer  S1: 64bit unsigned integer |  | |

ST

**Store word**

| Usage | ST D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | It is a data transfer instruction from register to memory.  The 32-bit data from register D0 is stored to the 32-bit aligned address specified by the sum of the values S0 and S1\*4. S1 is specifying a 32-bit word offset. | | |
| Behavior model | \*(S0+S1\*4) = D0; | |  |

STB

**Store byte**

| Usage | STB D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | It is a data transfer instruction from register to memory.  The lower 8-bit data from register D0 is stored to the 8-bit aligned address specified by the sum of the values S0 and S1. S1 is specifying a byte offset. | | |
| Behavior model | \*(S0+S1) = D0; | |  |

STBI

**Store byte with immediate offset**

| Usage | STBI D0, S0, #IMM |  | |
| --- | --- | --- | --- |
| Function | It is a data transfer instruction from register to memory.  The lower 8-bit data from register D0 is stored to the 8-bit aligned address specified by the sum of the values S0 and #IMM. The 8-bit signed immediate value is specifying a byte offset. | | |
| Behavior model | \*(S0+#IMM) = D0; | |  |

STC

**STore to Control Register**

| Usage | STC D0, S0 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction for storing data in a control register.  Writes the lower 16 bits of the data in the general purpose register S0 value to the control register D0. Specifying D0 as a control register that does not exist is prohibited. | | |
| Behavior model | ControlRegisterD0 = S0&0xffff; | |  |
| Operand type | D0: Control Register.  S0: 16-bit array | | |

STCI

**STore Immediate to Control Register**

| Usage | STCI D0 , #IMM |  |
| --- | --- | --- |
| Function | This is an instruction for storing data in a control register.  Writes the #IMM value to the control register D0. Specifying D0 as a control register that does not exist is prohibited. | |
| Behavior model | ControlRegisterD0 = #IMM; | |
| Operand type | D0: Control Register  #IMM: 16-bit array | |

STH

**Store half word**

| Usage | STH D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | It is a data transfer instruction from register to memory.  The lower 16-bit data from register D0 is stored to the 16-bit aligned address specified by the sum of the values S0 and S1\*2. S1 is specifying a 16-bit short offset. | | |
| Behavior model | \*(S0+S1\*2) = D0; | |  |

STHI

**Store half word with immediate offset**

| Usage | STHI D0, S0, #ASMIMM |  | |
| --- | --- | --- | --- |
| Function | It is a data transfer instruction from register to memory.  The lower 16-bit data from register D0 is stored to the 16-bit aligned address specified by the sum of the values S0 and #ASMIMM. The 8-bit signed immediate value is specifying a byte offset. | | |
| Behavior model | #IMM = #ASMIMM>>1  \*(S0+#IMM\*2) = D0; | |  |

STI

**Store word with immediate offset**

| Usage | STI D0, S0, #ASMIMM |  | |
| --- | --- | --- | --- |
| Function | It is a data transfer instruction from register to memory.  The 32-bit data from register D0 is stored to the 32-bit aligned address specified by the sum of the values S0 and #ASMIMM. The 8-bit signed immediate value is specifying a byte offset. | | |
| Behavior model | #IMM = #ASMIMM>>2  \*(S0+#IMM\*4) = D0; | |  |

STLL

**STore Long from Local working memory**

| Usage | STLL D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This instruction is available only for V3M Ver.2.0, V3H and V3U.  This is an instruction to transfer 64-bit data from a 64-bit pair register to local working memory. This instruction supports the common and non-common address space.  Writes the D0 data to the address specified by adding the S1×8 value to address S0 of local working memory.  When this instruction is used for other memory accesses, this instruction transfer 32-bit from the specified even register to the specified memory address.  The 64-bit access to LWM must be aligned, when unaligned, function is prohibited.  D0 is a 64-bit pair register, always the lower even register is specified, which holds the lower 32-bit. S0 and S1 are 32-bit registers. | | |
| Behavior model | \*(S0 + S1\*8) = D0; | |  |
| Operand type | D0: 64-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  S1: 32-bit signed integer | |  |

STLLI

**STore Long from Local working memory with Immediate offset**

| Usage | STLLI D0, S0, #ASMIMM |  | |
| --- | --- | --- | --- |
| Function | This instruction is available only for V3M Ver.2.0, V3H and V3U.  This is an instruction to transfer 64-bit data from local working memory to a 64-bit pair register. This instruction supports the common and non-common address space.  Writes the D0 data to the address specified by adding the #IMM×8 value to address S0 of local working memory.  When this instruction is used for other memory accesses, this instruction transfer 32-bit from the specified even register to the specified memory address.  The 64-bit access to LWM must be aligned, when unaligned, function is prohibited.  D0 is a 64-bit pair register, always the lower even register is specified, which holds the lower 32-bit. S0 is a 32-bit register. | | |
| Behavior model | #IMM = #ASMIMM>>3  \*(S0 + (char)#IMM\*8) = D0; | |  |
| Operand type | D0: 64-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  #IMM: 8-bit signed integer | |  |

STLRB

**STore Large Register Block**

| Usage | STLRB D0, S0, S1 |  |
| --- | --- | --- |
| Function | This is an instruction to transfer 512-bit data from 512-bit large register block to local working memory. This instruction supports only the common address space.  Writes the data read from D0 to the address specified by adding S1\*64 to address S0 of local working memory.  The access to LWM must be 64-bit aligned, when unaligned, function is prohibited.  D0 is specifying the lowest register of the register block. Allowed values are 16 (for register block R16-R31) and 32 (for register block R32-R47) | |
| Behavior model | \*(S0 + S1\*64) = D0; | |  |
| Operand type | D0: 512-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  S1: 16-bit signed integer | |  |

**STore Large Register Block with Immediate offset**

STLRBI

| Usage | STLRBI D0, S0, #IMM |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to transfer 512-bit data from 512-bit large register block to local working memory. This instruction supports only the common address space.  Writes the data read from D0 to the address specified by adding #IMM\*64 to address S0 of local working memory.  The access to LWM must be 64-bit aligned, when unaligned, function is prohibited.  D0 is specifying the lowest register of the register block. Allowed values are 16 (for register block R16-R31) and 32 (for register block R32-R47) | | |
| Behavior model | \*(S0 +#IMM\*64) = D0; | |  |
| Operand type | D0: 512-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  #IMM: 8-bit signed integer | |  |

STLRBC

**STore Large Register Block with LPC offset**

| Usage | STLRBC D0, S0 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to transfer 512-bit data from 512-bit large register block to local working memory. This instruction supports only the common address space.  Writes the data read from D0 to the address specified by adding LPC\*64 to address S0 of local working memory.  The access to LWM must be 64-bit aligned, when unaligned, function is prohibited.  D0 is specifying the lowest register of the register block. Allowed values are 16 (for register block R16-R31) and 32 (for register block R32-R47) | | |
| Behavior model | \*(S0 + LPC\*64) = D0; | |  |
| Operand type | D0: 512-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  LPC: 9-bit signed integer | |  |

STRCI

**STore to Relative Coordinate with Immediate offset**

| Usage | STRCI D0, #IMM1, #IMM2 | |  |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer from a register to memory.  Stores the D0 value in memory at X and Y coordinates which are specified in R0 and R1 respectively, with the #IMM1 value added to X and the #IMM2 value added to Y.  The full 32 bits of R0 and the 16 lower-order bits of R1 are handled as signed integers. #IMM1 and #IMM2 need to be specified with 8-bit signed integers.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine.  D0 is handled as an integer. | | |
| Behavior model | @(R0+(char)#IMM1, (short)(R1&0xffff)+(char)#IMM2) = D0; | | |
| Operand type | D0: 32-bit signed integer  #IMM1: 8-bit signed integer  #IMM2: 8-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer |  | |

STRCIU

**STore to Relative Coordinate with Immediate offset, then source Updated**

| Usage | STRCIU D0, #IMM1, #IMM2 | |  |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer from a register to memory.  Stores the D0 value in memory at X and Y coordinates which are specified in R0 and R1 respectively, with the #IMM1 value added to X and the #IMM2 value added to Y.  The full 32 bits of R0 and the 16 lower-order bits of R1 are handled as signed integers. #IMM1 and #IMM2 need to be specified with 8-bit signed integers. After operation, X(R0) and Y(R1) are updated by adding #IMM1 and #IMM2 respectively.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine.  D0 is handled as an integer. | | |
| Behavior model | @(R0+(char)#IMM1, (short)(R1&0xffff)+(char)#IMM2) = D0;  R0 = R0+(char)#IMM1; R1 = R1+(char)#IMM2; | | |
| Operand type | D0: 32-bit signed integer  #IMM1: 8-bit signed integer  #IMM2: 8-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer |  | |

STRXY

**STore to Relative Coordinate XY with register offset**

| Usage | STRXY D0, S0, S1 | |  |
| --- | --- | --- | --- |
| Function | This is an instruction for data transfer from a register to memory.  Stores the D0 value in memory at X and Y coordinates which are specified in R0 and R1 respectively, with the S0 value added to X and the S1 value added to Y.  The full 32 bits of R0 and S0 and the 16 lower-order bits of R1 and S1 are handled as signed integers.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine. | | |
| Behavior model | @(R0+S0, (short)(R1&0xffff)+(short)(S1&0xffff)) = D0; | | |
| Operand type | D0: 32-bit signed integer  S0: 32-bit signed integer  S1: 16-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer |  | |

STRXYU

**STore to Relative Coordinate XY with register offset, then source Updated**

| Usage | STRXYU D0, S0, S1 |  |
| --- | --- | --- |
| Function | This is an instruction for data transfer from a register to memory.  Stores the D0 value in memory at X and Y coordinates which are specified in R0 and R1 respectively, with the S0 value added to X and the S1 value added to Y.  The full 32 bits of R0 and S0 and the 16 lower-order bits of R1 and S1 are handled as signed integers. After operation, X(R0) and Y(R1) are updated by adding S0 and S1 respectively.  Operation cannot be guaranteed if the calculated offset is negative to the base address or obtained address is outside from the range handled by the CVengine. | |
| Behavior model | @(R0+S0, (short)(R1&0xffff)+(short)(S1&0xffff)) = D0;  R0 = R0+S0; R1 = (short)(R1&0xffff)+(short)(S1&0xffff); | |
| Operand type | D0: 32-bit signed integer  S0: 32-bit signed integer  S1: 16-bit signed integer  R0: 32-bit signed integer  R1: 16-bit signed integer | |

STSRB

**STore Small Register Block**

| Usage | STSRB D0, S0, S1 |  |
| --- | --- | --- |
| Function | This is an instruction to transfer 256-bit data from 256-bit large register block to local working memory. This instruction supports only the common address space.  Writes the data read from D0 to the address specified by adding S1\*32 to address S0 of local working memory.  The access to LWM must be 64-bit aligned, when unaligned, function is prohibited.  D0 is specifying the lowest register of the register block. Allowed values are 16 (for register block R16-R23), 24 (for register block R24-R31), 32 (for register block R32-R39) and 40 (for register block R40-R47) | |
| Behavior model | \*(S0 + S1\*32) = D0; | |
| Operand type | D0: 256-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  S1: 16-bit signed integer | |

STSRBI

**STore Small Register Block with Immediate offset**

| Usage | STSRBI D0, S0, #IMM |  |
| --- | --- | --- |
| Function | This is an instruction to transfer 256-bit data from 256-bit large register block to local working memory. This instruction supports only the common address space.  Writes the data read from D0 to the address specified by adding #IMM\*32 to address S0 of local working memory.  The access to LWM must be 64-bit aligned, when unaligned, function is prohibited.  D0 is specifying the lowest register of the register block. Allowed values are 16 (for register block R16-R23), 24 (for register block R24-R31), 32 (for register block R32-R39) and 40 (for register block R40-R47) | |
| Behavior model | \*(S0 +#IMM\*32) = D0; | |
| Operand type | D0: 256-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  #IMM: 8-bit signed integer | |

**STore Small Register Block with LPC offset**

STSRBC

| Usage | STSRBC D0, S0 |  |
| --- | --- | --- |
| Function | This is an instruction to transfer 256-bit data from 256-bit large register block to local working memory. This instruction supports only the common address space.  Writes the data read from D0 to the address specified by adding LPC\*32 to address S0 of local working memory.  The access to LWM must be 64-bit aligned, when unaligned, function is prohibited.  D0 is specifying the lowest register of the register block. Allowed values are 16 (for register block R16-R23), 24 (for register block R24-R31), 32 (for register block R32-R39) and 40 (for register block R40-R47) | |
| Behavior model | \*(S0 + LPC\*32) = D0; | |
| Operand type | D0: 256-bit unsigned integer  S0: 32-bit unsigned integer (Multiples of 8)  LPC: 9-bit signed integer | |

SUB

**SUB**

| Usage | SUB D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | Subtracts S1 from S0, then stores the result in D0.  S0 and S1 are handled as integers in the operation. | | |
| Behavior model | D0 = S0 - S1; | |  |

SUBS

**SUBtract with Saturation**

| Usage | SUBS D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to subtract a signed 32-bit source value S1 from a signed 32-bit source value S0 with saturation. Mapped on Thread. | | |
| Behavior model | D0 = SAT(S0 - S1, 32); | |  |
| Operand type | D0: 32-bit signed integer  S0: 32-bit signed integer  S1: 32-bit signed integer | | | |

SUBSS8

**SUBtract Signed with Saturation 8-bit values**

| Usage | SUBSS8 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to subtract 8 times signed 8-bit value source S1 from signed 8-bit value source S0 with saturation. | | |
| Behavior model | D0[n] = SAT(S0[n] - S1[n], 8), n = 0..7; | |  |
| Operand type | D0: 8 times 8-bit signed integer  S0: 8 times 8-bit signed integer  S1: 8 times 8-bit signed integer | | |

SUBSS16

**SUBtract Signed with Saturation 16-bit values**

| Usage | SUBSS16 D0, S0, S1 |  | |
| --- | --- | --- | --- |
| Function | This is an instruction to subtract 4 times signed 16-bit value source S1 from signed 16-bit value source S0 with saturation. | | |
| Behavior model | D0[n] = SAT(S0[n] - S1[n], 16), n = 0..3; | |  |
| Operand type | D0: 4 times 16-bit signed integer  S0: 4 times 16-bit signed integer  S1: 4 times 16-bit signed integer | | |

SUBRU

**SUB Uniform from Register**

| Usage | SUBRU D0, S0, #IMM |
| --- | --- |
| Function | Subtracts the uniform value indexed by #IMM from S0, then stores the result in D0.  S0 and the uniform value are handled as integers in the operation. |
| Behavior model | D0 = S0 - Uniform[#R6IMM]; |

SUBSRU

**SUBtract with Saturation from Register value a Unified memory value**

| Usage | SUBSRU D0, S0, #IMM |
| --- | --- |
| Function | This is an instruction to subtract a signed 32-bit value specified by an Uniform index #IMM from a signed 32-bit source value S0 with saturation. Mapped on Thread. |
| Behavior model | D0 = SAT(S0 - Uniform[#R6IMM], 32); |
| Operand type | D0: 32-bit signed integer  S0: 32-bit signed integer |

SUBSUR

**SUBtract with Saturation from Unified memory value a Register value**

| Usage | SUBSUR D0, S0, #IMM |
| --- | --- |
| Function | This is an instruction to subtract a signed 32-bit source value S0 from a signed 32-bit value specified by an Uniform index #IMM with saturation. Mapped on Thread. |
| Behavior model | D0 = SAT(Uniform[#R6IMM] - S0, 32); |
| Operand type | D0: 32-bit signed integer  S0: 32-bit signed integer |

SUBUR

**SUB Register from Uniform**

| Usage | SUBUR D0, S0, #IMM |
| --- | --- |
| Function | Subtracts S0 from the uniform value indexed by #IMM, then stores the result in D0.  S0 and the uniform value are handled as integers in the operation. |
| Behavior model | D0 = Uniform[#R6IMM] - S0; |

SUBUS8

**SUBtract Unsigned with Saturation 8-bit values**

| Usage | SUBUS8 D0, S0, S1 |
| --- | --- |
| Function | This is an instruction to subtract 8 times unsigned 8-bit value source S1 from unsigned 8-bit value source S0 with saturation. |
| Behavior model | D0[n] = SAT(S0[n] - S1[n], 8), n = 0..7; |
| Operand type | D0: 8 times 8-bit unsigned integer  S0: 8 times 8-bit unsigned integer  S1: 8 times 8-bit unsigned integer |

SUBUS16

**SUBtract Unsigned with Saturation 16-bit values**

| Usage | SUBUS16 D0, S0, S1 |
| --- | --- |
| Function | This is an instruction to subtract 4 times unsigned 16-bit value source S1 from unsigned 16-bit value source S0 with saturation. |
| Behavior model | D0[n] = SAT(S0[n] - S1[n], 16), n = 0..3; |
| Operand type | D0: 4 times 16-bit unsigned integer  S0: 4 times 16-bit unsigned integer  S1: 4 times 16-bit unsigned integer |

SYNCG

**SYNC Group**

| Usage | SYNCG |  | |
| --- | --- | --- | --- |
| Function | Syncronize all threads inside a master-slave group. A SYNCG inside a thread causes the qarbiter to set the bit for this thread inside the SYNCG register and the thread is waiting till a continue is received from qarbiter, After all bits from a master-slave group are set in the SYNCG register, the qarbiter is sending the continue signal and the bits are cleared inside the SYNCG register. A master thread can issue this instruction. | | |
| Behavior model | - | |  |

SYNCP

**SYNC PBUF**

| Usage | SYNCP #IMM |  | |
| --- | --- | --- | --- |
| Function | Clear or flush pbuf. The behavior of the instruction is configured as follows.  1: Clear read buffer  2: Flush write buffer  3: Clear read buffer and flush write buffer.  Other immediate value is prohibited.  The “pbuf” is an alias of 0th data cache associated with each thread. | | |
| Behavior model | - | |  |

TRAP

**TRAP program**

| Usage | TRAP (no operand) |  | |
| --- | --- | --- | --- |
| Function | Stops the program and transits the thread to the idle state. | | |
| Behavior model | - | |  |

UPLSB8

**Unpack the 4 lowest bits of S0 into the LSB's of 4 8-bit values stored in D0**

| Usage | UPLSB8 D0, S0 |
| --- | --- |
| Function | This is an instruction to unpack the 4 lowest bits of S0 into the LSB's of 4 8-bit values stored in D0. The values of other bits are retained. Implemented in each thread. |
| Behavior model | D0[0] = S0[0]; D0[8] = S0[1]; D0[16] = S0[2]; D0[24] = S0[3]; |
| Operand type | D0: 32bit unsigned integer  S0: 32bit unsigned integer (4bit used) |

UPLSB16

**Unpack the 2 lowest bits of S0 into the LSB's of 2 16-bit values stored in D0**

| Usage | UPLSB16 D0, S0 |
| --- | --- |
| Function | This is an instruction to unpack the 2 lowest bits of S0 into the LSB's of 2 16-bit values stored in D0. The values of other bits are retained. Implemented in each thread. |
| Behavior model | D0[0] = S0[0]; D0[16] = S0[1]; |
| Operand type | D0: 32bit unsigned integer  S0: 32bit unsigned integer (2bit used) |

VPEBS

**Variable Position Extract Byte Signed extension**

| Usage | VPEBS D0, S0, #IMM | |  |
| --- | --- | --- | --- |
| Function | The byte value stored at a position defined by #IMM in S0 gets sign extended to 32-bit and stored into D0. #IMM can take values 0 till 24. | | |
| Behavior model | D0 = ((S0 >> #IMM) << 24) >> 24 |  | |

VPEBU

**Variable Position Extract Byte Unsigned extension**

| Usage | VPEBU D0, S0, #IMM | |  |
| --- | --- | --- | --- |
| Function | The byte value stored at a position defined by #IMM in S0 gets unsigned extended to 32-bit and stored into D0. #IMM can take values 0 till 24. | | |
| Behavior model | D0 = ((S0 >> #IMM) << 24) >>> 24 |  | |

VPEHS

**Variable Position Extract Half word Signed extension**

| Usage | VPEHS D0, S0, #IMM | |  |
| --- | --- | --- | --- |
| Function | The half word value stored at a position defined by #IMM in S0 gets sign extended to 32-bit and stored into D0. #IMM can take values 0 till 16. | | |
| Behavior model | D0 = ((S0 >> #IMM) << 16) >> 16 |  | |

VPEHU

**Variable Position Extract Half word Unsigned extension**

| Usage | VPEHU D0, S0, #IMM | |  |
| --- | --- | --- | --- |
| Function | The half word value stored at a position defined by #IMM in S0 gets unsigned extended to 32-bit and stored into D0. #IMM can take values 0 till 16. | | |
| Behavior model | D0 = ((S0 >> #IMM) << 16) >>> 16 |  | |

WAITDMA

**Wait for DMA**

| Usage | WAITDMA |  | |
| --- | --- | --- | --- |
| Function | Wait till qarbiter sends continue signal which happens after the DMA transfer is finished. | | |
| Behavior model | - | |  |

# Instruction Format, Code and Pipeline Operation

An instruction has a fixed length of 32 bits and are classified into five instruction types of as follows: NI, TypeI16, TypeI8(xy), TypeI8(i), TypeRU and TypeR. The bit format of each instruction type is described in section 10.1.

All instructions have the F1 bit and F0 bit to conditionally execute the instruction depending on the status register value. If the F0 and F1 bits are both 0, then the instruction is always executed regardless of the status register value. If the F1 bit is 1, then the instruction is executed when the status register is 1, whereas if the F1 bit is 0, the instruction is regarded as NOP. If the F0 bit is 0, then the instruction is executed when the status register is 0, whereas if the F0 bit is 1, the instruction is regarded as NOP. If the F1 and F0 bits are both 1, operation is not guaranteed.

The opcode of each instruction is listed in section 10.2. The instruction group to which the instruction belongs is also described in section 10.2.

The pipeline operation of each instruction group is described in section 10.3. Since the pipeline is stalled depending on the combination of the instructions, the conditions described in section 10.3 must be considered during programming.

## Instruction Format

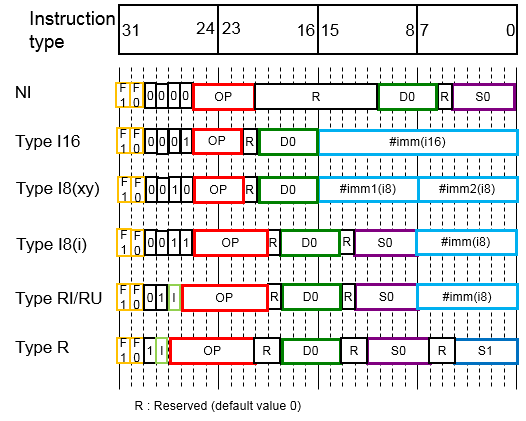


Figure 10‑1 List of Instruction Formats (V3M)

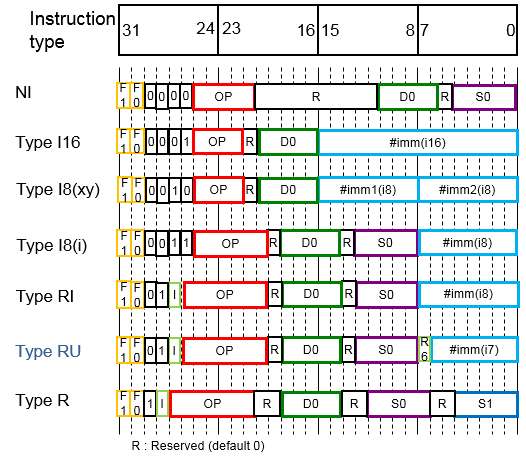


Figure 10‑2 List of Instruction Formats (V3M Ver.2.0, V3H)

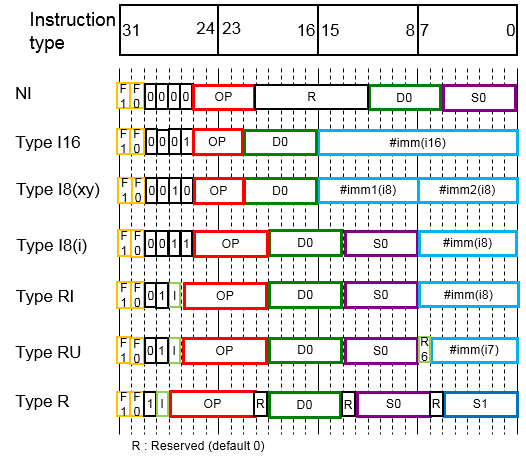


Figure 10‑3 List of Instruction Formats (V3U)

One instruction has a fixed length of 32 bits. Each instruction belongs to one of the six types of instructions: TypeNI, TypeI16, TypeI8(xy), TypeI8(i), TypeRI, TypeRU, and TypeR.

NI No Immediate. Instruction format with two register operands.

I16 Instruction format with one register operand and one immediate source value of 16-bit signed integer data.

I8(xy) Instruction format with one register operand and two immediate source values of 8-bit signed integer data.

I8(i) Instruction format with two register operands and one immediate source value of 8-bit integer data.

RI Instruction format with two register operands and one immediate source value of 8-bit integer data. Whether an immediate value is signed or unsigned differs with the instruction. The I bit exists to classify integer operation (I=1) and floating point operation (I=0).

RU Instruction format with two register operands and one immediate source value of 8-bit integer data. Whether an immediate value is signed or unsigned differs with the instruction. The I bit exists to classify integer operation (I=1) and floating point operation (I=0).

For V3M Ver.2.0, V3H and V3U, instruction format with two register operands and one immediate source value of 7-bit integer data The I bit exists to classify integer operation (I=1) and floating point operation (I=0). The R6 bit is used to distinguish if R6 is referred to access the uniform storage. When the R6 bit is 1, R6 can be used as a general purpose register. To control this bit, refer to Section 9 Note 4.

R Instruction format with three register operands. The I bit exists to classify integer operation (I=1) and floating point operation (I=0).

Execution or non-execution of the instruction can be selected depending on the value of the status register (SR) by setting either the F0 bit or F1 bit of the instruction to 1. This enables the instruction to be conditionally executed without using a branch instruction.

The instruction is always executed when the F0 and F1 bits are both set to 0. Setting both the F0 bit and F1 bit to 1 is prohibited.

Table 10‑1 Execution/Non-Execution Based on the Status Register and F0/F1 Flag State

|  | {F1, F0} == {0, 0} | {F1, F0} == {0, 1} | {F1, F0} == {1, 0} | {F1, F0} == {1, 1} |
| --- | --- | --- | --- | --- |
| SR==0 | Always executed\*1 | Executed\*1 | Not executed\*2 | Setting prohibited\*2 |
| SR==1 | Not executed\*2 | Executed\*1 |

\*1: Illegal instruction error can take place when the command is corrupted. \*2: No illegal instruction error takes place in spite of other bits corrupted.

## Instruction Code List

Table 10‑2 Instruction Code List

| Instruction Type | Mnemonic | Instruction Group | Number of Execution Steps | Number of Issuance Cycles | Code | I Bit | SAM |
| --- | --- | --- | --- | --- | --- | --- | --- |
| NI | NOP | CO | - (Control instruction) | 1 | 0x00 | - | - |
| CALLR | BR | - (Branch instruction) | 1 | 0x02 | - | - |
| ACTST | ET1 | 1 | 1 | 0x04 | - | - |
| BRAR | BR | - (Branch instruction) | 1 | 0x05 | - | - |
| TRAP | CO | - (Control instruction) | 1 | 0x06 | - | - |
| RETA | BR | - (Branch instruction) | 1 | 0x07 | - | - |
| INT | CO | - (Control instruction) | 1 | 0x08 | - | - |
| LMBD | ES2 | 2 | 1 | 0x09 | 1 | SAM4 |
| RMBD | ES2 | 2 | 1 | 0x0A | 1 | SAM4 |
| SYNCG | CO | - (Control instruction) | 1 | 0x0B | - | - |
| UPLSB8 | ET1 | 1 | 1 | 0x0C | 1 | - |
| WAITDMA | CO | - (Control instruction) | 1 | 0x0D | - | - |
| UPLSB16 | ET1 | 1 | 1 | 0x0E | 1 | - |
| RXCH | ET1 | 1 | 1 | 0x0F | 1 | - |
| INV64 | ES2 | 2 | 1 | 0x10 | 1 | - |
| LPE | BR | 1 | 1 | 0x11 | - | - |
| LDLRBC | LS | 4 | 1 | 0x12 | - | - |
| STLRBC | LS | 1 | 1 | 0x13 | - | - |
| LDSRBC | LS | 4 | 1 | 0x14 | - | - |
| STSRBC | LS | 1 | 1 | 0x15 | - | - |
| ABS8 | ES2 | 2 | 1 | 0x16 | 1 | SAM4 |
| ABS16 | ES2 | 2 | 1 | 0x17 | 1 | SAM4 |
| I16 | MOVI | ET1 | 1 | 1 | 0x00 | - | - |
| SETHI | ET1 | 1 | 1 | 0x01 | - | - |
| CALL | BR | - (Branch instruction) | 1 | 0x02 | - | - |
| CALLA | BR | - (Branch instruction) | 1 | 0x03 | - | - |
| STCI | CO | - | 1 | 0x04 | - | - |
| BRA | BR | - (Branch instruction) | 1 | 0x05 | - | - |
| LDCI | CO | - (Control instruction) | 1 | 0x06 | - | - |
| FLDCI | CO | - (Control instruction) | 1 | 0x07 | - | - |
| I8(XY) | LDRCI | LS | 4 | 1 | 0x00 | - | - |
| FLDRCI | LS | 4 | 1 | 0x01 | - | - |
| STRCI | LS | 1 | 1 | 0x02 | - | - |
| FSTRCI | LS | 1 | 1 | 0x03 | - | - |
| LDRCIU | LS | 4 | 1 | 0x04 | - | - |
| FLDRCIU | LS | 4 | 1 | 0x05 | - | - |
| STRCIU | LS | 1 | 1 | 0x06 | - | - |
| FSTRCIU | LS | 1 | 1 | 0x07 | - | - |
| SBOCRD | LS | 1 | 1 | 0x08 | - | - |
| I8(i) | LDBUI | LS | 4 | 1 | 0x00 | - | - |
| LDBSI | LS | 4 | 1 | 0x01 | - | - |
| STBI | LS | 1 | 1 | 0x02 | - | - |
| LDHUI | LS | 4 | 1 | 0x04 | - | - |
| LDHSI | LS | 4 | 1 | 0x05 | - | - |
| STHI | LS | 1 | 1 | 0x06 | - | - |
| LDI | LS | 4 | 1 | 0x08 | - | - |
| STI | LS | 1 | 1 | 0x0A | - | - |
| VPEBS | ET1 | 1 | 1 | 0x0C | - | - |
| VPEBU | ET1 | 1 | 1 | 0x0D | - | - |
| VPEHS | ET1 | 1 | 1 | 0x0E | - | - |
| VPEHU | ET1 | 1 | 1 | 0x0F | - | - |
| SYNCP | CO | - (Control instruction) | 1 | 0x10 | - | - |
| LDLLI | LS | 4 | 1 | 0x14 | - | - |
| STLLI | LS | 1 | 1 | 0x15 | - | - |
| LPS | BR | 1 | 1 | 0x16 | - | - |
| DMAWAITS | CO | - (Control instruction) | 1 | 0x17 | - | - |
| RI | ADDI | ET1 | 1 | 1 | 0x20 | 1 | - |
| SHLI | ET1 | 1 | 1 | 0x21 | 1 | - |
| SHRAI | ET1 | 1 | 1 | 0x22 | 1 | - |
| SHRLI | ET1 | 1 | 1 | 0x23 | 1 | - |
| BRCID | BR | - (Branch instruction) | 1 | 0x24 | 1 | - |
| BRTID | BR | - (Branch instruction) | 1 | 0x25 | 1 | - |
| LDLRBI | LS | 4 | 1 | 0x28 | - | - |
| STLRBI | LS | 1 | 1 | 0x29 | - | - |
| LDSRBI | LS | 4 | 1 | 0x2A | - | - |
| STSRBI | LS | 1 | 1 | 0x2B | - | - |
| SHLI8 | ES2 | 2 | 1 | 0x2C | 1 | SAM4 |
| SHLI16 | ES2 | 2 | 1 | 0x2D | 1 | SAM4 |
| SHAI8 | ES2 | 2 | 1 | 0x2E | 1 | SAM4 |
| SHAI16 | ES2 | 2 | 1 | 0x2F | 1 | SAM4 |
| ADDSI | ET1 | 1 | 1 | 0x30 | 1 | - |
| BTSTI | ET1 | 1 | 1 | 0x33 | 1 | - |
| RU | ADDU | ET1 | 1 | 1 | 0x02 | 1 | - |
| ADDSU | ET1 | 1 | 1 | 0x03 | 1 | - |
| SUBRU | ET1 | 1 | 1 | 0x04 | 1 | - |
| SUBSRU | ET1 | 1 | 1 | 0x05 | 1 | - |
| SUBUR | ET1 | 1 | 1 | 0x06 | 1 | - |
| SUBSUR | ET1 | 1 | 1 | 0x07 | 1 | - |
| MULU | ES3 | 3 | 1 | 0x08 | 1 | SAM8 |
| MULSU | ES3 | 3 | 1 | 0x09 | 1 | SAM8 |
| EORU | ET1 | 1 | 1 | 0x0A | 1 | - |
| ORU | ET1 | 1 | 1 | 0x0B | 1 | - |
| ANDU | ET1 | 1 | 1 | 0x0C | 1 | - |
| MAXSU | ET1 | 1 | 1 | 0x0E | 1 | - |
| MINSU | ET1 | 1 | 1 | 0x10 | 1 | - |
| CMPU.SGT | ET1 | 1 | 1 | 0x12 | 1 | - |
| CMPU.SLE | ET1 | 1 | 1 | 0x14 | 1 | - |
| CMPU.SGE | ET1 | 1 | 1 | 0x16 | 1 | - |
| CMPU.SLT | ET1 | 1 | 1 | 0x18 | 1 | - |
| CMPU.EQ | ET1 | 1 | 1 | 0x1A | 1 | - |
| CMPU.NEQ | ET1 | 1 | 1 | 0x1B | 1 | - |
| MULLSU | ES3 | 3 | 1 | 0x1C | 1 | SAM8 |
| MOVU | ET1 | 1 | 1 | 0x1F | 1 | - |
| BTSTU | ET1 | 1 | 1 | 0x32 | 1 | - |
| ASMU | ES3 | 3 | 1 | 0x52 | 1 | SAM4 |
| ADDDBLU | ET1 | 1 | 1 | 0x53 | 1 |  |
| FADDU | ESF3 | 3 | 1 | 0x02 | 0 | SAM4/ - \* |
| FSUBU | ESF3 | 3 | 1 | 0x04 | 0 | SAM4/ - \* |
| FMULU | ESF3 | 3 | 1 | 0x08 | 0 | SAM4/ - \* |
| FMAXU | ET1 | 1 | 1 | 0x0E | 0 | - |
| FMINU | ET1 | 1 | 1 | 0x10 | 0 | - |
| FCMPU.OGT | ET1 | 1 | 1 | 0x12 | 0 | - |
| FCMPU.UGT | ET1 | 1 | 1 | 0x13 | 0 | - |
| FCMPU.OLE | ET1 | 1 | 1 | 0x14 | 0 | - |
| FCMPU.ULE | ET1 | 1 | 1 | 0x15 | 0 | - |
| FCMPU.OEQ | ET1 | 1 | 1 | 0x16 | 0 | - |
| FCMPU.UEQ | ET1 | 1 | 1 | 0x17 | 0 | - |
| FCMPU.OGE | ET1 | 1 | 1 | 0x18 | 0 | - |
| FCMPU.UGE | ET1 | 1 | 1 | 0x19 | 0 | - |
| FCMPU.OLT | ET1 | 1 | 1 | 0x1A | 0 | - |
| FCMPU.ULT | ET1 | 1 | 1 | 0x1B | 0 | - |
| FCMPU.ONE | ET1 | 1 | 1 | 0x1C | 0 | - |
| FCMPU.UNE | ET1 | 1 | 1 | 0x1D | 0 | - |
| FCMPU.O | ET1 | 1 | 1 | 0x1E | 0 | - |
| FCMPU.UO | ET1 | 1 | 1 | 0x1F | 0 | - |
| FMADU | ESF4 | 4 | 1 | 0x20 | 0 | SAM4/ - \* |
| FMSUU | ESF4 | 4 | 1 | 0x21 | 0 | SAM4/ - \* |
| R | LDRXY | LS | 4 | 1 | 0x00 | 1 | - |
| STRXY | LS | 1 | 1 | 0x01 | 1 | - |
| INV | ET1 | 1 | 1 | 0x02 | 1 | - |
| ABS | ET1 | 1 | 1 | 0x03 | 1 | - |
| ADD | ET1 | 1 | 1 | 0x04 | 1 | - |
| ADDS | ET1 | 1 | 1 | 0x05 | 1 | - |
| SUB | ET1 | 1 | 1 | 0x06 | 1 | - |
| SUBS | ET1 | 1 | 1 | 0x07 | 1 | - |
| MUL | ES3 | 3 | 1 | 0x08 | 1 | SAM8 |
| MULS | ES3 | 3 | 1 | 0x09 | 1 | SAM8 |
| EOR | ET1 | 1 | 1 | 0x0A | 1 | - |
| OR | ET1 | 1 | 1 | 0x0B | 1 | - |
| AND | ET1 | 1 | 1 | 0x0C | 1 | - |
| AND64 | ES2 | 2 | 1 | 0x0D | 1 | - |
| I2F | ET1 | 1 | 1 | 0x0E | 1 | - |
| SHUF | ES1 | 1 | 1 | 0x0F | 1 | - |
| MAXS | ET1 | 1 | 1 | 0x10 | 1 | - |
| GMADBUL | ES3 | 3 | 1 | 0x11 | 1 | SAM2 |
| MINS | ET1 | 1 | 1 | 0x12 | 1 | - |
| GMADBSL | ES3 | 3 | 1 | 0x13 | 1 | SAM2 |
| SCSCS | ES2 | 2 | 1 | 0x14 | 1 | SAM8 |
| NCSCS | ES2 | 2 | 1 | 0x15 | 1 | SAM8 |
| STC | CO | - (Control instruction) | 1 | 0x16 | 1 | - |
| LLROT | ES3 | 3 | 1 | 0x17 | 1 | SAM4 |
| LDLRB | LS | 4 | 1 | 0x18 | - | - |
| STLRB | LS | 1 | 1 | 0x19 | - | - |
| CNT0 | ET1 | 1 | 1 | 0x1A | 1 | - |
| CNT1 | ET1 | 1 | 1 | 0x1B | 1 | - |
| MOV | ET1 | 1 | 1 | 0x1C | 1 | - |
| SHL | ET1 | 1 | 1 | 0x1D | 1 | - |
| SHRA | ET1 | 1 | 1 | 0x1E | 1 | - |
| SHRL | ET1 | 1 | 1 | 0x1F | 1 | - |
| L2F | ES3 | 3 | 1 | 0x20 | 1 | SAM8 |
| BTST | ET1 | 1 | 1 | 0x21 | 1 | - |
| PACKH0 | ES2 | 2 | 1 | 0x22 | 1 | SAM8 |
| PACKH1 | ES2 | 2 | 1 | 0x23 | 1 | SAM8 |
| PACKB0 | ES2 | 2 | 1 | 0x24 | 1 | SAM8 |
| PACKB1 | ES2 | 2 | 1 | 0x25 | 1 | SAM8 |
| PACKB2 | ES2 | 2 | 1 | 0x26 | 1 | SAM8 |
| PACKB3 | ES2 | 2 | 1 | 0x27 | 1 | SAM8 |
| GMADBU | ES3 | 3 | 1 | 0x28 | 1 | SAM4/SAM2\* |
| GMADBS | ES3 | 3 | 1 | 0x29 | 1 | SAM4/SAM2\* |
| GMADHU | ES3 | 3 | 1 | 0x2A | 1 | SAM4/SAM2\* |
| GMADHS | ES3 | 3 | 1 | 0x2B | 1 | SAM4/SAM2\* |
| GADDBU | ES3 | 3 | 1 | 0x2C | 1 | SAM4 |
| GADDBS | ES3 | 3 | 1 | 0x2D | 1 | SAM4 |
| GADDHU | ES3 | 3 | 1 | 0x2E | 1 | SAM4 |
| GADDHS | ES3 | 3 | 1 | 0x2F | 1 | SAM4 |
| GSUBBU | ES3 | 3 | 1 | 0x30 | 1 | SAM4 |
| GSUBBS | ES3 | 3 | 1 | 0x31 | 1 | SAM4 |
| GSUBHU | ES3 | 3 | 1 | 0x32 | 1 | SAM4 |
| GSUBHS | ES3 | 3 | 1 | 0x33 | 1 | SAM4 |
| GSADBU | ES3 | 3 | 1 | 0x34 | 1 | SAM4 |
| GSADBS | ES3 | 3 | 1 | 0x35 | 1 | SAM4 |
| GSADHU | ES3 | 3 | 1 | 0x36 | 1 | SAM4 |
| GSADHS | ES3 | 3 | 1 | 0x37 | 1 | SAM4 |
| CSUSU | ES2 | 2 | 1 | 0x38 | 1 | SAM8 |
| CSUSS | ES2 | 2 | 1 | 0x39 | 1 | SAM8 |
| MACS | ES3 | 3 | 1 | 0x3A | 1 | SAM8 |
| GMADHUL | ES3 | 3 | 1 | 0x3B | 1 | SAM2 |
| MULLS | ES3 | 3 | 1 | 0x3C | 1 | SAM8 |
| GMADHSL | ES3 | 3 | 1 | 0x3D | 1 | SAM2 |
| LDBU | LS | 4 | 1 | 0x3E | 1 | - |
| LDBS | LS | 4 | 1 | 0x3F | 1 | - |
| LDHU | LS | 4 | 1 | 0x40 | 1 | - |
| LDHS | LS | 4 | 1 | 0x41 | 1 | - |
| LD | LS | 4 | 1 | 0x42 | 1 | - |
| ST | LS | 1 | 1 | 0x43 | 1 | - |
| STB | LS | 1 | 1 | 0x44 | 1 | - |
| STH | LS | 1 | 1 | 0x45 | 1 | - |
| CMP.SGT | ET1 | 1 | 1 | 0x46 | 1 | - |
| PLSB8 | ET1 | 1 | 1 | 0x47 | 1 | - |
| CMP.SLE | ET1 | 1 | 1 | 0x48 | 1 | - |
| PLSB16 | ET1 | 1 | 1 | 0x49 | 1 | - |
| CMP.SGE | ET1 | 1 | 1 | 0x4A | 1 | - |
| CMV8 | ET1 | 1 | 1 | 0x4B | 1 | - |
| CMP.SLT | ET1 | 1 | 1 | 0x4C | 1 | - |
| CMV16 | ET1 | 1 | 1 | 0x4D | 1 | - |
| CMP.EQ | ET1 | 1 | 1 | 0x4E | 1 | - |
| CMP.NEQ | ET1 | 1 | 1 | 0x4F | 1 | - |
| LDRXYU | LS | 4 | 1 | 0x50 | 1 | - |
| STRXYU | LS | 1 | 1 | 0x51 | 1 | - |
| ASM | ES3 | 3 | 1 | 0x52 | 1 | SAM4 |
| ADDDBL | ET1 | 1 | 1 | 0x53 | 1 | - |
| LDLL | LS | 4 | 1 | 0x54 | 1 | - |
| STLL | LS | 1 | 1 | 0x55 | 1 | - |
| PCMPAEQ | ET1 | 1 | 1 | 0x56 | 1 | - |
| PCMPOEQ | ET1 | 1 | 1 | 0x57 | 1 | - |
| LDSRB | LS | 4 | 1 | 0x58 | - | - |
| STSRB | LS | 1 | 1 | 0x59 | - | - |
| PCMPAGT | ET1 | 1 | 1 | 0x5A | 1 | - |
| PCMPOGT | ET1 | 1 | 1 | 0x5B | 1 | - |
| PCMPALT | ET1 | 1 | 1 | 0x5C | 1 | - |
| PCMPOLT | ET1 | 1 | 1 | 0x5D | 1 | - |
| ADDSS8 | ES2 | 2 | 1 | 0x5E | 1 | SAM4 |
| ADDSS16 | ES2 | 2 | 1 | 0x5F | 1 | SAM4 |
| ADDUS8 | ES2 | 2 | 1 | 0x60 | 1 | SAM4 |
| ADDUS16 | ES2 | 2 | 1 | 0x61 | 1 | SAM4 |
| SUBSS8 | ES2 | 2 | 1 | 0x62 | 1 | SAM4 |
| SUBSS16 | ES2 | 2 | 1 | 0x63 | 1 | SAM4 |
| SUBUS8 | ES2 | 2 | 1 | 0x64 | 1 | SAM4 |
| SUBUS16 | ES2 | 2 | 1 | 0x65 | 1 | SAM4 |
| MULSS8 | ES3 | 3 | 1 | 0x66 | 1 | SAM4 |
| MULSS16 | ES3 | 3 | 1 | 0x67 | 1 | SAM4 |
| MULUS8 | ES3 | 3 | 1 | 0x68 | 1 | SAM4 |
| MULUS16 | ES3 | 3 | 1 | 0x69 | 1 | SAM4 |
| MULLS8 | ES3 | 3 | 1 | 0x6A | 1 | SAM4 |
| MULLS16 | ES3 | 3 | 1 | 0x6B | 1 | SAM4 |
| MULLU8 | ES3 | 3 | 1 | 0x6C | 1 | SAM4 |
| MULLU16 | ES3 | 3 | 1 | 0x6D | 1 | SAM4 |
| SHL8 | ES2 | 2 | 1 | 0x6E | 1 | SAM4 |
| SHL16 | ES2 | 2 | 1 | 0x6F | 1 | SAM4 |
| SHA8 | ES2 | 2 | 1 | 0x70 | 1 | SAM4 |
| SHA16 | ES2 | 2 | 1 | 0x71 | 1 | SAM4 |
| CMP.SLE8 | ES2 | 2 | 1 | 0x72 | 1 | SAM4 |
| CMP.SLE16 | ES2 | 2 | 1 | 0x73 | 1 | SAM4 |
| CMP.ULE8 | ES2 | 2 | 1 | 0x74 | 1 | SAM4 |
| CMP.ULE16 | ES2 | 2 | 1 | 0x75 | 1 | SAM4 |
| CMP.SLT8 | ES2 | 2 | 1 | 0x76 | 1 | SAM4 |
| CMP.SLT16 | ES2 | 2 | 1 | 0x77 | 1 | SAM4 |
| CMP.ULT8 | ES2 | 2 | 1 | 0x78 | 1 | SAM4 |
| CMP.ULT16 | ES2 | 2 | 1 | 0x79 | 1 | SAM4 |
| CMP.EQ8 | ES2 | 2 | 1 | 0x7A | 1 | SAM4 |
| CMP.EQ16 | ES2 | 2 | 1 | 0x7B | 1 | SAM4 |
| CMP.NEQ8 | ES2 | 2 | 1 | 0x7C | 1 | SAM4 |
| CMP.NEQ16 | ES2 | 2 | 1 | 0x7D | 1 | SAM4 |
| OR64 | ES2 | 2 | 1 | 0x7E | 1 | - |
| EOR64 | ES2 | 2 | 1 | 0x7F | 1 | - |
| FLDRXY | LS | 4 | 1 | 0x00 | 0 | - |
| FSTRXY | LS | 1 | 1 | 0x01 | 0 | - |
| FABS | ET1 | 1 | 1 | 0x03 | 0 | - |
| FADD | ESF3 | 3 | 1 | 0x04 | 0 | SAM4/ - \* |
| FSUB | ESF3 | 3 | 1 | 0x06 | 0 | SAM4/ - \* |
| FMUL | ESF3 | 3 | 1 | 0x08 | 0 | SAM4/ - \* |
| F2I | ET1 | 1 | 1 | 0x0E | 0 | - |
| FMAX | ET1 | 1 | 1 | 0x10 | 0 | - |
| FMIN | ET1 | 1 | 1 | 0x12 | 0 | - |
| FSTC | CO | - (Control instruction) | 1 | 0x14 | 0 | - |
| FFRC | ESF4 | 4 | 1 | 0x1A | 0 | SAM4 |
| FFLR | ESF3 | 3 | 1 | 0x1B | 0 | SAM4 |
| FMAD | ESF4 | 4 | 1 | 0x1C | 0 | SAM4/ - \* |
| FMSU | ESF4 | 4 | 1 | 0x1D | 0 | SAM4/ - \* |
| FMP | ESF4 | 4 | 1 | 0x1E | 0 | SAM4 |
| FCMP.OGT | ET1 | 1 | 1 | 0x20 | 0 | - |
| FCMP.UGT | ET1 | 1 | 1 | 0x21 | 0 | - |
| FCMP.OLE | ET1 | 1 | 1 | 0x22 | 0 | - |
| FCMP.ULE | ET1 | 1 | 1 | 0x23 | 0 | - |
| FCMP.OGE | ET1 | 1 | 1 | 0x24 | 0 | - |
| FCMP.UGE | ET1 | 1 | 1 | 0x25 | 0 | - |
| FCMP.OLT | ET1 | 1 | 1 | 0x26 | 0 | - |
| FCMP.ULT | ET1 | 1 | 1 | 0x27 | 0 | - |
| FCMP.OEQ | ET1 | 1 | 1 | 0x28 | 0 | - |
| FCMP.UEQ | ET1 | 1 | 1 | 0x29 | 0 | - |
| FCMP.ONE | ET1 | 1 | 1 | 0x2A | 0 | - |
| FCMP.UNE | ET1 | 1 | 1 | 0x2B | 0 | - |
| FCMP.O | ET1 | 1 | 1 | 0x2C | 0 | - |
| FCMP.UO | ET1 | 1 | 1 | 0x2D | 0 | - |
| FLDRXYU | LS | 4 | 1 | 0x50 | 0 | - |
| FSTRXYU | LS | 1 | 1 | 0x51 | 0 | - |
| MAXS8 | ES2 | 2 | 1 | 0x60 | 0 | SAM4 |
| MAXS16 | ES2 | 2 | 1 | 0x61 | 0 | SAM4 |
| MAXU8 | ES2 | 2 | 1 | 0x62 | 0 | SAM4 |
| MAXU16 | ES2 | 2 | 1 | 0x63 | 0 | SAM4 |
| MINS8 | ES2 | 2 | 1 | 0x64 | 0 | SAM4 |
| MINS16 | ES2 | 2 | 1 | 0x65 | 0 | SAM4 |
| MINU8 | ES2 | 2 | 1 | 0x66 | 0 | SAM4 |
| MINU16 | ES2 | 2 | 1 | 0x67 | 0 | SAM4 |

\* The latter is only for V3M Ver.2.0, V3H and V3U.

## Pipeline Operation of CVengine Thread

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| IF | ID | EX1 | EX2 | EX3 | EX4 | WB |

Figure 10‑4 Pipeline Configuration of CVengine Thread

The CVengine thread is a microprocessor that issues single instructions to be processed in a pipeline. The pipeline is shown in the above picture. The pipeline has seven stages: instruction fetching (IF), decoding and register reading (ID), 1st execution (EX1), 2nd execution (EX2), 3rd execution (EX3), 4th execution (EX4), and write-back (WB). A single instruction is executed as a combination of pipeline stages.

### Pipeline operations for each instruction group

The pipeline operations for each instruction group are shown below.

##### ET1 group

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| IF | ID | EX1 | EX2 | EX3 | EX4 | WB |
| - Instruction fetch | - Instruction decode  - Forwarding  - Register read | - Operation |  |  |  | - Write-back |

##### ES2 group

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| IF | ID | EX1 | EX2 | EX3 | EX4 | WB |
| - Instruction fetch | - Instruction decode  - Forwarding  - Register read | - Operation | - Operation |  |  | - Write-back |

##### ES3 group

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| IF | ID | EX1 | EX2 | EX3 | EX4 | WB |
| - Instruction fetch | - Instruction decode  - Forwarding  - Register read | - Operation | - Operation | - Operation |  | - Write-back |

##### ESF3 group

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| IF | ID | EX1 | EX2 | EX3 | EX4 | WB |
| - Instruction fetch | - Instruction decode  - Forwarding  - Register read | - Operation | - Operation | - Operation |  | - Write-back |

##### ESF4 group

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| IF | ID | EX1 | EX2 | EX3 | EX4 | WB |
| - Instruction fetch | - Instruction decode  - Forwarding  - Register read | - Operation | - Operation | - Operation | - Operation | - Write-back |

##### LS group

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| IF | ID | EX1 | EX2 | EX3 | EX4 | WB |
| - Instruction fetch | - Instruction decode  - Forwarding  - Register read | - Memory data access | - Memory data access | - Memory data access | - Memory data access | - Write-back |

##### BR group

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| IF | ID | EX1 | EX2 | EX3 | EX4 | WB |
| - Instruction fetch | - Instruction decode  - Forwarding  - Register read  - Branch determination |  |  |  |  |  |

##### CO group

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| IF | ID | EX1 | EX2 | EX3 | EX4 | WB |
| - Instruction fetch | - Instruction decode  - Forwarding |  |  |  |  | - Control |

For instructions of the ET1 and CO groups, the number of instruction steps and the number of issuance cycles are both 1, and the subsequent instruction is executed consecutively. Even if an instruction immediately following an ET1 or CO group instruction is dependent on that ET1 or CO group instruction, forwarding from the EX1 stage to the ID stage is performed and the subsequent instruction can be executed without stalling the pipeline. Also, even if there is an instruction that is dependent on the ET1 or CO group instruction that is multiple instructions before it, forwarding from each stage is performed and the pipeline is not stalled. However, forwarding does not take place exceptionally if the dependent register is R0, R1, or R6.

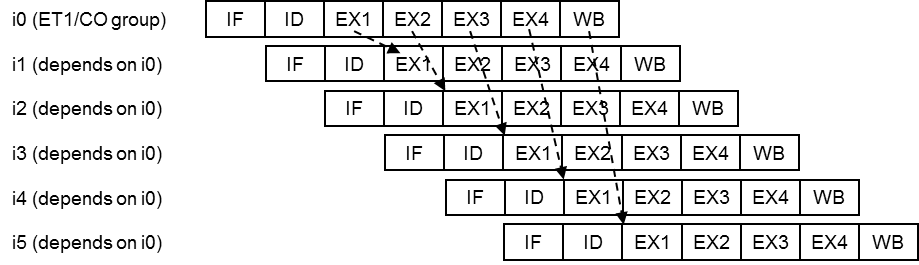


Figure 10‑5 Execution of an ET1/CO Group Instruction and its Dependent Instructions

For instructions of the ES2 group, the number of instruction steps is 2 and the number of issuance cycles is 1. The subsequent instruction is executed consecutively in case it is not dependent on the ES2 group instruction. If an instruction immediately following an ES2 group instruction is dependent on that ES2 group instruction, the pipeline is stalled by one cycle. If there is an instruction that is dependent on the ES2 group instruction that is at least two instructions before it, forwarding from each stage is performed and the pipeline is not stalled. However, forwarding does not take place exceptionally if the dependent register is R0, R1, or R6.

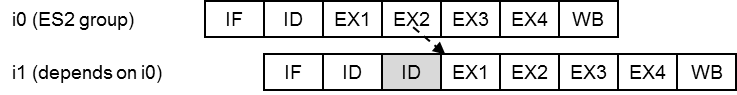


Figure 10‑6 Execution of an ES2 Group Instruction and its Dependent Instructions

For instructions of the ES3/ESF3 group, the number of instruction steps is 3 and the number of issuance cycles is 1. The subsequent instruction is executed consecutively in case it is not dependent on the ES3/ESF3 group instruction. If there is an instruction that is dependent on the ES3/ESF3 group instruction that is immediately before or two instruction before it, the pipeline is stalled. If there is an instruction that is dependent on the ES3/ESF3 group instruction that is at least three instructions before it, forwarding from each stage is performed and the pipeline is not stalled. However, forwarding does not take place exceptionally if the dependent register is R0, R1, or R6.

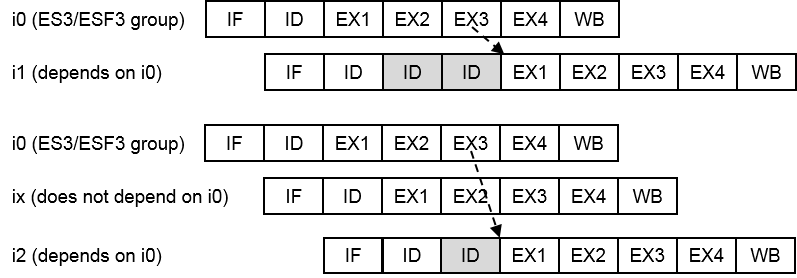


Figure 10‑7 Execution of an ES3/ESF3 Group Instruction and its Dependent Instructions

For instructions of the ESF4 group, the number of instruction steps is 4 and the number of issuance cycles is 1. The subsequent instruction is executed consecutively in case it is not dependent on the ESF4 group instruction. If there is an instruction that is dependent on the ESF4 group instruction that is immediately before, two instruction before, or three instructions before it, the pipeline is stalled. If there is an instruction that is dependent on the ESF4 group instruction that is at least four instructions before it, forwarding from each stage is performed and the pipeline is not stalled. However, forwarding does not take place exceptionally if the dependent register is R0, R1, or R6.

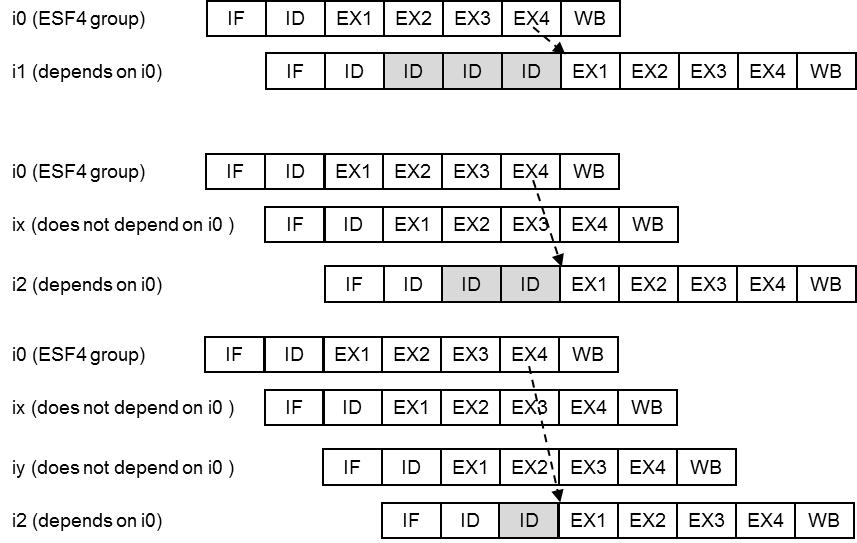


Figure 10‑8 Execution of an ESF4 Group Instruction and its Dependent Instructions

For instructions of the LS group, the number of instruction steps is 4 and the number of issuance cycles is 1 for loading and the number of instruction steps and the number of issuance cycles are both 1 for storage. For a load instruction, the subsequent instruction is executed consecutively in case it is not dependent on the LS group instruction. If an instruction immediately following a load instruction is dependent on that load instruction, the pipeline is stalled by three cycle. If there is an instruction that is dependent on the LS group instruction that is at least four instruction before it, forwarding is performed and the pipeline is not stalled. However, forwarding does not take place exceptionally if the dependent register is R0, R1, or R6.

Since an LS group instruction accesses the data cache, the pipeline is stalled when a cache miss occurs. The number of stall cycles depends on the bus congestion which the CVengine is connected to and the external memory state.

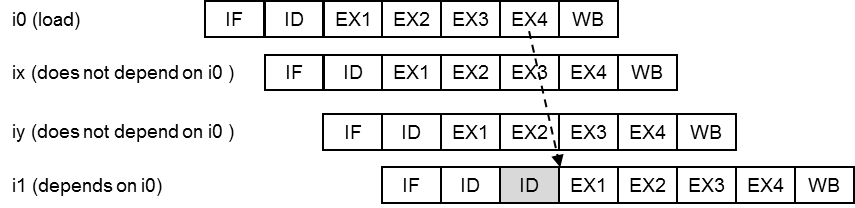


Figure 10‑9 Execution of a Load Instruction and its Dependent Instructions

For storage, the subsequent instruction is always executed consecutively.



Figure 10‑10 Execution of a Store Instruction and its Dependent Instructions

A BR group instruction always causes 1-cycle stall when the branch condition is satisfied.



Figure 10‑11 Execution When the Branch Condition of a BR Group Instruction is Satisfied

If the branch condition is not satisfied, the pipeline is not stalled.



Figure 10‑12 Execution When the Branch Condition of a BR Group Instruction is Not Satisfied

If a flag control instruction is immediately before a BR group instruction, the pipeline is stalled by one cycle. For V3U, there is no pipeline stall.

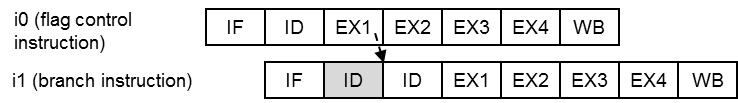


Figure 10‑13 Execution of a BR Group Instruction Immediately Preceded by a Flag Control Instruction

### Pipeline stall on arbitration

Multiple threads in the same core share the operation unit used in executing instructions of the ES2, ESF3 and ESF4 groups. Therefore, when more than two threads are requesting for the same operation unit at the same time, the pipeline may be stalled due to arbitration.

### Pipeline stall on SR flag

When a SR referring instruction comes just after a CMP instruction, it causes for V3M and V3H 1 cycle pipeline stall.

The examples below show cases of 1 cycle stall.

Example 1:

CMP.SLT R3, R0, R30

BF LABEL

Example 2:

CMPU.EQ R3, R25, 8

MOVI R19, 0x0F, SR

For V3U, there is no pipeline stall.

### Pipeline stall on FMAD/FMSU and FADD/FSUB instructions

Pipeline stall on FMAD/FMADU/FMSU/FMSUU and FADD/FADDU/FSUB/FSUBU

FMAD/FMSU followed by an FADD/FSUB will cause a pipeline stall due to arbitration since FMSU/FMAD is executed as a FMUL+FSUB/FADD. The instructions for Uniform storage (e.g. FMADU) are the same. For V3M Ver.2.0, V3H and V3U, the following instruction (e.g. FADD) is delayed by 1 clock cycle.

Example:

FMSU R20, R21, R22

FADD R26, R25, R24

### Forwarding with 32-bit and 64-bit instructions

A 32-bit result cannot be forwarded to a 64-bit source. The code below is an example of no forwarding.

ADD R25, R17, R19

ADD R24, R21, R23

ASMU R24, R24, 14

For V3U, such forwarding from a 32-bit result to a 64-bit source has been added.

On the other hand, a 64-bit result to a 32-bit source, forwarding is done. Below is an example of forwarding is done.

MULLS R16, R22, R23

ADD R25, R17, R19

### Forwarding with R6 register

For V3M Ver.2.0, V3H and V3U, R6 register can be used as a general purpose register. Refer to section 9 Note 4 for detail. Since R6 is not a regular general purpose register, forwarding is done differently to other general purpose registers.

* When LD instruction (only if accessing to Uniform) and RU type instruction are both used, conflict can occur on ID and EX2 stages so pipeline stalls 1 cycle.

LD R16, R17, R3 (Accessing to Uniform)

ADD R18, R19, R20

ADDU R21, R22, $2 (Using R6 as offset base)

* Forwarding is not possible to RU type instructions; pipeline stalls for both cases: using / not using R6 as offset base.

ADD R6, R16, R17

ADDU R18, R6, 4 (RU type)

Forwarding is done to non-RU type instructions, no stall happens in the case below.

ADD R6, R16, R17

ADD R18, R6, R19 (non-RU type)

* Forwarding is not supported for branch instructions. The number of execution steps for branch instructions that update the general purpose register set (e.g. CALL and CALLR) is 1 clock cycle for general purpose registers. For register R6, which does not belong to the general purpose register set, no forwarding is supported, so 5 NOP cycles are added.

## Number of Execution Cycles of Special Instructions of CVengine Thread

The CVengine thread has operation units dedicated to special operations, such as square root. A special operation is executed by writing to a specific register number and the result is stored in the same register number.

After these operations are started, the number of cycles shown in the following table is necessary at minimum to be able to read the result. Since these operation units are shared by multiple threads, if the same operation unit is accessed simultaneously, the number of required cycles will increase when the access right cannot be acquired due to arbitration. Also, as RCP0 (R14) and RCP1 (R15) use the same operation unit, contention occurs when operations to write to these registers happen at the same time.

Table 10‑3 Number of Execution Cycles of Special Instructions

| Operation Type (Register) | Number of Execution Cycles |
| --- | --- |
| SIN (R8) | 5 |
| COS (R9) | 5 |
| 1/SQRT (5) | 5 |
| EXP2 (R11) | 5 |
| LOG2 (R12) | 5 |
| SQRT (R13) | 5 |
| RCP (R14 and R15) | 6 |

When executing an instruction that uses these register values before the result of writing to these registers are ready to be read, the thread will stall until the results are ready to be read.

# Vertex Issue

The Vertex Issue is a sub module of CVengine. The Vertex Issue manages inside of a CVengine cluster. A typical sequence of Vertex Issue behavior is the following.

1. Vertex Issue is invoked by the host processor.
2. Vertex Issue fetches command from the Command List.
3. Upon WPR commands in the Command List, Vertex Issue sets values to the Configuration Registers.
4. Upon POINT and RECT commands in the Command List, Vertex Issue throws a set of XY coordinates to a thread in idle state. Which XY coordinates are assigned to which thread is determined by Vertex Issue automatically.
5. When all the coordinates are processed, the Vertex Issue returns interrupt to the host processor.

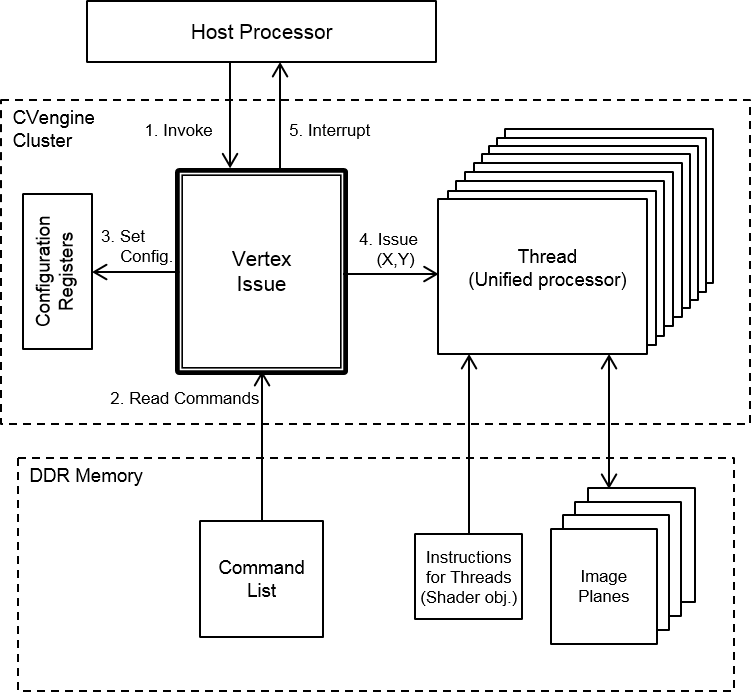


Figure 11.1 Vertex Issue behavior

Sychronization with WUP and SLP commands

To synchronize the PSC with the IMP DMAC, legacy core, and CVengine core, two commands (WUP and SLP) are provided in the IFID block of the PSC.

WUP: This command is issued by a core to wake up another core from the sleep (SLP) state.   
If the WUP command is issued to a core that is operating and the core later attempts to enter the SLP state, the core will immediately wake up and cannot enter the SLP state. However, note that even if the WUP command is issued to a core two or more times, the requests are not accumulated.  
When entering the SLP state, each core specifies which cores should issue the WUP command to wake it up from the SLP state. Each core wakes up only after it has received the WUP signal from all specified cores.

SLP: This command stops command fetch and decoding in the core that has issued the command and makes the core wait for the WUP signal input in the SLP state.  
When issuing this command, the cores that should wake up the current core by sending the WUP signal should be specified. Upon receiving the WUP signal from all specified cores, the core wakes up from the SLP state and resumes command fetch and decoding.

The following illustrates synchronization operation.

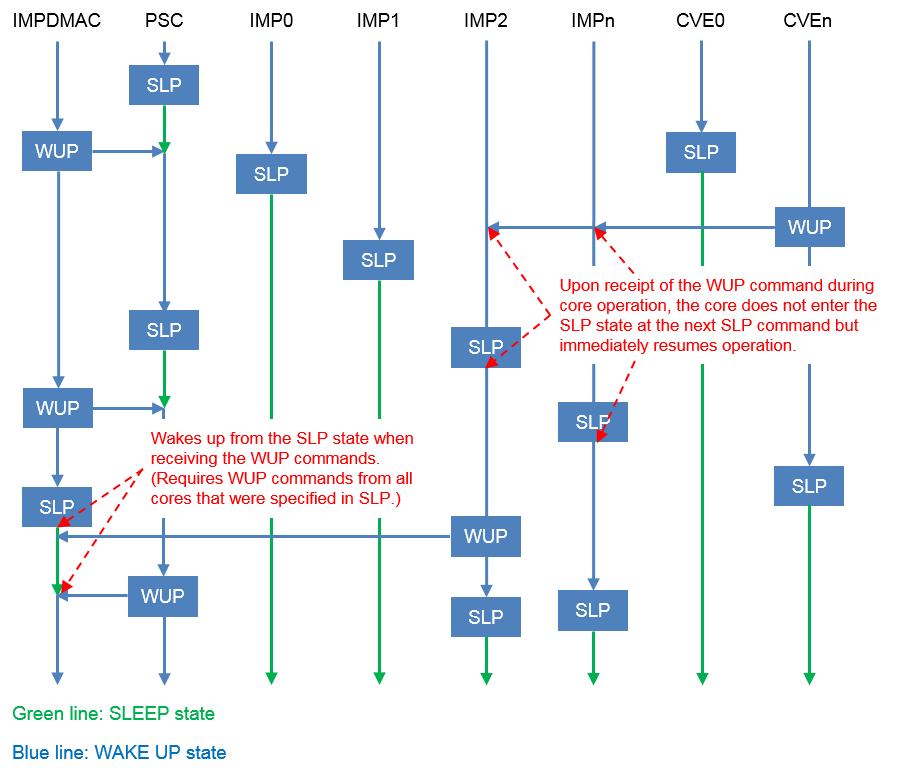


Figure 11‑1 Vertex Issue behavior

# Configuration Registers

It is prohibited to access from ARM CPU except for Status Register1(SR1) while executing CVengine. Also, it is prohibited to change values using WPR command while executing threads. The read values from un-mapped addresses are undefined. Access to addresses other than those in the following lists of registers is prohibited. Operation cannot be guaranteed if other addresses are written.

The registers having address from H’780 to H’790 are duplicated to each thread and accessible from the thread program only. Host CPU cannot access them.

Value of the registers with “WPR support” checked can be set using WPR command in the Command List.

The base addresses of Cluster 0 and Cluster 1 are H’FF980000 and H’FF990000 respectively. For V3H, the base addresses of Cluster 0 to Cluster 4 are H’FF980000, H’FF990000, H’FF9A0000, H’FF9B0000 and H’FFA40000. For V3U, the base addresses of Cluster 0 to Cluster 7 are H’FFA40000, H’FFA50000, H’FFA60000, H’FFA70000, H’FFB40000. H’FFB50000, H’FFB60000 and H’FFB70000.

Table 12‑1 Register Configuration - Control Registers

| Name | Symbol | R/W | Offset | Access Size | Initial Value | Reset | WPR Support |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Version Control Register 0 | VCR0 | R | H'0000 | 32 | H'00000003 | ― | ― |
| ReSeT Register | RSTR | R/W | H'0008 | 32 | H'00000000 | H | ― |
| Control Register | CR | R/W | H'000C | 32 | H'00000000 | S/H | ― |
| Status Register 0 | SR0 | R | H'0010 | 32 | H'00000000 | S/H | ― |
| Status Register 1 | SR1 | R | H'0014 | 32 | H'00000000 | S/H | ― |
| Status Clear Register 1 | SCR1 | -/W | H'0018 | 32 | H'00000000 | ― | ― |
| Interrupt Control Register 1 | ICR1 | R/W | H'001C | 32 | H'00000000 | S/H | ― |
| Interrupt Mask Register 1 | IMR1 | R/W | H'0020 | 32 | H'00000000 | S/H | ― |
| Status Register 2 | SR2 | R | H'0024 | 32 | H'00000000 | S/H | ― |
| Status Clear Register 2 | SCR2 | -/W | H'0028 | 32 | H'00000000 | ― | ― |
| Interrupt Control Register 2 | ICR2 | R/W | H'002C | 32 | H'00000000 | S/H | ― |
| Status Register 3 | SR3 | R | H'0034 | 32 | H'00000000 | S/H | ― |
| Shader Core MaSK Register | SCMSKR | R/W | H'0120 | 32 | H'00000000 | S/H | ― |
| Thread Group Control Regisgter0 | TGCR0 | R/W | H'012C | 32 | H'00000000 | S/H | ―/√/√ |
| SBO CouNT Register0 | SBOCNTR0 | R | H'0130 | 32 | H'00000000 | S/H | ― |
| Descriptor List Start Address Register | DLSAR | R/W | H'0180 | 32 | H'00000000 | S/H | ― |
| CLiPping MINimum Register | CLPMINR | R/W | H'0200 | 32 | H'00000000 | S/H | √ |
| CLiPping MAXimum Register | CLPMAXR | R/W | H'0204 | 32 | H'1FFF1FFF | S/H | √ |
| Shader Performance Control Register | SPCR | R/W | H'0280 | 32 | H'00000000 | S/H | √ |
| Shader Performance BuSY CouNTer Register | SPBSYCNTR | R/W | H'0284 | 32 | H'00000000 | S/H | ― |
| Shader Performance Busy Counter THReshold Register | SPBCTHRR | R/W | H'028C | 32 | H'FFFFFFFF | S/H | ― |
| Data Cache BYPass Area 0 MIN | DCBYPA0MIN | R/W | H'02A0 | 32 | H'FFFFFFFF | S/H | ― |
| Data Cache BYPass Area 0 MAX | DCBYPA0MAX | R/W | H'02A4 | 32 | H'00000000 | S/H | ― |
| Data Cache BYPass Area Enable | DCBYPAEN | R/W | H'02B0 | 32 | H'00000000 | S/H | ― |
| Data Cache BYPass Area ConTroL | DCBYPACTL | R/W | H'02B4 | 32 | H'00000000 | S/H | ― |
| Vs PC Start Address Register | VPCSAR | R/W | H'0320 | 32 | H'00000000 | S/H | √ |
| Vs SINI Start Address Register | VSINISAR | R/W | H'0324 | 32 | H'00000000 | S/H | √ |
| Ps PC Start Address Register | PPCSAR | R/W | H'0328 | 32 | H'00000000 | S/H | √ |
| Ps SINI Start Address Register | PSINISAR | R/W | H'032C | 32 | H'00000000 | S/H | √ |
| Vs I$ Base Address Register | VIBAR | R/W | H'0340 | 32 | Undefined | ― | √ |
| Ps I$ Base Address Register | PIBAR | R/W | H'0344 | 32 | Undefined | ― | √ |
| SBO Base Address Register 0 | SBOBAR0 | R/W | H'0360 | 32 | Undefined | ― | √ |
| SBO Max Memory Size Register 0 | SBOMMSR0 | R/W | H'0364 | 32 | H'00000000 | S/H | √ |
| SBO Control Register 0 | SBOCR0 | R/W | H'0368 | 32 | H'00000000 | S/H | √ |
| DMac Control Register 0 | DMCR0 | R/W | H'03A0 | 32 | H’00003000 | S/H | √ |
| DMac EXTernal MEMory OFfSet ADDress Register | DMEXTMEMOFSADDR | R/W | H'03A4 | 32 | H'00000000 | ― | √ |
| DMac GWM Base ADDress Register | DMGWMBADDR | R/W | H' 03A8 | 32 | H'00000000 | ― | √ |
| DMac GWM STRide Register | DMGWMSTRR | R/W | H'03AC | 32 | H'00000000 | ― | √ |
| DMac LeNGth Register | DMLNGR | R/W | H'03B0 | 32 | H'00000000 | ― | √ |
| IMaGe Base Address Register 0 | IMGBAR0 | R/W | H'0400 | 32 | Undefined | ― | √ |
| IMaGe STRide Register 0 | IMGSTR0 | R/W | H'0404 | 32 | Undefined | — | √ |
| IMaGe Control0 Register 0 | IMGC0R0 | R/W | H'0408 | 32 | H'00000100 | S/H | √ |
| IMaGe Base Address Register 1 | IMGBAR1 | R/W | H'040C | 32 | Undefined | — | √ |
| IMaGe STRide Register 1 | IMGSTR1 | R/W | H'0410 | 32 | Undefined | — | √ |
| IMaGe Control0 Register 1 | IMGC0R1 | R/W | H'0414 | 32 | H'00000100 | S/H | √ |
| IMaGe Base Address Register 2 | IMGBAR2 | R/W | H'0418 | 32 | Undefined | — | √ |
| IMaGe STRide Register 2 | IMGSTR2 | R/W | H'041C | 32 | Undefined | — | √ |
| IMaGe Control0 Register 2 | IMGC0R2 | R/W | H'0420 | 32 | H'00000100 | S/H | √ |
| IMaGe Base Address Register 3 | IMGBAR3 | R/W | H'0424 | 32 | Undefined | — | √ |
| IMaGe STRide Register 3 | IMGSTR3 | R/W | H'0428 | 32 | Undefined | — | √ |
| IMaGe Control0 Register 3 | IMGC0R3 | R/W | H'042C | 32 | H'00000100 | S/H | √ |
| IMaGe Base Address Register 4 | IMGBAR4 | R/W | H'0430 | 32 | Undefined | — | √ |
| IMaGe STRide Register 4 | IMGSTR4 | R/W | H'0434 | 32 | Undefined | — | √ |
| IMaGe Control0 Register 4 | IMGC0R4 | R/W | H'0438 | 32 | H'00000100 | S/H | √ |
| IMaGe Base Address Register 5 | IMGBAR5 | R/W | H'043C | 32 | Undefined | — | √ |
| IMaGe STRide Register 5 | IMGSTR5 | R/W | H'0440 | 32 | Undefined | — | √ |
| IMaGe Control0 Register 5 | IMGC0R5 | R/W | H'0444 | 32 | H'00000100 | S/H | √ |
| IMaGe Base Address Register 6 | IMGBAR6 | R/W | H'0448 | 32 | Undefined | — | √ |
| IMaGe STRide Register 6 | IMGSTR6 | R/W | H'044C | 32 | Undefined | — | √ |
| IMaGe Control0 Register 6 | IMGC0R6 | R/W | H'0450 | 32 | H'00000100 | S/H | √ |
| IMaGe Base Address Register 7 | IMGBAR7 | R/W | H'0454 | 32 | Undefined | — | √ |
| IMaGe STRide Register 7 | IMGSTR7 | R/W | H'0458 | 32 | Undefined | — | √ |
| IMaGe Control0 Register 7 | IMGC0R7 | R/W | H'045C | 32 | H'00000100 | S/H | √ |
| IMaGe SIZE Register 0 | IMGSIZER0 | R/W | H’0460 | 32 | Undefined | — | √ |
| IMaGe CoNSTant Register 0 | IMGCNSTR0 | R/W | H’0464 | 32 | Undefined | — | √ |
| SYNC command CoNFiguration Register03 | SYNCCR1 | R/W | H’04C0 | 32 | H'00000000 | S/H | ― |
| SYNC command CoNFiguration Register47 | SYNCCR2 | R/W | H’04C4 | 32 | H'00000000 | S/H | ― |
| SYNC command CoNFiguration Register811 | SYNCCR3 | R/W | H’04C8 | 32 | H'00000000 | S/H | ― |
| SYNC command CoNFiguration Register1215 | SYNCCR4 | R/W | H’04CC | 32 | H'00000000 | S/H | ― |
| TG Shader Enable Register1 | TGSEN1 | R/W | H’0708 | 32 | H’000000FF | S/H | √ |
| Unified Shader PerFormance Counter ConTrol Register | USPFCCTLR | R/W | H’0740 | 32 | H'00000000 | S/H | √ |
| PerFormance Counter TOTAL Register0 | PFCTOTALR0 | R/W | H’0744 | 32 | H'00000000 | S/H | √ |
| Unified Shader ConTRoL Register 0 | USCTRL0 | R/W | H’0758 | 32 | H'00010000 | S/H | √ |
| TGDMac Control Register0 | TGDMCR0 | R/W | H’0780 | 32 | H’00003000 | S/H | ― |
| TGDMac EXTernal MEMory OFfSet ADDress Register | TGDMEXTMEMOFSADDR | R/W | H’0784 | 32 | H'00000000 | S/H | ― |
| TGDMac LWM Base ADDress Register | TGDMLWMBADDR | R/W | H’0788 | 32 | H'00000000 | S/H | ― |
| TGDmac LWM STRide Register | TGDMLWMSTRR | R/W | H’078C | 32 | H'00000000 | S/H | ― |
| TGDmac LeNGth Register | TGLNGR | R/W | H’0790 | 32 | H'00000000 | S/H | ― |
| UNiform Register n  (n = 0 to 63) \*1 | UNRn | R/W | H’0C00  to H’0CFC | 32 | Undefined | S/H | √ |

\*1: From ARM core, UNR16-63 are write only registers. The read values from these registers are always 0. For V3U, all UNR0-63 are readable and writable. From CVengine thread, all UNR registers are read only.

## Register Description

**[Legend]**

Initial value: Register value after a reset

⎯: Undefined value

R/W: Readable/writable. The written value can be read.

R: Read-only. The write value should be the same as the read value. Only modify the target bit (Read Modify Write) because the initial value should not be changed.

-/W: Write-only. The read value is undefined.

### Version Control Register 0 (VCR0)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Version0 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Version0 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1/0 | 1/0 | 1/0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | Version0 | Product dependent | R | These bits indicate the version of this module.  The value is read.  R-Car V3M: H'00000003  R-Car V3M Ver.2.0: H'00000004  R-Car V3H: H'00000004  R-Car V3U: H'00000005 |

### ReSeT Register (RSTR)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | SWRST |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 1 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 0 | SWRST | 0 | R/W | Software Reset  Software Reset is supplied to CVengine Core if this bit is "1".  It is required to write "0" to this bit to release the software reset before writing other registers.  0: Release the software reset  1: Supply the software reset  Setting this bit to 1 is prohibited when the PS bit in the CR register is 1. In that case, operation is not guaranteed. While CL, Thread or TGDMAC is being executed, the PS bit shows “1”. |

### Control Register (CR)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | PS |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 1 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 0 | PS | 0 | R/W | Processing Start  If "1" is written to this bit, CVengine Core Issue starts fetching and executing CL from the address specified in DLSAR register. During the execution, "1" is read from this bit. Otherwise, "0" is read. It is prohibited to write "1" and "0" during its execution. |

### Status Register 0 (SR0)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | INTCODE | | | | | | | | TRAPCODE | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 24 | INTCODE | H'00 | R | INT interrupt code  When INT command in the command list is executed, the CODE field of the INT command that can be defined by the user is copied to this field. Refer to Section 13.7. This field is updated regardless of the setting of ICR register. |
| 23 to 16 | TRAPCODE | H'00 | R | TRAP interrupt code  When TRAP command in the command list is executed, the CODE field of the TRAP command that can be defined by the user is copied to this field. Refer to Section 13.5. This field is updated regardless of the setting of ICR register. |
| 15 to 0 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |

### Status Register 1 (SR1)

Each bit of this register is only set to 1 when the corresponding bit in the interrupt control register (ICR) is 1. When any bit of this register is set to 1 while the corresponding bit in the interrupt mask register (IMR) is 0, an interrupt is generated. Each bit of this register is cleared to 0 when 1 is written to the corresponding bit in the status clearing register (SCR).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | USINT | USIER | — | — | — | — | — | — | — | SBO0ME | — / PB  COVF | INT | IER | TRAP |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 14 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 13 | USINT | 0 | R | INT instruction interruption of thread master. |
| 12 | USIER | 0 | R | Command error decode  This bit shows that an illegal instruction is decoded in the thread program.  0: Any illegal instruction has not been decoded.  1: An illegal instruction has been decoded.  Additional condition the illegal instruction error takes place is described in Table 10‑1. |
| 11 to 5 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 4 | SBO0ME | 0 | R | SBO0 memory size error |
| 3 | ⎯  PBCOVF | 0 | R | Reserved  This bit is always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here.  Performance Busy Counter Overflow(V3M Ver.2.0, V3H and V3U only)  0: Any performance counter overflow has not been detected.  1: A performance counter overflow has been detected.  Refer to Section 8.9 for detail. |
| 2 | INT | 0 | R | INT command decode  This bit shows that INT command is decoded. Once this bit gets "1", the CL execution is stopped unless this status becomes "0".  0: Any INT command has not been decoded.  1: An INT command has been decoded. |
| 1 | IER | 0 | R | Undefined command decode in CL  This bit shows that undefined command is decoded. Once an undefined command is decoded, the CL execution is stopped.  0: Any unknown command has not been decoded.  1: An unknown command has been decoded. |
| 0 | TRAP | 0 | R | TRAP command decode  This bit shows that TRAP command is decoded, and the CL execution finishes.  0: Any TRAP command has not been decoded.  1: A TRAP command has been decoded. |

### Status Clear Register 1 (SCR1)

Writing 1 to each effective bit of this register clears the corresponding bit in status register 1 (SR1) to 0.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | USINTCLR | USIERCLR | — | — | — | — | — | — | — | SBO0MECLR | — /  PBCOVFCLR | INTCLR | IERCLR | TRACLR |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | -/W | -/W | R | R | R | R | R | R | R | -/W | R-/W | -/W | -/W | -/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 14 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 13 | USINTCLR | 0 | -/W | By writing "1" to this bit, the USINT bit in SR1 register is cleared. |
| 12 | USIERCLR | 0 | -/W | By writing "1" to this bit, the USIER bit in SR1 register is cleared. |
| 11 to 5 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 4 | SBO0MECLR | 0 | -/W | By writing "1" to this bit, the SBO0ME bit in SR1 register is cleared. |
| 3 | ⎯  PBCOVFCLR | 0 | R  -/W | Reserved  This bit is always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here.  By writing "1" to this bit, the PBCOVF bit in SR1 register is cleared.  (For V3M Ver.2.0, V3H and V3U only) |
| 2 | INTCLR | 0 | -/W | By writing "1" to this bit, the INT bit in SR1 register is cleared. |
| 1 | IERCLR | 0 | -/W | By writing "1" to this bit, the IER bit in SR1 register is cleared. |
| 0 | TRACLR | 0 | -/W | By writing "1" to this bit, the TRAP bit in SR1 register is cleared. |

### Interrupt Control Register 1 (ICR1)

When each effective bit of this register is 0, the corresponding bit in status register 1 (SR1) is not set to 1.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | USINTENB | USIERENB | — | — | — | — | — | — | — | SBO0MEENB | — / PBCOVFENB | INTENB | IERENB | TRAENB |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R/W | R/W | R | R | R | R | R | R | R | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 14 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 13 | USINTENB | 0 | R/W | By writing "1" to this bit, the USINT bit in SR1 register can be set. |
| 12 | USIERENB | 0 | R/W | By writing "1" to this bit, the USIER bit in SR1 register can be set. |
| 11 to 5 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 4 | SBO0MEENB | 0 | R/W | By writing "1" to this bit, the SBO0ME bit in SR1 register can be set. |
| 3 | ⎯  PBCOVFENB | 0 | R  R/W | Reserved  This bit is always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here.  By writing "1" to this bit, the PBCOVF bit in SR1 register can be set. (For V3M Ver.2.0, V3H and V3U only) |
| 2 | INTENB | 0 | R/W | By writing "1" to this bit, the INT bit in SR1 register can be set. |
| 1 | IERENB | 0 | R/W | By writing "1" to this bit, the IER bit in SR1 register can be set. |
| 0 | TRAENB | 0 | R/W | By writing "1" to this bit, the TRAP bit in SR1 register can be set. |

### Interrupt Mask Register 1 (IMR1)

An interrupt is generated when any bit of status register 1 (SR1) is set to 1 while the corresponding bit in this register is 0.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | USINTMSK | USIERMSK | — | — | — | — | — | — | — | SBO0MEMSK | — / PBCOVFMSK | INTMSK | IERMSK | TRAMSK |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R/W | R/W | R | R | R | R | R | R | R | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 14 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 13 | USINTMSK | 0 | R/W | By writing "1" to this bit, the interrupt occurred by USINT bit in SR1 register is masked. |
| 12 | USIERMSK | 0 | R/W | By writing "1" to this bit, the interrupt occurred by USIER bit in SR1 register is masked. |
| 11 to 5 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 4 | SBO0MEMSK | 0 | R/W | By writing "1" to this bit, the interrupt occurred by SBO0ME bit in SR1 register is masked. |
| 3 | ⎯  PBCOVFMSK | 0 | R  R/W | Reserved  This bit is always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here.  By writing "1" to this bit, the interrupt occurred by PBCOVF bit in SR1 register is masked. (For V3M Ver.2.0, V3H and V3U only) |
| 2 | INTMSK | 0 | R/W | By writing "1" to this bit, the interrupt occurred by INT bit in SR1 register is masked. |
| 1 | IERMSK | 0 | R/W | By writing "1" to this bit, the interrupt occurred by IER bit in SR1 register is masked. |
| 0 | TRAMSK | 0 | R/W | By writing "1" to this bit, the interrupt occurred by TRAP bit in SR1 register is masked. |

### Status Register 2 (SR2)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SR2 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SR2 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | SR2 | All 0 | R | INT command decode for thread  These bit show that INT command is decoded by thread. Once each bit gets "1", the CL execution is stopped unless each status becomes "0".  0: Any INT command has not been decoded by thread.  1: A INT command has been decoded by thread.  bit31: core3.thread7, bit30: core3.thread6, bit29: cor3.thread5, bit28: core3.thread4, bit27: core3.thread3, bit26: core3.thread2, bit25: core3.thread1, bit24: core3.thread0  bit23: core2.thread7, bit22: core2.thread6, bit21: cor2.thread5, bit20: core2.thread4, bit19: core2.thread3, bit18: core2.thread2, bit17: core2.thread1, bit16: core2.thread0  bit15: core1.thread7, bit14: core1.thread6, bit13: cor1.thread5, bit12: core1.thread4, bit11: core1.thread3, bit10: core1.thread2, bit9: core1.thread1, bit8: core1.thread0  bit7: core0.thread7, bit6: core0.thread6, bit5: cor0.thread5, bit4: core0.thread4, bit3: core0.thread3, bit2: core0.thread2, bit1: core0.thread1, bit0: core0.thread0 |

### Status Clear Register 2 (SCR2)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SCR2 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SCR2 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W | -/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | SCR2 | All 0 | -/W | By writing "1" to each bit, each bit in SR2 register is cleared.  bit31: core3.thread7, bit30: core3.thread6, bit29: cor3.thread5, bit28: core3.thread4, bit27: core3.thread3, bit26: core3.thread2, bit25: core3.thread1, bit24: core3.thread0  bit23: core2.thread7, bit22: core2.thread6, bit21: cor2.thread5, bit20: core2.thread4, bit19: core2.thread3, bit18: core2.thread2, bit17: core2.thread1, bit16: core2.thread0  bit15: core1.thread7, bit14: core1.thread6, bit13: cor1.thread5, bit12: core1.thread4, bit11: core1.thread3, bit10: core1.thread2, bit9: core1.thread1, bit8: core1.thread0  bit7: core0.thread7, bit6: core0.thread6, bit5: cor0.thread5, bit4: core0.thread4, bit3: core0.thread3, bit2: core0.thread2, bit1: core0.thread1, bit0: core0.thread0 |

### Interrupt Control Register 2 (ICR2)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | ICR2 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | ICR2 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | ICR2 | All 0 | R/W | By writing "1" to each bit, each bit in SR2 register can be set when each thread decodes INT command.  bit31: core3.thread7, bit30: core3.thread6, bit29: cor3.thread5, bit28: core3.thread4, bit27: core3.thread3, bit26: core3.thread2, bit25: core3.thread1, bit24: core3.thread0  bit23: core2.thread7, bit22: core2.thread6, bit21: cor2.thread5, bit20: core2.thread4, bit19: core2.thread3, bit18: core2.thread2, bit17: core2.thread1, bit16: core2.thread0  bit15: core1.thread7, bit14: core1.thread6, bit13: cor1.thread5, bit12: core1.thread4, bit11: core1.thread3, bit10: core1.thread2, bit9: core1.thread1, bit8: core1.thread0  bit7: core0.thread7, bit6: core0.thread6, bit5: cor0.thread5, bit4: core0.thread4, bit3: core0.thread3, bit2: core0.thread2, bit1: core0.thread1, bit0: core0.thread0 |

### Status Register 3 (SR3)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SR3 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SR3 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | SR3 | All 0 | R | Indicates whether any illegal instruction has been decoded. To clear any bit of this register, software reset is needed.  0: No illegal instruction has been decoded in the thread.  1: An illegal instruction has been decoded in the thread.  bit31: core3.thread7, bit30: core3.thread6, bit29: cor3.thread5, bit28: core3.thread4, bit27: core3.thread3, bit26: core3.thread2, bit25: core3.thread1, bit24: core3.thread0  bit23: core2.thread7, bit22: core2.thread6, bit21: cor2.thread5, bit20: core2.thread4, bit19: core2.thread3, bit18: core2.thread2, bit17: core2.thread1, bit16: core2.thread0  bit15: core1.thread7, bit14: core1.thread6, bit13: cor1.thread5, bit12: core1.thread4, bit11: core1.thread3, bit10: core1.thread2, bit9: core1.thread1, bit8: core1.thread0  bit7: core0.thread7, bit6: core0.thread6, bit5: cor0.thread5, bit4: core0.thread4, bit3: core0.thread3, bit2: core0.thread2, bit1: core0.thread1, bit0: core0.thread0 |

### Shader Core MaSK Register (SCMSKR)

This register is available only for V3U. When only 1 core is activated, 8 out of 32 threads work. This register is to reduce power consumption. To use debugger, remain this register in default.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | SCMSKR | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 4 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 3 to 0 | SCMSKR | All 0 | R/W | Specify if each core in a cluster is used or not.  [Bit3] 0: Core3 is used. 1: Core3 is not used.  [Bit2] 0: Core2 is used. 1: Core2 is not used.  [Bit1] 0: Core1 is used. 1: Core1 is not used.  [Bit0] 0: Core0 is used. 1: Core0 is not used. |

### Thread Group Control Register 0 (TGCR0)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | TGRSEL | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 3 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 2 to 0 | TGRSEL | All 0 | R/W | To select which core to read the performance counter.  0 : Read PFCTOTALR0 from Core 0  1 : Read PFCTOTALR0 from Core 1  2 : Read PFCTOTALR0 from Core 2  3 : Read PFCTOTALR0 from Core 3  This field works with SELUSPFC. Refer to 12.1.50. |

### SBO CouNT Register0 (SBOCNTR0)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | SBOCNTR0 | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SBOCNTR0 | | | | | | | | | | | | | | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 29 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 28 to 2 | SBOCNTR0 | All 0 | R | It expresses "SBO0 Output Number" (4byte unit).  These bits are cleared by reset, Processing Start, and writing 1 to the SBOCNTCLR bit in SBOCR0.  When SBOBAR0 is written, this register must be cleared. |
| 1 to 0 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |

### Descriptor List Start Address Register (DLSAR)

The “Descripter List” is an alias of “Command List”.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | DLSAR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | DLSAR | | | | | | | | | | | | | | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 2 | DLSAR | All 0 | R/W | Descripter List start address  This register specifies the start address of CL execution. The start address should be specified as 4-byte alignment. |
| 1,0 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |

### CLiPping MINimum Register (CLPMINR)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | XCLPMINR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | YCLPMINR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 16 | XCLPMINR | H'0000 | R/W | X's clipping minimum area (-32768 - 32767) |
| 15 to 0 | YCLPMINR | H'0000 | R/W | Y's clipping minimum area (-32768 - 32767) |

### CLiPping MAXimum Register (CLPMAXR)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | XCLPMAXR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | YCLPMAXR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 16 | XCLPMAXR | H'FFFF | R/W | X's clipping maximum area (-32768 - 32767) |
| 15 to 0 | YCLPMAXR | H'FFFF | R/W | Y's clipping maximum area (-32768 - 32767) |

### Shader Performance Control Register (SPCR)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | BSYCNTENB |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 1 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 0 | BSYCNTENB | 0 | R/W | BuSY CoNTer ENaBle  Counts cycle numbers of CVengine cluster is busy at cluster clock frequency. Value must be set while CVengine cluster is not working.  0: not count busy cycles of CVengine cluster.  1: count busy cycles of CVengine cluster. |

### Shader Performance BuSY CouNTer Register (SPBSYCNTR)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SPBSYCNTR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SPBSYCNTR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | SPBSYCNTR | All 0 | R/W | While SPCR.BSYCNTENB is 1, this register counts cycles of CVengine cluster's busy. CVengine cluster is busy while at least one thread is working. In other words, this counter starts just after invoking POINTS/POINT/RECT and stops just after the TRAP instruction of the last thread is completed.  Value of this register must be written or read while CVengine cluster is not working.  Write 0 to this register to clear the counter.  Hardware does not take care any overflow handlings, so software must take care of it.. |

### Shader Performance Busy Counter THReshold Register (SPBCTHRR)

This register is available only for V3M Ver.2.0, V3H and V3U.

This register is used for the Guard Timer. Refer to Section 8.9 for detail.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SPBSYCNTR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SPBSYCNTR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | SPBSYCNTR | All 1 | R/W | Shader Performance Busy Counter Threshold value.  This register is compared with SPBSYCNTR. |

### Data Cache BYPass Area 0 MIN (DCBYPA0MIN)

This register is available only for V3U.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | DCBYPA0MIN | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | DCBYPA0MIN | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | DCBYPA0MIN | All 1 | R/W | This register holds memory address of the lower border of the D$ bypass memory area. |

### Data Cache BYPass Area 0 MAX (DCBYPA0MAX)

This register is available only for V3U.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | DCBYPA0MAX | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | DCBYPA0MAX | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | DCBYPA0MAX | All 0 | R/W | This register holds memory address of the higher border of the D$ bypass memory area. |

### Data Cache BYPass Area Enable (DCBYPAEN)

This register is available only for V3U.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | DCBYPAEN | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | DCBYPAEN | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | DCBYPAEN | All 0 | R/W | Specify the thread to enable D$ bypassing. Refer to section 8.10.  bit31: Core3 Thread7, bit30: Core3 Thread6,  bit29: Core3 Thread5, bit28: Core3 Thread4,  bit27: Core3 Thread3, bit26: Core3 Thread2  bit25: Core3 Thread1, bit24: Core3 Thread0,  bit23: Core2 Thread7, bit22: Core2 Thread6,  bit21: Core2 Thread5, bit20: Core2 Thread4,  bit19: Core2 Thread3, bit18: Core2 Thread2  bit17: Core2 Thread1, bit16: Core2 Thread0,  bit15: Core1 Thread7, bit14: Core1 Thread6,  bit13: Core1 Thread5, bit12: Core1 Thread4,  bit11: Core1 Thread3, bit10: Core1 Thread2  bit9: Core1 Thread1, bit8: Core1 Thread0,  bit7: Core0 Thread7, bit6: Core0 Thread6,  bit5: Core0 Thread5, bit4: Core0 Thread4,  bit3: Core0 Thread3, bit2: Core0 Thread2  bit1: Core0 Thread1, bit0: Core0 Thread0, |

### Data Cache BYPass Area ConTroL (DCBYPACTL)

This register is available only for V3U.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | DCBYPACTL | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | DCBYPACTL | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | DCBYPACTL | All 0 | R/W | Specify the thread to enable Core# and Thread# replacement to the upper 16 bits of writing data. Refer to section 8.10.  bit31: Core3 Thread7, bit30: Core3 Thread6,  bit29: Core3 Thread5, bit28: Core3 Thread4,  bit27: Core3 Thread3, bit26: Core3 Thread2  bit25: Core3 Thread1, bit24: Core3 Thread0,  bit23: Core2 Thread7, bit22: Core2 Thread6,  bit21: Core2 Thread5, bit20: Core2 Thread4,  bit19: Core2 Thread3, bit18: Core2 Thread2  bit17: Core2 Thread1, bit16: Core2 Thread0,  bit15: Core1 Thread7, bit14: Core1 Thread6,  bit13: Core1 Thread5, bit12: Core1 Thread4,  bit11: Core1 Thread3, bit10: Core1 Thread2  bit9: Core1 Thread1, bit8: Core1 Thread0,  bit7: Core0 Thread7, bit6: Core0 Thread6,  bit5: Core0 Thread5, bit4: Core0 Thread4,  bit3: Core0 Thread3, bit2: Core0 Thread2  bit1: Core0 Thread1, bit0: Core0 Thread0, |

### Vs PC Start Address Register (VPCSAR)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | VPCSAR | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | VPCSAR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 18 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 17 to 0 | VPCSAR | H'0 0000 | R/W | Start address for master thread.  When RECT, POINT and POINTS is executed, thread (master) executes from the address of VPCSAR+VIBAR.(VIBAR is the beginning of the instruction area and VPCSAR is the offset in the area) The area of the instructions must be less than 256KB.  The values of the lowest 2 bits are fixed to 0. |

### Vs SINI Start Address Register (VSINISAR)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | VSINISAR | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | VSINISAR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 18 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 17 to 0 | VSINISAR | H'0 0000 | R/W | SINI start address for master thread.  When SINI is executed, thread (master) executes from the address of VSINISAR+VIBAR(VIBAR is the beginning of the instruction area and VSINISAR is the offset in the area). The area of the instructions must be less than 256KB. The values of the lowest 2 bits are fixed to 0. |

### Ps PC Start Address Register (PPCSAR)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | PPCSAR | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | PPCSAR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 18 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 17 to 0 | PPCSAR | H'0 0000 | R/W | Start address for slave thread.  When RECT, POINT and POINTS are executed, thread (slave) executes from the address of PPCSAR+PIBAR(PIBAR is the beginning of the instruction area and PPCSAR is the offset in the area). The area of the instructions must be less than 256KB. The values of the lowest 2 bits are fixed to 0. |

### Ps SINI Start Address Register (PSINISAR)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | PSINISAR | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | PSINISAR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 18 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 17 to 0 | PSINISAR | H'0 0000 | R/W | SINI start address for slave thread.  When SINI is executed, thread (slave) executes from the address of PSINISAR+PIBAR(PIBAR is the beginning of the instruction area and PSINISAR is the offset in the area). The area of the instructions must be less than 256KB. The values of the lowest 2 bits are fixed to 0. |

### Vs I$ Base Address Register (VIBAR)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | VIBAR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | VIBAR | | | | | | | | | | | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | — | — | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R | R | R | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 5 | VIBAR | Undefined | R/W | These bits set the base address of I$ in the thread (master)  The address is obtained by multiplying the setting by 64. Note that bit 5 must be set to 0. Operation cannot be guaranteed if a value other than 0 is written in bit 5. |
| 4 to 0 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |

### Ps I$ Base Address Register (PIBAR)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | PIBAR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | PIBAR | | | | | | | | | | | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | — | — | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R | R | R | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 5 | PIBAR | Undefined | R/W | These bits set the base address of I$ in thread (slave).  The address is obtained by multiplying the setting by 64. Note that bit 5 must be set to 0. Operation cannot be guaranteed if a value other than 0 is written in bit 5. |
| 4 to 0 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |

### SBO Base Address Register 0 (SBOBAR0)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SBOBAR0 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SBOBAR0 | | | | | | | | | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R | R | R | R | R | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 7 | SBOBAR0 | Undefined | R/W | When this register is written, SBOCNTR0 must be cleared.  Note: Before rewriting this register, SBO0 must be flushed by SYNCM command. |
| 6 to 0 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |

### SBO Max Memory Size Register 0 (SBOMMSR0)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | SBOMMSR0 | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SBOMMSR0 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 29 | ⎯ | All 0 | R/W | Reserved  These bits are always read as 0.  These bits must be set to 0 when these are written.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 28 to 0 | SBOMMSR0 | All 0 | R/W | This register must be set in 4-byte unit.  If it is over this limited value, the SBO0 is issued the transaction with byte-enable 0, and output the interrupt signal.  Note: Before rewriting this register, SBO0 must be flushed by SYNCM command. |

### SBO Control Register 0 (SBOCR0)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SBOCNTCLR | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | SIGNEDEXT | PCKMODE | XYPCK |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 | SBOCNTCLR | 0 | R/W | By writing "1" to this bit, SBOCNTR0 is cleared. The read value is always 0. |
| 30 to 3 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 2 | SIGNEDEXT | 0 | R/W | By writing "1" to this bit, the SBO0 expands X and Y to 32-bit range as signed integer.  By writing "0" to this bit, the SBO0 expands X and Y to 32-bit range as unsigned integer. |
| 1 | PCKMODE | 0 | R/W | It decides the way of concatenation about the coordinates.  0: {x[15:0], y[15:0]}  1: {y[15:0], x[15:0]} |
| 0 | XYPCK | 0 | R/W | By writing "1" to this bit, the SBO0 handles X(16-bit) and Y(16-bit) as a pair of data. |

### DMac Control Register 0 (DMCR0)

This register is for GWM-DMAC. The host processor cannot access this register and only Command list can access..

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | SELOFSADR | — | — | DIR | — | SELPLN | | | SELWAIT | — | DMACST | DMACEX |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R/W | R/W | R/W | R | R | R/W | R | R/W | R/W | R/W | R/W | R | R | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 14 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 13,12 | ⎯ | All 1 | R/W | Reserved  These bits are always read as 1.  **These bits must be set to 1 when these are written.** |
| 11 | SELOFSADR | 0 | R/W | 0: This field is used for DMEXTMEMOFSADDR as X,Y coordinate.  1: This field is used for DMEXTMEMOFSADDR as the offset address in 32-bit format. |
| 10,9 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 8 | DIR | 0 | R/W | Specify the direction of DMA transfer.  0: GWM to the external memory  1: The external memory to GWM |
| 7 | ⎯ | 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 6 to 4 | SELPLN | 000 | R/W | Specify the plane number (0-7) which determines the address and the stride to calculate a base address of the external memory.  Addition of the base address of the plane number specified in this field and the offset makes the base address of the external memory. |
| 3 | SELWAIT | 0 | R/W | Specify the mode of SYNCS instruction of command list.  0: SYNCS exits when the DMAC transfer completes.  1: SYNCS exits when the DMAC startup completes. DMAC registers can be set for a next DMA transfer after the DMAC startup completes. |
| 2 | ⎯ | 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 1 | DMACST | 0 | R | This bit becomes "1" when DMAC is started and returns to "0" when DMAC transfer is finished. |
| 0 | DMACEX | 0 | R/W | By writing "1" to this bit, DMAC is started.  This bit becomes "0" when DMAC has been started.  Note: It is prohibited to write DMAC registers before this bit becomes "0". |

### DMac EXTernal MEMory OFfSet ADDress Register (DMEXTMEMOFSADDR)

This register is for GWM-DMAC. The host processor cannot access this register and only Command list can access.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | DMEXTMEMOFSADDR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | DMEXTMEMOFSADDR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | DMEXTMEMOFSADDR | All 0 | R/W | SELOFSADR=0: bit31-16: X coordinate (signed) \* Depth, bit15-0: Y coordinate (signed).  Addition of the address of the plane number specified by SELPLN and the address of TGDMEXTMEMOFSADDR makes the base address of the external memory in bytes. |

### DMac GWM Base ADDress Register (DMGWMBADDR)

This register is for GWM-DMAC. The host processor cannot access this register and only Command list can access.

Bit 31 to 16 of GWM address will be filled automatically by HW.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | DMGWMBADDR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | RW | RW | RW | RW | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 16 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 15 to 0 | DMGWMBADDR | All 0 | R/W | Specify the base address of GWM. The bit 31 to 16 are ignored. |

### DMac GWM STRide Register (DMGWMSTRR)

This register is for GWM-DMAC. The host processor cannot access this register and only Command list can access.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | DMGWMSTRR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 16 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 15 to 0 | DMGWMSTRR | All 0 | R/W | These bits set the stride of GWM by byte unit. The bit 31 to 16 must be a sign-extension for the bit 15. |

### DMac LeNGth Register (DMLNGR)

This register is for GWM-DMAC. The host processor cannot access this register and only Command list can access.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | XLNG | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | YLNG | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

For V3M Ver.2.0, V3H and V3U.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | XLNG | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | YLNG | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 29  31 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written. |
| 28 to 16  30 to 16 | XLNG | H'0000 | R/W | A pixel width in the range of 1 to 4096.  A pixel width in the range of 1 to 16384. (For V3M Ver.2.0, V3H and V3U only) |
| 15 to 13  15 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written. |
| 12 to 0  14 to 0 | YLNG | H'0000 | R/W | A pixel height in the range of 1 to 4096.  A pixel height in the range of 1 to 16384. (For V3M Ver.2.0, V3H and V3U only) |

### IMaGe Base Address Register n (IMGBARn) (n = 0 to 7)

The number n (n = 0 to 7) of this register shall be set to the Control Registers CR8 and CR9 to select which plane to be read or written by a thread. Refer to section 6.2.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | IMGBARn | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | IMGBARn | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | IMGBARn | Undefined | R/W | When Pixel Format is unsigned char or char, this register must be set in 1-byte unit.  When Pixel Format is unsigned short or short, this register must be set in 2-byte unit.  When Pixel Format is int or float, this register must be set in 4-byte unit. |

### IMaGe STRide Register n (IMGSTRn) (n = 0 to 7)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | IMGSTRn | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 16 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 15 to 0 | IMGSTRn | Undefined | R/W | When Pixel Format is unsigned char or char, this register must be aligned to 1-byte borders.  When Pixel Format is unsigned short or short, this register must be aligned to 2-byte borders.  When Pixel Format is int or float, this register must be aligned to 4-byte borders.  This register must be specified in 1-byte unit in any case. |

### IMaGe Control0 Register n (IMGC0Rn) (n = 0 to 7)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | CLPMODE | | | — | — | — | — | — | PFORMAT2 | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R/W | R/W | R/W | R | R | R | R | R | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | RNDMODE\_F2I | | | — | — | — | SATENBO | — | PFORMAT | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R/W | R/W | R/W | R | R | R | R/W | R | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 27 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 26 to 24 | CLPMODE | 000 | R/W | Specify the Clipping Mode (Border Mode of OpenCV to set the extra padding to the image). Refer to section 8.8.  3'b000: NORMAL (no border)  3'b001: REPLICATE  3'b010: REFLECT  3'b011: REFLECT101  3'b100: WRAP  3'b101: CONSTANT  3'b110: It is prohibited to write  3'b111: It is prohibited to write  Note: The number of times of the repetition is limited once for each top/bottom/left/right of image size in the clipping (except for CONSTANT). |
| 23 to 19 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 18 to 16 | PFORMAT2 | 000 | R/W | Pixel format for thread (slave).  0: unsigned char  1: char  2: unsigned short  3: short  4: Reserved  5: int  6: Reserved  7: float |
| 15 to 11 | ⎯ | 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 10 to 8 | RNDMODE\_F2I | 001 | R/W | Float to Integer Format Conversion Round Mode  0: IEEE round to nearest (Round half to even)  1: IEEE round to zero  2: IEEE round to positive infinity (Not supported)  3: IEEE round to negative infinity (Not supported)  4: round to nearest up (Not supported)  5: round away from zero (Not supported)  6: Reserved  7: Reserved |
| 7 to 5 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 4 | SATENBO | 0 | R/W | Saturation Enable (Output)  0: Disable saturation function.  1: Enable saturation function.  If saturation occurs when the thread write some data to external memory, the output data will be treated as maximum value.  This function has no effect when PFORMAT is set to7 (float). |
| 3 | ⎯ | 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 2 to 0 | PFORMAT | 000 | R/W | Pixel Format for thread (master).  0: unsigned char  1: char  2: unsigned short  3: short  4: Reserved  5: int  6: Reserved  7: float |

### IMaGe SIZE Register 0 (IMGSIZER0)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | IMGSIZER0 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | IMGSIZER0 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | IMGSIZER0 | Undefined | R/W | Specify the range of X/Y coordinate for the clipping mode when CLPMODE in IMGC0R is REPLICATE/REFLECT/REFLECT101/WRAP/CONSTANT.  bit 31-16 : X coordinate (signed, but positive number 0-32767 is valid.)  bit 15-0 : Y coordinate (signed, but positive number 0-32767 is valid.) |

### IMaGe CoNSTant Register 0 (IMGCNSTR0)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | IMGCNSTR0 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | IMGCNSTR0 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | IMGCNSTR0 | Undefined | R/W | Specify the constant value for padding the image for the clipping mode when CLPMODE in IMGC0R is CONSTANT.  It must set in accordance with IMGC0R0.PFORMAT/PFORMAT2.  8bpp: bit7-0 is effective. Bit31-8 must be set to 0.  16bpp: bit15-0 is effective. Bit31-16 must be set to 0.  32bpp: bit31-0 is effective. |

### SYNChronization Control Register 0 (SYNCCR0)

It is recommended to write this register while all the assigned cores are not running. Refer to Section 11.1, Section 13.13and Section 0 for detail.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SYNCC3 | | | | | | | | SYNCC2 | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SYNCC1 | | | | | | | | SYNCC0 | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 24 | SYNCC3 | All 0 | R/W | Specify the core corresponding bit 3 of CORE Number field of WUP and SLP commands.  [31:28] : Core Type  [27:24] : Core Number |
| 23 to 16 | SYNCC2 | All 0 | R/W | Specify the core corresponding bit 2 of CORE Number field of WUP and SLP commands.  [23:20] : Core Type  [19:16] : Core Number |
| 15 to 8 | SYNCC1 | All 0 | R/W | Specify the core corresponding bit 1 of CORE Number field of WUP and SLP commands.  [15:12] : Core Type  [11:8] : Core Number |
| 7 to 0 | SYNCC0 | All 0 | R/W | Specify the core corresponding bit 0 of CORE Number field of WUP and SLP commands.  [7:4] : Core Type  [3:0] : Core Number |

### SYNChronization Control Register 1 (SYNCCR1)

It is recommended to write this register while all the assigned cores are not running. Refer to Section 11.1, Section 13.13and Section 0 for detail.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SYNCC7 | | | | | | | | SYNCC6 | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SYNCC5 | | | | | | | | SYNCC4 | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 24 | SYNCC7 | All 0 | R/W | Specify the core corresponding bit 7 of CORE Number field of WUP and SLP commands.  [31:28] : Core Type  [27:24] : Core Number |
| 23 to 16 | SYNCC6 | All 0 | R/W | Specify the core corresponding bit 6 of CORE Number field of WUP and SLP commands.  [23:20] : Core Type  [19:16] : Core Number |
| 15 to 8 | SYNCC5 | All 0 | R/W | Specify the core corresponding bit 5 of CORE Number field of WUP and SLP commands.  [15:12] : Core Type  [11:8] : Core Number |
| 7 to 0 | SYNCC4 | All 0 | R/W | Specify the core corresponding bit 4 of CORE Number field of WUP and SLP commands.  [7:4] : Core Type  [3:0] : Core Number |

### SYNChronization Control Register 2 (SYNCCR2)

It is recommended to write this register while all the assigned cores are not running. Refer to Section 11.1, Section 13.13and Section 0 for detail.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SYNCC11 | | | | | | | | SYNCC10 | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SYNCC9 | | | | | | | | SYNCC8 | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 24 | SYNCC11 | All 0 | R/W | Specify the core corresponding bit 11 of CORE Number field of WUP and SLP commands.  [31:28] : Core Type  [27:24] : Core Number |
| 23 to 16 | SYNCC10 | All 0 | R/W | Specify the core corresponding bit 10 of CORE Number field of WUP and SLP commands.  [23:20] : Core Type  [19:16] : Core Number |
| 15 to 8 | SYNCC9 | All 0 | R/W | Specify the core corresponding bit 9 of CORE Number field of WUP and SLP commands.  [15:12] : Core Type  [11:8] : Core Number |
| 7 to 0 | SYNCC8 | All 0 | R/W | Specify the core corresponding bit 8 of CORE Number field of WUP and SLP commands.  [7:4] : Core Type  [3:0] : Core Number |

### SYNChronization Control Register 3 (SYNCCR3)

It is recommended to write this register while all the assigned cores are not running. Refer to Section 11.1, Section 13.13and Section 0 for detail.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SYNCC15 | | | | | | | | SYNCC14 | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | SYNCC13 | | | | | | | | SYNCC12 | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 24 | SYNCC15 | All 0 | R/W | Specify the core corresponding bit 15 of CORE Number field of WUP and SLP commands.  [31:28] : Core Type  [27:24] : Core Number |
| 23 to 16 | SYNCC14 | All 0 | R/W | Specify the core corresponding bit 14 of CORE Number field of WUP and SLP commands.  [23:20] : Core Type  [19:16] : Core Number |
| 15 to 8 | SYNCC13 | All 0 | R/W | Specify the core corresponding bit 13 of CORE Number field of WUP and SLP commands.  [15:12] : Core Type  [11:8] : Core Number |
| 7 to 0 | SYNCC12 | All 0 | R/W | Specify the core corresponding bit 12 of CORE Number field of WUP and SLP commands.  [7:4] : Core Type  [3:0] : Core Number |

### TG Shader Enable Register 1 (TGSEN1)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | SLVEN | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | MSTEN | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| R/W: | R | R | R | R | R | R | R | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 24 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 23 to 16 | SLVEN | H'00 | R/W | Slave Thread Enable.  Specify Slave Thread threads.  When the SINI command is used, execution of Slave Thread starts from the PC set in the PSINISAR register. When Slave Thread has been started by Master Thread, execution starts from the PC set in the PPCSAR register.  [Bit7] 0 : Does not use Thread 7 as Slave Thread. 1: Uses Thread 7 as Slave Thread.  [Bit6] 0 : Does not use Thread 6 as Slave Thread. 1: Uses Thread 6 as Slave Thread.  [Bit5] 0 : Does not use Thread 5 as Slave Thread. 1: Uses Thread 5 as Slave Thread.  [Bit4] 0 : Does not use Thread 4 as Slave Thread. 1: Uses Thread 4 as Slave Thread.  [Bit3] 0 : Does not use Thread 3 as Slave Thread. 1: Uses Thread 3 as Slave Thread.  [Bit2] 0 : Does not use Thread 2 as Slave Thread. 1: Uses Thread 2 as Slave Thread.  [Bit1] 0 : Does not use Thread 1 as Slave Thread. 1: Uses Thread 1 as Slave Thread.  [Bit0] 0 : Does not use Thread 0 as Slave Thread. 1: Uses Thread 0 as Slave Thread.  Note: If both MSTEN and SLVEN are set to 0, the thread is disabled. It is prohibited to set both MSTEN and SLVEN to 1. |
| 15 to 8 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 7 to 0 | MSTEN | All 1 | R/W | Master Thread Enable.  Specify Master Thread.  When the SINI command is used, execution of Master Thread starts from the PC set in the VSINISAR register. When Master Thread has been started by Vertex Issue, execution starts from the PC set in the VPCSAR register.  [Bit7] 0 : Does not use Thread 7 as Master Thread. 1: Uses Thread 7 as Master Thread.  [Bit6] 0 : Does not use Thread 6 as Master Thread. 1: Uses Thread 6 as Master Thread.  [Bit5] 0 : Does not use Thread 5 as Master Thread. 1: Uses Thread 5 as Master Thread.  [Bit4] 0 : Does not use Thread 4 as Master Thread. 1: Uses Thread 4 as Master Thread.  [Bit3] 0 : Does not use Thread 3 as Master Thread. 1: Uses Thread 3 as Master Thread.  [Bit2] 0 : Does not use Thread 2 as Master Thread. 1: Uses Thread 2 as Master Thread.  [Bit1] 0 : Does not use Thread 1 as Master Thread. 1: Uses Thread 1 as Master Thread.  [Bit0] 0 : Does not use Thread 0 as Master Thread. 1: Uses Thread 0 as Master Thread.  Note: If both MSTEN and SLVEN are set to 0, the thread is disabled. It is prohibited to set both MSTEN and SLVEN to 1. |

### Unified Shader PerFormance Counter ConTrol Register (USPFCCTLR)

This register is for the performance counter which allows user to aquire detailed performance information from each thread in each core.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | PFCENB | PFCTOVF | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | RW | RW | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | SELUSPFC | | | PFCMOD | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 | PFCENB | 0 | R/W | PerFormance Counter ENaBle  0: Performance counter is disable.  1: Performance counter is enable.  Even if this bit is 1, performance counter is counted only during thread executing. But performance counter is not counted during SINI executing. |
| 30 | PFCTOVF | 0 | R/W | PerFormance Counter Total register OVerFlow  0: Performance Counter Total register is not overflow.  1: Performance Counter Total register is overflow. |
| 29 to 11 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 10 to 8 | SELUSPFC | All 0 | R/W | SELecting Unified Shader to use PerFormance Counter.  0: Select thread 0  1: Select thread 1  2: Select thread 2  3: Select thread 3  4: Select thread 4  5: Select thread 5  6: Select thread 6  7: Select thread 7  This field works with TGRSEL. Refer to 12.1.14. |
| 7 to 0 | PFCMOD | All 0 | R/W | PerFormance Counter MODe  0x00: Number of thread invocation.  0x03: Number of cycles executed (From thread invocation to TRAP)  0x04: Number of instructions fetched  0x05: Number of stall cycles - instruction fetch  0x18: Number of data loaded  0x19: Number of data stored  0x1a: Number of stall cycles – data load/store |

### PerFormance Counter TOTAL Register0 (PFCTOTALR0)

This register works with the register USPFCCTLR. Refer to 12.1.50.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | PFCTOTALR0 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | PFCTOTALR0 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | PFCTOTALR0 | All 0 | R/W | PerFormance Counter TOTAL Register0  This register shows the total cycles counted in the mode set by PFCMOD. Refer to 12.1.50. |

### Unified Shader ConTRoL Register 0 (USCTRL0)

This register is available only for V3M Ver.2.0, V3H and V3U.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | PFTCHEN |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 17 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 16 | PFTCHEN | 1 | R/W | Pre-FeTCH ENable.  0: Disables pre fetch function of thread instruction buffer.  1: Enables pre fetch function of thread instruction buffer. |
| 15 to 0 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |

### TGDMac Control Register 0 (TGDMCR0)

This register is for LWM-TGDMAC. The host processor cannot access this register and only threads can access. Each thread has own register to control LWM-TGDMAC.ashost processor cannot access l use only.is decoded in the Shader thread program.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | EXTSIZE | | SELOFSADR | — | — | DIR | — | SELPLN | | | SELWAIT | — | DMACST | DMACEX |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R/W | R/W | R/W | R | R | R/W | R | R/W | R/W | R/W | R/W | R | R | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 14 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 13,12 | EXTSIZE | 11 | R/W | Specify the transaction size of external memory.  For IMPC:  2'b00: 32 bytes  2'b01: 64 bytes  2'b11: 128 bytes  For DDR access, these bits must be set to 2’b11. |
| 11 | SELOFSADR | 0 | R/W | 0: This field is used for TGDMEXTMEMOFSADDR as X,Y coordinate.  1: This field is used for TGDMEXTMEMOFSADDR as the offset address in 32-bit format. |
| 10,9 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 8 | DIR | 0 | R/W | Specify the direction of DMA transfer.  0: LWM to the external memory  1: The external memory to LWM |
| 7 | ⎯ | 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 6 to 4 | SELPLN | 000 | R/W | Specify the plane number (0-7) which determines the address and the stride for calculating base address of the external memory.  Addition of the base address of the plane number specified in this field and the external offset address makes the base address of the external memory. |
| 3 | SELWAIT |  | R/W | Specify the mode of WAITDMAC instruction.  0: WAITDMAC exits when the DMAC transfer completes.  1: WAITDMAC exits when the DMAC startup completes. DMAC registers can be set for a next DMA transfer after the DMAC startup completes. |
| 2 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 1 | DMACST | 0 | R | This bit becomes "1" when DMAC is started and becomes "0" when DMAC transfer is finished. |
| 0 | DMACEX | 0 | R/W | By writing "1" to this bit, DMAC is started.  This bit becomes "0" when DMAC has been started.  Note: It is prohibited to write DMAC registers before this bit becomes "0". |

### TGDMac EXTernal MEMory OFfSet ADDress Register (TGDMEXTMEMOFSADDR)

This register is for LWM-TGDMAC. The host processor cannot access this register and only threads can access. Each thread has own register to control LWM-TGDMAC. ashost processor cannot access l use only.is decoded in the Shader thread program.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | TGDMEXTMEMOFSADDR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | TGDMEXTMEMOFSADDR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | TGDMEXTMEMOFSADDR | All 0 | R/W | SELOFSADR=0: bit31-16: X coordinate (signed), bit15-0: Y coordinate (signed).  Addition of the address of the plane number specified by SELPLN and the address of TGDMEXTMEMOFSADDR makes the base address of the external memory in bytes. |

### TGDMac LWM Base ADDress Register (TGDMLWMBADDR)

This register is for LWM-TGDMAC. The host processor cannot access this register and only threads can access. Each thread has own register to control LWM-TGDMAC.

Bit 31 to 16 of LWM address will be filled automatically by HW.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | TGDMLWMBADDR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | RW | RW | RW | RW | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 16 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 15 to 0 | TGDMLWMBADDR | All 0 | R/W | Specify the base address of LWM. The bit 31 to 16 are ignored. |

### TGDmac LWM STRide Register (TGDMLWMSTRR)

This register is for LWM-TGDMAC. The host processor cannot access this register and only threads can access. Each thread has own register to control LWM-TGDMAC.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R | R |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | TGDMLWMSTRR | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 16 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 15 to 0 | TGDMLWMSTRR | All 0 | R/W | These bits set the stride of LWM by byte unit. The bit 31 to 16 must be a sign-extension for the bit 15. |

### TGDmac LeNGth Register (TGLNGR)

This register is for LWM-TGDMAC. The host processor cannot access this register and only threads can access. Each thread has own register to control LWM-TGDMAC.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | XLNG | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | — | — | YLNG | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

For V3M Ver.2.0, V3H and V3U.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | XLNG | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | — | YLNG | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R/W: | R | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 29  31 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 28 to 16  30 to 16 | XLNG | H'0000 | R/W | A width in the range of 1 to 4096 in byte unit.  A width in the range of 1 to 16384 in byte unit. (For V3M Ver.2.0, V3H and V3U only) |
| 15 to 13  15 | ⎯ | All 0 | R | Reserved  These bits are always read as 0.  Operation cannot be guaranteed if a value other than 0 is written here. |
| 12 to 0  14 to 0 | YLNG | H'0000 | R/W | A height in the range of 1 to 4096 in byte unit.  A height in the range of 1 to16384 in byte unit. (For V3M Ver.2.0, V3H and V3U only) |

### Uiform Register n (UNRn) (n = 0 to 63)

From ARM core, UNR16-63 are write only registers. The read values from these registers are always 0. For V3U, all UNR0-63 are readable and writable. From CVengine, all UNR registers are read only.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | UNRn | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bit: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | UNRn | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Initial value: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| R/W: | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

| Bit | Bit Name | Initial Value | R/W | Description |
| --- | --- | --- | --- | --- |
| 31 to 0 | UNRn | Undefined | R/W | Set contents of the n-th Uniform storage |

# Command List (CL)

It is prohibited to use instructions not listed on this section.

## POINT (Point)

#### Functionality

Issue coordinates specified in CL to the thread. The coordinates are placed along with the command.

The coordinate space is 65,536  in which -32,768  X  32,767 and -32,768  Y  32,767.

When clip bit is 1, (X, Y) coordinate is clipped according to the clipping area. Refer to section 8.7.

Note: User is required to execute SYNCS instruction with bit[15:13]==b'111 before and after executing this instruction.

#### Command Format

For V3M.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=0011\_0000(H'30) | | | | | | | | Reserved (all 0) | | | | | | | clip | Reserved (all 0) | | | | | | | | N(1≦N≦255) | | | | | | | |
| X0 (signed 16-bit, -32768 - 32767) | | | | | | | | | | | | | | | | Y0 (signed 16-bit, -32768 - 32767) | | | | | | | | | | | | | | | |
| X1(signed 16-bit, -32768 - 32767 | | | | | | | | | | | | | | | | Y1(signed 16-bit, -32768 - 32767) | | | | | | | | | | | | | | | |

…

|  |  |
| --- | --- |
| X(N-1)( signed 16-bit, -32768 - 32767) | Y(N-1)( signed 16-bit, -32768 - 32767) |

For V3M Ver.2.0, V3H and V3U.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=0011\_0000(H'30) | | | | | | | | Reserved (all 0) | | | | | | | clip | N(1≦N≦65535)  N(1≦N≦255) (V3M) | | | | | | | | | | | | | | | |
| X0 (signed 16-bit, -32768 - 32767) | | | | | | | | | | | | | | | | Y0 (signed 16-bit, -32768 - 32767) | | | | | | | | | | | | | | | |
| X1(signed 16-bit, -32768 - 32767 | | | | | | | | | | | | | | | | Y1(signed 16-bit, -32768 - 32767) | | | | | | | | | | | | | | | |

…

|  |  |
| --- | --- |
| X(N-1)( signed 16-bit, -32768 - 32767) | Y(N-1)( signed 16-bit, -32768 - 32767) |

## POINTS (Point Separate)

#### Functionality

Issue coordinates specified in CL to the thread. The coordinates should be placed at the address specified by the “Address” field of this command.

The coordinate space is 65,536  in which -32,768  X  32,767 and -32,768  Y  32,767.

When clip bit is 1, (X, Y) coordinate is clipped according to the clipping area. Refer to section 8.7.

Note: User is required to execute SYNCS instruction with bit[15:13]==b'111 before and after executing this instruction.

#### Command Format

For V3M.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=0011\_0011(H'33) | | | | | | | | Reserved (all 0) | | | | | | | clip | Reserved (all 0) | | | | | | | | N(1≦N≦255) | | | | | | | |
| Address | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

For V3M Ver.2.0, V3H and V3U.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=0011\_0011(H'33) | | | | | | | | Reserved (all 0) | | | | | | | clip | N(1≦N≦65535) | | | | | | | | | | | | | | | |
| Address | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

## RECT (RECTangle)

#### Functionality

Generate coordinates automatically, and issue the generated coordinates to the thread. The coordinates are generated in Raster-scan order.

Note: User is required to execute SYNCS instructions with bit[15:13]==b'111 before and after executing this instruction.

When clip bit is 1, (X, Y) coordinate is clipped according to the clipping area. Refer to section 8.7.

From (Xs, Ys), it generates coordinates according to the parameters. The (X[i][j], Y[i][j]) coordinate is generated like the following behavior model, then it is clipped according to the clipping area if clip bit is 1.

The dX1, dY1, dX2, and dY2 are specified by signed 16-bit fixed point format. The number of the bits of decimal point is decided by sft.

Xs, Ys are specified by unsigned fixed point format. The number of the bits of decimal point is decided by sftxy. The value of sft must be more than the value of sftxy.

(Fo any sftxy value, Xs and Ys work as signed 16-bit)

Note: Uses X[i][j] and Y[i][j] in the following ranges:.

-32768<=X[i][j]<=32767, -32768<=Y[i][j]<=32767

< behavior model >

(0<= sft <= 11, 0<=sftxy<=3, sft >= sftxy)

X\_ls = Xs << (sft - sftxy);

Y\_ls = Ys << (sft - sftxy);

for (j=0; j<Ylen; j++) {

X0[0][j] = X\_ls;

Y0[0][j] = Y\_ls;

X[0][j] = X0[0][j] >> sft;

Y[0][j] = Y0[0][j] >> sft;

for(i=1; i<Xlen; i++) {

X0[i][j] = X0[i-1][j] + dX1;

Y0[i][j] = Y0[i-1][j] + dY2;

X[i][j] = X0[i][j] >> sft;

Y[i][j] = Y0[i][j] >> sft;

}

X\_ls = X\_ls + dX2;

Y\_ls = Y\_ls + dY1;

}



Figure 13‑1 RECT command parameters

#### Command Format

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=0010\_0010(H'22) | | | | | | | | Reserved (all 0) | | | | | | | clip | Reserved (all 0) | | | | | | | | | | sftxy | | sft (0<=sft<=11) | | | |
| dX1(signed 16-bit fixed point) | | | | | | | | | | | | | | | | dY1(signed 16-bit fixed point) | | | | | | | | | | | | | | | |
| dX2(signed 16-bit fixed point) | | | | | | | | | | | | | | | | dY2(signed 16-bit fixed point) | | | | | | | | | | | | | | | |
| Xs(sftxy=0: signed 16-bit,-32768<=Xs<=32767) | | | | | | | | | | | | | | | | Ys(sftxy=0: signed 16-bit, -32768<=Ys<=32767) | | | | | | | | | | | | | | | |
| Xlen(unsigned 16-bit, 1<=Xlen<=65535) | | | | | | | | | | | | | | | | Ylen(unsigned16-bit, 1<=Ylen<=65535) | | | | | | | | | | | | | | | |

## NOP(No OPeration)

#### Functionality

Wait for N cycle, and go to the next command.

#### Command Format

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=1000\_0000(H'80) | | | | | | | | Reserved (all 0)  Syncport  ort  P | | | | | | | | | | | | | | | | | | | N(1≦N≦16) | | | | |

## TRAP(TRAP)

#### Functionality

Finishes CL execution, and reflects it to the Status Register if permitted by Interrupt Control Register. Also, regardless of Interrupt Control Register, the value of CODE field is reflected to Status Register.

Note: User is required to execute SYNCM instruction with bit11=1, bit2=1 and bit0=1 before executing this instruction.

#### Command Format

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=1000\_1000(H'88) | | | | | | | | Reserved (all 0)  Syncport  ort | | | | | | | | | | | | | | | | CODE | | | | | | | |

## WPR(Write PaRameter)

#### Functionality

From the register address specified from ADD field, the successive N registers are written by DATA0 to DATA N-1 respectively. ADD[23:8] should be specified by 1Byte unit.

When WPR is used after executing POINT, POINTS or RECT, SYNCS with bit15=1 and bit14=1 must be executed before executing WPR.

#### Command Format

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=0101\_0101(H'55) | | | | | | | | ADD  Syncport  ort  P  N | | | | | | | | | | | | | | | | N(>=1) | | | | | | | |
| DATA 0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| DATA 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| DATA 2 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

…

|  |
| --- |
| DATA N-1 |

## INT(INTerrupt)

#### Functionality

The INT bit of Status Register gets “1” if it is permitted by Interrupt Control Register. Also, regardless of Interrupt Control Register, the value of CODE field is reflected to Status Register. The CL execution is stopped unless the INT bit of Status Register becomes “0”.

Before stopping CL execution, the instruction buffer in Vertex Issue and the instruction cache of the thread are cleared, and SBO and data cache of the thread are flushed. Since the pre-fetch instruction buffer is cleared with this command, CL commands can be re-written during interruption.

Note: SBO Base Address Register must be set before next CL execution when SBO is used.

**Remarks:**

* Renesas uses the INT command internally in software debug solution.
* INT codes 0xe0 to 0xff is reserved for this debug solution or Renesas internal purpose. 0x00 to 0xdf are available for users.

INT\_PBCOVF is used for the Guard Timer. Refer to Section 8.9 for detail.

#### Command Format

For V3M.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=1000\_0001(H'81) | | | | | | | | Reserved (all 0)  Syncport  ort  P  N＝0 | | | | | | | | | | | | | | | | CODE | | | | | | | |

For V3M Ver.2.0, V3H and V3U.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=1000\_0001(H'81) | | | | | | | | Reserved (all 0) | | | | | | | | \*1 | Reserved (all 0)  Syncport  ort  P  N＝0 | | | | | | | CODE | | | | | | | |

\*1 INT\_PBCOVF

## GOSUB(GOSUB)

#### Functionality

Saves current CL address to the hardware buffer and jumps to the address defined in the command.

#### Command Format

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=1001\_0001(H'91) | | | | | | | | Reserved (all 0)  N | | | | | | | | | | | | | | | | | | | | | | | |
| Address  Syncport  ort  CODE | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

## RET(RETurn)

#### Functionality

Jumps to the address saved by a GOSUB command. Initial value of address buffer is not defined, so executing RET command to an address buffer which had not been saved by GOSUB command is prohibited.

#### Command Format

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=1001\_1001(H'99) | | | | | | | | Reserved (all 0)  N | | | | | | | | | | | | | | | | | | | | | | | |

## SYNCM(SYNChronize Memory)

#### Functionality

Flush or clear each cache specified in the command. After all cache operation specified in the command finishes, the next command is fetched and executed. It is prohibited to specify both flush and clear to the same cache at the same time.

If I$ CLR bit is 1, Instruction cache of the thread is cleared. If SBO CLR bit is 1, SBO is cleared and the data left in SBO disappear. If D$ CLR bit is 1, Data cache of the thread is cleared. If SBO FLUSH bit is 1, SBO is flushed. If D$ FLUSH bit is 1, Data cache of the thread is flushed.

If SBO had been flushed or cleared, SBO Base Address Register must be set before next CL execution.

#### Command Format

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=1000\_1010(H'8A) | | | | | | | | Reserved (all 0) | | | | | | | | | | | 0 | I$  CLR | SBO CLR | 0 | D$  CLR | Reserved (all 0) | | | | | SBO  FLUSH | 0 | D$  FLUSH |

## SYNCS(SYNChronize Shader)

#### Functionality

Wait for the execution of Threads. After all Threads get idle, the next command is executed.

#### Command Format

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=1000\_1011(H'8B) | | | | | | | | Reserved (all 0)  Syncport  ort  P  SEL  Reserved (all 0)  SEL | | | | | | | | See Below | | | Reserved (all 0) | | | | | | | | | | | | |

　　bit[15:13]==b'111 : Wait until CVengine, LWM TGDMAC and GWM-TGDMAC are completed.

　　bit[15:13]==b'110 : Wait until CVengine and LWM TGDMAC is completed.

　　bit[15:13]==b'001 : Wait until GWM-DMAC is completed.

　　bit[15:13]==else : Do not use(b'000 or b'011 or b'101).

## SINI(Shader INItialize)

#### Functionality

This command executes SINI section in CVengine program and waits for the execution of all threads. After all threads get idle, get ready to start all threads. Refer to section 12.1.27 and 12.1.29.

#### Command Format

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=0001\_0001(H'11) | | | | | | | | Reserved (all 0) | | | | | | | | | | | | | | | | | | | | | | | |

## SLP(SLeeP)

#### Functionality

This command stops execution of the succeeding commands in the CL until all specified WUP signals are received.

Specify the necessary WUP signals as the core numbers in the CORE Number field. Multiple bits can be set to 1 at the same time.

The correspondence between core numbers and actual cores can be specified through the SYNCCR1, SYNCCR2, SYNCCR3 and SYNCCR4 registers. Refer to Section 12.1.45, Section 12.1.46, Section 12.1.47 and Section 12.1.48.

Each SYNCCRn register contains 4 SYNCCn fields. Each SYNCCn field is 8-bit long. Its upper 4 bits determine CORE Type and lower 4 bits determine CORE Number.

The available CORE Types are:

0: IMP Core

1: CVengine

2: IMP DMAC

3: Pyramid Scaler

4: Debug Trace Agent (Used by debugger)

For V3M Ver.2.0 and V3H:

0: IMP Core

1: CVengine

2: IMP DMAC

3: Pyramid Scaler

4: CNN

For V3U:

0: IMP Core

1: CVengine

2: IMP DMAC

3: Pyramid Scaler

4: CNN

5: LSDMAC

And possible CORE Numbers for each CORE Type are:

IMP Core: 0-3

CVengine: 0-1

DMAC: 0-1

Pyramid Scaler: 0

Debug Trace Agent: 0 (Used by debugger)

For V3M Ver.2.0:

IMP Core: 0-3

CVengine: 0-1

DMAC: 0-1

Pyramid Scaler: 0

CNN: 0

For V3H:

IMP Core: 0-5 (5 is for Slim-IMP Core)

CVengine: 0-4

DMAC: 0-3

Pyramid Scaler: 0-1

CNN: 0

For V3U:

IMP Core: 0-4

CVengine: 0-7

DMAC: 0-3

Pyramid Scaler: 0-1

CNN: 0-2

LSDMAC: 0

All the cores must have the same CORE Type and CORE Number settings.

Using this command and the WUP command enables synchronization among cores. Refer to Section 11.1.

#### Command Format

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=1010\_0001(H'A1) | | | | | | | | Reserved (all 0) | | | | | | | | SYNCC enable | | | | | | | | | | | | | | | |

## WUP(Wake UP)

#### Functionality

This command issues the WUP signal to the cores specified in this command. Multiple bits can be set to 1 at the same time.

The WUP signal wakes up the target core from the SLP state. The WUP command can be issued to a core even when it is not in the SLP state. However, two or more WUP requests are not accumulated (issuing the WUP command to a core and issuing it again to the same core before the core wakes up is not supported). For V3H and V3U (not for V3M), between 2 WUP commands at least 8 other commands must be inserted.

The correspondence between core numbers and actual cores can be specified through the SYNCCR1, SYNCCR2, SYNCCR3 and SYNCCR4 registers. Refer to Section 12.1.45, Section 12.1.46, Section 12.1.47 and Section 12.1.48.

For settings of SYNCCRn register, refer to 13.13 SLP(SLeeP).

Using this command and the SLP command enables synchronization among cores. Refer to Section 11.1.

#### Command Format

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OP CODE=1010\_0000(H'A0) | | | | | | | | Reserved (all 0) | | | | | | | | SYNCC enable | | | | | | | | | | | | | | | |

# Appendix

This section describes the four methods for improving performance of the CVengine thread.

### A.1.1 Effective Use of 0th Level Cache

Each thread implements the 0th level cache for accessing the D$.

The size of the 0th level cache is 256 bits. When reading data from the D$ in a single access operation, each thread reads the surrounding data that includes the accessed data and is aligned at the 256-bit boundary.

Data to be written from each thread is directly written to the D$. If a hit occurs in the 0th level cache, the 0th level cache is overwritten. If a miss occurs in the 0th level cache, data is written to only the D$ and 0th level cache is not changed.

As a result of monitoring programs, access to the D$ has been observed in multiple cases to be a bottleneck in good performance.

Accordingly, each thread being able to effectively use the 256-bit data in the 0th level cache is expected to improve performance.

Be noticed that the 0th level cache is only connected to D$ thus it does not work for accessing to any other memory (e.g. LWM).

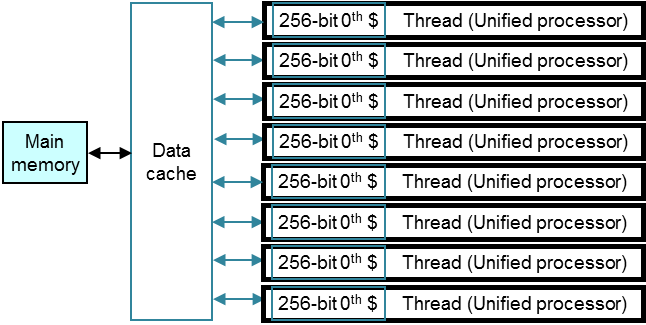


Figure A.1 Arrangement and Configuration of 0th Level Cache

For example, in a case of processing an 8-bit/pixel image, programming so that 32 (256 bits/8 bits = 32) consecutive pixels are processed by a single thread enables all of the necessary data to be accessed by a single access to the D$ and also prevents multiple threads from reading the same 256 bits of data. In a case of 32-bit/pixel data, programming so that 8 (256 bits/32 bits = 8) consecutive pixels are processed by a single thread similarly enables all of the necessary data to be accessed by a single access to the D$ and also prevents multiple threads from reading the same 256 bits of data.

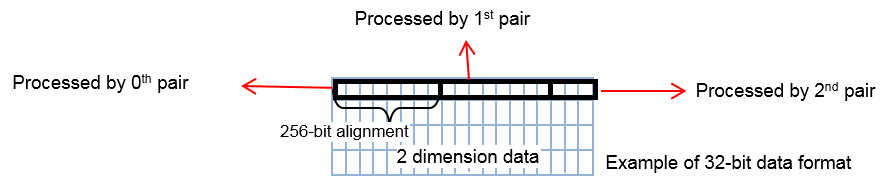


Figure A.2 Example of Efficiently Using 0th Level Cache

### A.1.2 Kernel Processing for Each Area

One thread needs to read multiple surrounding pixels of the center pixel in kernel processing that is specific and frequently used in image recognition. In kernel processing, a surrounding pixel that has been read once becomes the center pixel in the next kernel or can be used as a surrounding pixel. Therefore, in kernel processing, if a certain thread has processed a kernel, the same thread processing the neighboring kernels can reduce the overhead in data loading.

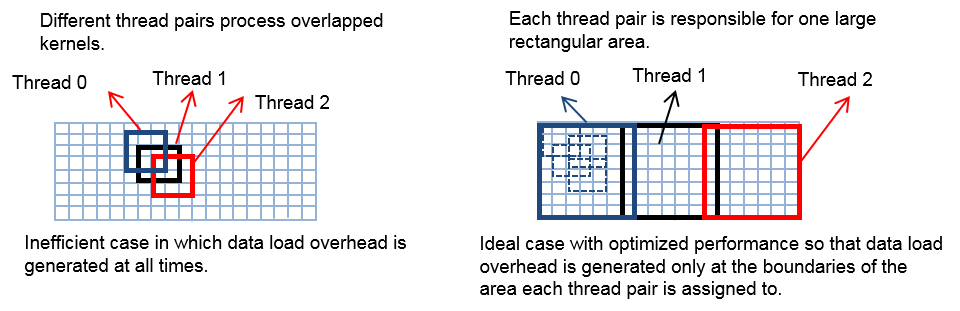


Figure A.3 Example of Efficiently Assigning Processing Areas

### A.1.3 Loop Unrolling

Instructions of the BR group always cause 1-cycle stall when the branch condition is satisfied. Therefore, when there is a loop in a program to generate a branch, stall cycles for the number of loops will occur. Cycles to calculate the number of loops and execute the compare instructions for determining the loop are also required.

Therefore, when there is sufficient program capacity and the number of loops is unique, optimization by loop unrolling is effective to improve the thread performance.

### A.1.4 Effective Use of Instruction Buffers

Each thread implements buffers to hold the instructions fetched from the I$.

This level-0 instruction cache is implemented as a quad buffer, with 256 bits for each of the quad buffers. When fetching an instruction from the L1 I$ in a single access operation, each thread reads the surrounding data that includes the instruction to be fetched and is aligned at the 256-bit boundary.

If the instruction to be fetched is not in the buffer, the surrounding data that includes the instruction to be fetched and is aligned at the 256-bit boundary is stored in one plane. The subsequent 256-bit data is stored in the other plane.

When the last instruction in a buffer X gets executed, then buffer (X + 2)%4 gets pre-loaded with 8 instructions (256 bits).

If inner loop is larger than 23 instructions but less than or equal 32 instructions, disabling pre-fetch function of the instruction buffer can improve performance. The preloading which is taken place at instruction 24 can be prevented and 32 instructions of inner loop can be hold inside instruction buffer. Refer to section 12.1.52. Disabling pre-fetch is available only for V3M Ver.2.0, V3H and V3U.

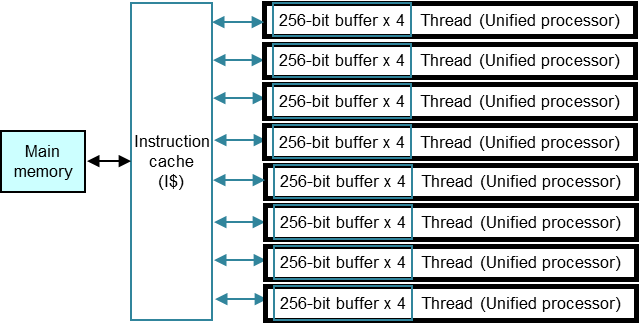


Figure A.4 Arrangement and Configuration of Instruction Buffers

For example, if there is a loop in the program, storing the first instruction of a loop in the start address of the 256-bit aligned memory may improve the efficiency of instruction fetch. The effect is greater especially if the number of iteration of the loop is large.

### A.1.5 Tiling Approach

For V3M Ver.2.0, V3H and V3U, size of D$ is reduced from former R-Car SoCs, refer to 1.2. So it is recommended to use DMA for data loading from DDR to Scratchpad memory or LWM, refer to section 8.3.