ATI CTM Guide

Technical Reference Manual

Version 1.01

Trademarks

AMD, the AMD Arrow logo, AMD Athlon, AMD Opteron and combinations thereof, AMD-XXXX, ATI and ATI product and product-feature names are trademarks of Advanced Micro Devices, Inc.

HyperTransport is a licensed trademark of the HyperTransport Technology Consortium.

Microsoft is a registered trademark of Microsoft Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Disclaimer

While every precaution has been taken in the preparation of this document, Advanced Micro Devices, Inc. assumes no liability with respect to the operation or use of AMD hardware, software, or other products and documentation described herein, for any act or omission of AMD concerning such products or this documentation, for any interruption of service, loss or interruption of business, loss of anticipatory profits, or for punitive, incidental or consequential damages in connection with the furnishing, performance, or use of the AMD hardware, software, or other products and documentation provided herein.

Advanced Micro Devices, Inc. reserves the right to make changes without further notice to a product or system described herein to improve reliability, function or design. With respect to AMD products which this document relates, AMD disclaims all express or implied warranties regarding such products, including but not limited to, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Documentation Updates

AMD is constantly improving its product and associated documentation. To maximize the value of your AMD product, you should ensure that you have the latest documentation. AMD's documentation contains helpful installation/configuration tips and other valuable feature information.

Table of Contents

Chapter 1: Introduction 1

1.1 About this Document 1

1.2 Audience 1

1.3 Related Documents 1

Chapter 2: Specifications 3

2.1 CTM Units 3
2.1.1 The ATI Data Parallel Processor Array 4
2.1.2 Processor Execution Unit 4
2.1.3 Conditional Operation Unit 5
2.1.4 Memory Controller Unit 5

2.2 CTM Commands 8
2.2.1 Processor Execution Unit Commands 10
2.2.2 Memory Controller Unit Commands 13
2.2.3 Conditional Output Unit Commands 21

Chapter 3: DPP Array Instruction Set Architecture 23

3.1 Instructions 23

3.2 Instruction Words 23
3.2.1 Synchronization of instruction streams 24

3.3 ALU Instructions 25
3.3.1 Sources 25
3.3.2 Presubtract 26
3.3.3 Inputs 26
3.3.4 The Operation 27
3.3.5 Instruction Modifiers 29
3.3.6 Writemasks 30
3.3.7 Destination 30
3.3.8 Output 30
3.3.9 Setting Predicate Bits 31
3.3.10 ALU Result 32

3.4 Texture Instructions 32
3.4.1 Operations 33
3.4.2 Semaphore 33

3.5 Flow Control 34
3.5.1 Dynamic Flow Control 34
3.5.2 Stacks and Branch Counters 35
3.5.3 Fields 36
3.5.4 Common Flow Control Statements 38
3.5.5 Optimizations 40

3.6 Note on Floating Point 40
3.6.1 Pervasive Deviations from IEEE 40
3.6.2 ALU Non-Transcendental Floating Point 41
3.6.3 ALU Transcendental Floating Point 42
3.6.4 Texture Floating Point 43

3.7 Errata 43

Chapter 4:
DPP Application Binary Interface 45

4.1 Executable Files 45
4.1.1 File Format 45
4.1.2 ELF Header 45
4.1.3 Program Code Sections 46
4.1.4 Program Loading 46

Chapter 5:
Device Interface 51

5.1 Functions 51
5.1.1 amCloseManagedConnection 51
5.1.2 amCommandBufferConsumed 51
5.1.3 amOpenManagedConnection 51
5.1.4 amSubmitCommandBuffer 52

Chapter 1
Introduction

1.1 About this Document

ATI's "Close To the Metal" (CTM) Device is designed to open up the high-performance, floating-point, parallel processor array found in ATI graphics hardware. CTM consists of this processor array plus a handful of supporting components that control and feed the array.

This manual provides a programmatic overview of the CTM.

1.2 Audience

This manual is intended for experienced design engineers.

1.3 Related Documents

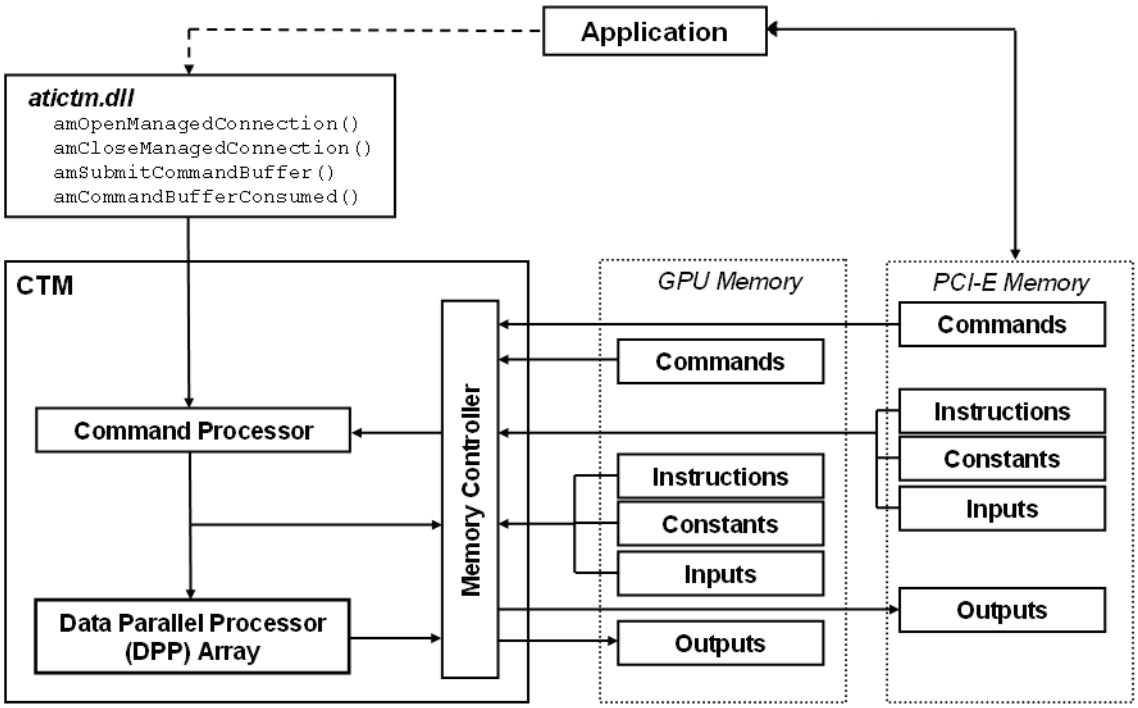
- ATI CTM Device Interface included with CTM distibution.
- Assembler/Disassembler Guide included with CTM distibution.

Chapter 2
Specifications

CTM is designed to expose the parallel array of floating-point processors found in ATI graphics hardware. It is controlled with a few commands to set parameters, invalidate and flush caches, and start the processors in the processing array. These commands reside in memory (see ATI CTM Device Interface for further information). This chapter specifies how CTM reads these commands and its behavior upon processing each.

A block diagram of CTM is presented in Figure 2-1. In addition to the ATI Data Parallel Processor Array (DPP), CTM comprises three major components: the Processor Execution Unit (PE), the Conditional Operation Unit (CO), and the Memory Contoller Unit (MC).

Figure 2-1: CTM Block Diagram



The PE reads commands sequentially from a specified area of memory. Besides redirecting commands to other units within CTM, the PE distributes processing work to the DPP. The computation on an individual processor is subject to a condition returned by the CO. Program output results are written to memory, also based upon a condition returned by the CO. Memory for instructions, constants, program inputs, program outputs, and a buffer used by the CO is accessed through the MC. In addition to satisfying read and write requests, the MC computes memory address offsets, based on a description of the format of the requested data in memory.

2.1 CTM Units

The following sections describe the CTM units in detail.

2.1.1 The ATI Data Parallel Processor Array

The ATI Data Parallel Processor (DPP) Array comprises one or more processors, each a programmable unit that can execute a series of instructions.

Each processor in the array is directed by the Processor Execution Unit (see Section 2.1.2). If a processor is idle, the PE may request that it execute a program. It does so by passing to the processor an identifier pair (i, j), where i and j are non-negative (range-limited) integers, as well as its conditional value. Upon receiving the identifier pair and conditional value, the processor informs the PE that it is busy, resets its program counter to zero, and begins program execution. The processor remains busy until its internal program counter reaches the end of the program, as specified in the Application Binary Interface. After the processor executes the instruction at the end-of-program address, the processor halts and informs that PE that it is again idle.

Instructions for a program, as well as constants, inputs, and outputs to which the program may refer, are stored in memory, and read or written through the MC (see Section 2.1.4). Conceptually, each processor maintains a separate interface to the MC. This interface consists of two non-negative (range-limited) integer indices (x, y) and a field identifying the type of memory access the processor is requesting (program instruction, floating-point constant, integer constant, boolean constant, or input read; or output write).

The (x, y) pair is different for each of the types of memory that a processor may request. For instructions, (x, y) is equal to (pc, 0), where pc is the current program counter. The index pair for each of the constants is (c, 0), where c is the index specified by the program instruction requesting the constant. The index pair and identifier for inputs are specified by the program instruction requesting an input value. The index pair for an output is always the pair assigned to the processor by the PE (i, j), while the identifier for the output is specified in the requesting program instruction.

If conditional output is enabled, output write requests by a processor are conditionally generated, based on a value returned by the Conditional Output Unit (see Section 2.1.3). The processor sends a conditional value (v) and its (i, j) index pair to the CO, and the CO then performs a conditional test based on the value and index pair. If the test passes, the processors dispatch l output write requests to the MC; otherwise no output write requests are generated. The conditional value, v, depends on program that is currently being executed. The value may be specified directly in an instruction in the program, or it may equal the conditional value sent to the processor by the PE. If conditional output is disabled, the processor behaves as if the conditional output test always passes. Conditional output is enabled by setting the condition location to the processors with the **set_cond_loc** command (see page 22).

All processors refer to the same instructions and constants, but may index different input, output, and conditional data. Thus, if multiple processors are working simultaneously, CTM exports a SIMD programming model. Individual processors, however, may or may not execute in SIMD lock-step in a particular CTM implementation; the behavior of individual processors relative to other processors is unspecified.

2.1.2 Processor Execution Unit

The Processor Execution Unit interprets commands from a command buffer, forwarding them to other units in CTM if necessary. Under normal operation, the PE consumes commands as fast as it can process them or pass them along. If, however, the PE receives a **wait_for_idle** command (see page 11), it stops reading commands until all processors in the processor array are idle. Once the processors are idle, the PE again starts to read commands, beginning with the one following the **wait_for_idle** command.

In addition to parsing the command buffer, the PE is responsible for distributing work to the processors in the processor array. The PE's distribution of work is based on the rectangular domain D \subset Z^2, with D = { (i,j) | i0 <= i <= i1, j0 <= j <= j1 }. The parameters i0, j0, i1, and j1 are specified to the PE through the **set_domain** command (see page 10).

When the PE receives a **start_program** command (see page 11), it begins allocating work for the processors. If conditional program execution is disabled, the PE schedules the processors to run the current program once for each index pair (i, j) in the current domain. The specific partitioning of work among the processors and the order in which the index pairs are scheduled is unspecified. As described in Section 2.1.1, the PE sends a corresponding index pair and its conditional value to an idle processor in order to execute the program for that index pair. The result of the entire computation is as if the program were executed in SIMD across all index pairs.[x]

If conditional program execution is enabled, however, program execution on a given pair may be skipped. Prior to scheduling an index pair to a processor, the PE sends it to the Conditional Operation Unit (CO), along with its conditional value. The CO performs a conditional test (as described in Section 2.1.3) based on the index pair and conditional value, and returns its result to the PE. If the test fails, the index pair is skipped; otherwise a processor in the array is scheduled to run the current program on the pair. The conditional value is set with the **set_cond_val** command (see page 10). Conditional program execution is enabled by setting the conditional location to the PE with the **set_cond_loc** command (see page 22).

The PE maintains counters that may be useful in analyzing CTM's performance. These are total clocks since last reset and total clocks during which at least one processor was active since last reset. The **init_perf_counters** (see page 12 for performance counter commands) sets up the counters for initial use. **Start_perf_counters** resets the counters and starts them counting; **Stop_perf_counters** stops the counters. The counters may be read into an array in GPU-addressable system memory using read_perf_counters. **Read_perf_counters** takes one parameter, which gives the GPU address of the first element of the array. Total clocks is written in the first element; the second element is clocks active.

2.1.3 Conditional Operation Unit

The Conditional Operation Unit (CO) performs a conditional test for clients within CTM. Clients include the PE (for conditional program execution) and the DPP (for conditional program output). The CO evaluates a condition based on an index pair (i, j) and a conditional value v, which are both sent to the CO by a requesting client.

The CO test is one of three possible cases: the test always passes, the test always fails, or the test is a comparison between the conditional value v from a client to a value b read from a conditional output buffer residing in memory:

result = v op b, where op is one of <, <=, =, >=, or >

The conditional output buffer value b is obtained by the CO issuing a read request to the Memory Controller Unit with index pair (i, j) and the conditional output buffer identifier (see page 7). The CO test is set with the **set_cond_test** command (see page 21).

In addition, if the conditional test passes, the CO will write the client conditional value v to the conditional output buffer by issuing a write request to the MC with index pair (i, j) and the conditional output identifier.

2.1.4 Memory Controller Unit

The Memory Controller Unit (MC) translates addresses and satisfies requests to read and write memory for clients within CTM. Clients include the PE (for command buffers), the CO (for the conditional buffer), and the DPP (for program instructions, floating-point, integer, and boolean constants, inputs, and outputs). The MC can read or write two kinds of memory: private memory that is accessible only by the MC (local memory), and memory that is accessible both by the MC and a host processor (remote or system memory).

An MC memory address is a 32-bit unsigned integer. The MC distinguishes between local and remote memory by maintaining distinct address ranges for each, within its 32-bit address range. The address mapping is system-dependent and is described in the ATI CTM Device Interface.

The MC computes the memory address as a function of an index pair, (x, y), the number of elements in each row of data (pitch), a base address offset (offset), a tiling format (linear or tiled plus an optional 2x2 superfine tiling on single-channel input data reads), and the bytes per element (bpe) derived from the data format (format). The amount of data read from or written to memory at this address is given by the bytes per element.

The address translation for the different data formats is summarized in the following two tables. If the tiling format of the memory is linear, the address is provided in Table 1. If, on the other hand, the memory is tiled, the address is given in Table 2.

1. Address Translation for Linear Memory Format

Bytes	Bits[31:5]	Bits [4:0]
1	y[11:0]*pitch[13:5]+x[11:5]+offset[31:5]	x[4:0]
2	y[11:0]*pitch[13:4]+x[11:4]+offset[31:5]	x[3:0],0
4	y[11:0]*pitch[13:3]+x[11:3]+offset[31:5]	x[2:0],00
8	y[11:0]*pitch[13:2]+x[11:2]+offset[31:5]	x[1:0],000
16	y[11:0]*pitch[13:1]+x[11:1]+offset[31:5]	x[0],0000

2. Address Translation for Tiled Memory Formats

Bytes	Bits [31:11]	Bits [10:9]	Bits [8:7]	Bits [6:5]	Bits [4:0]
1	y[11:5]*pitch[13:6]+x[11:6]+ offset[31:11]	y[4]^x[6],x[5]^y[5]	y[3]^x[5],x[4]^y[4]	y[2],x[3]	y[1:0],x[2:0]
2	y[11:5]*pitch[13:5]+x[11:5]+ offset[31:11]	y[4]^x[5],x[4]^y[5]	y[3]^x[4],x[3]^y[4]	y[2],x[2]	y[1:0],x[1:0],0
4	y[11:4]*pitch[13:5]+x[11:5]+ offset[31:11]	y[3]^x[5],x[4]^y[4]	y[2]^x[4],x[3]^y[3]	y[1],x[2]	y[0],x[1:0],00
8	y[11:4]*pitch[13:4]+x[11:4]+ offset[31:11]	y[3]^x[4],x[3]^y[4]	y[2]^x[3],x[2]^y[3]	y[1],x[1]	y[0],x[0],000
16	y[11:3]*pitch[13:4]+x[11:4]+ offset[31:11]	y[2]^x[4],x[3]^y[3]	y[1]^x[3],x[2]^y[2]	y[0],x[1]	x[0], 0000

The 2x2 superfine tiling option augments the linear and tiled format addresses described above. When applied to single-channel inputs, it operates as if four independent data elements are requested with index pairs given by (x+1, y), (x, y+1), (x+1, y+1), and (x, y). These four values are packed into the four channels (c0, c1, c2, c3), respectively, of the register specified by the program making the memory request. Without the 2x2 superfine tiling option, a program would need to make four independent input memory requests, across four independent instructions, to achieve the same result. The 2x2 superfine tiling is ignored for all memory clients besides inputs, and its behavior is undefined if the input memory format has more than one channel.

The index pair (x, y) and a unique identifier are sent to the MC by the client requesting the memory read or write. The pitch, offset, tiling, and format parameters associated with the client identifier are maintained in the MC (commands to set these parameters are summarized below), and they are accessed when a client requests a memory transaction. The parameters passed to the memory control unit are different for each of the clients making a memory request. They are detailed, for each of the possible identifiers, in the following subsections.

Input Parameters

The MC supports clients (processors in the data parallel processor array) requesting a memory read for up to 16 distinct program inputs. The (x, y) index pair for a given request is specified in an instruction being executed on one of the processors. The index pair is sent to the MC along with the program input identifier, also specified in the requesting instruction. The pitch, offset, tiling, and format for each input identifier are shared among all processors in the processor array. These values are provided to the MC with the **set_inp_fmt** command (see page 15).

The MC may service any number of requests from the processors during program execution. If the data can be found in the MC input read cache, then the MC will satisfy the request from the cache. Otherwise, it will pull data into the cache (either from GPU or system memory, as appropriate), in the process of servicing the request. The input read cache is shared among all 16 program inputs, and must be invalidated to guarantee correct reading of data that has changed in memory. The input read cache is invalidated with the **inv_inp_cache** command (see page 20).

Output Parameters

The MC supports clients (processors in the data parallel processor array) requesting memory be written for up to 4 distinct program outputs. The (x, y) pair for a given output memory request is equal to the (i, j) output domain pair assigned to a processor or conditional output unit by the PE as described in Section 2.1.2. This index is sent to the MC along with the program output identifier that is specified in the program instruction being executed. The pitch, offset, tiling, and format for each output identifier is shared among all processors in the processor array and the conditional output block. These values are provided to the MC with the **set_out_fmt** command (see page 16).

The MC may service any number of requests from the processors during program execution. If the data can be placed in the MC output write cache, then the MC will place the incoming data into the cache. Otherwise, it will push out a portion of the cache (either to GPU or system memory, as appropriate), if necessary, to free space in the cache for the new request. The output write cache is shared among all 4 program outputs, and must be flushed to guarantee that any data written by the processors are placed in memory. The output write cache is flushed with the **flush_out_cache** command (see page 20).

A write request to the output buffers can be masked with the **set_out_mask** command (see page 21). This command specifies, per output channel, whether the output buffer is updated. If the writemask for a given output channel is 0, then its values are not updated when the MC services a write request.

Conditional Operation Parameters

The MC supports clients (the CO) requesting memory be read and written for a single conditional output buffer. The (x, y) pair is equal to the (i, j) output domain pair assigned to the CO by the PE as described in Section 2.1.2. This pair is sent to the MC along with the conditional output identifier. The pitch, offset, tiling, and format for the output are provided to the MC with the **set_cond_out_fmt** command (see page 18).

The MC may service any number of requests from the CO during program execution. These may be either read requests or write requests.

If the data for a read request can be found in the MC conditional output cache, then the MC will satisfy the request from the cache. Otherwise, it will pull data into the cache (either from local or remote memory, as appropriate), in the process of servicing the request. The conditional output cache must be invalidated to guarantee that data that has changed in memory is properly read. The cache can be invalidated with the **inv_cond_out_cache** command (see page 20).

If the data for a write request can be placed in the MC conditional output cache, then the MC will place the incoming data into the cache. Otherwise, it will push out a portion of the cache (either to GPU or system memory, as appropriate), if necessary, to free space in the cache for the new request. The conditional output cache must be flushed to guarantee that any data written by the conditional output block is placed in memory. The cache can be flushed with the **flush_cond_out_cache** command (see page 20).

A write request to the conditional output buffer can be suppressed with the **set_cond_out_mask** command (see page 21). If the writemask set by this command is 0, then no values are written to the conditional output cache or memory.

Floating Point, Integer, and Boolean Constant Parameters

The Memory Contoller Unit supports clients (processors in the data parallel processor array) requesting a memory read for floating point, integer, and boolean program constants. The (x, y) index pair for a given constant memory request is obtained from the constant index referenced in the program instruction being executed. The index pair is sent to the MCU along with the constant type identifier, which also is specified in the program instruction. The pitch, offset, tiling, and format for each type of constant (three types) is shared among all processors in the processor array. These values are provided to the MCU with the **set_constf_fmt**, **set_consti_fmt**, and **set_constb_fmt** commands.

The MC may service any number of requests from the processors during program execution. If the data can be found in the corresponding MC constant read cache, then the MC will satisfy the request from the cache. Otherwise, it will pull data into the cache (either from GPU or system memory, as appropriate), in the process of servicing the request. The constant read caches must be invalidated to guarantee that data that has changed in memory is properly read. The constant read caches can be invalidated with the **inv_constf_cache**, **inv_consti_cache**, and **inv_constb_cache** commands.

Instruction Parameters

The Memory Contoller Unit supports clients (processors in the data parallel processor array) requesting a memory read for program instructions. The (x, y) index pair for a given memory request is obtained from an internal program counter in each processor that is incremented as the program is executed. The index pair is then sent to the MCU. The pitch, offset, tiling, and format for the instructions are shared among all processors in the processor array. These values are provided to the MC with the **set_inst_fmt** command.

The MC may service any number of requests from the processors during program execution. If the data can be found in the corresponding MC instruction read cache, then the MC will satisfy the request from the cache. Otherwise, it will pull data into the cache (either from GPU or system memory, as appropriate), in the process of servicing the request. The instruction read cache must be invalidated to guarantee that data that has changed in memory is properly read. The instruction read cache can be invalidated with the **inv_inst_cache** command.

2.2 CTM Commands

CTM commands are packed into CTM Command Buffers. A CTM Command Buffer is a tightly packed region in memory interpreted as a sequence of commands and their parameters, starting at the base address of the command buffer. Each CTM command is a 32-bit unsigned integer. It is followed immediately in memory by one or more parameters, each of which is also a 32-bit unsigned integer. No commands take a variable number of parameters and all commands have at least one parameter.

All CTM Commands can be found in Table 3. The table contains the command name, unique opcode, parameters, and a brief description. More complete descriptions are given in the following subsections. Some commands will result in undefined behavior if a program is currently executing on the processor array. Such a command must follow a wait_for_idle command and precede any subsequent start_program command in the command buffer to be predictable. Other commands are pipelined within CTM, and can be issued at any time.

3. CTM Unit Commands

	Opcode	Pipelined	Parameters	Description
Processor Execution Unit Commands				
init_perf_counters	**x'C0010200**	**'yes**	**0: flags**	**Initialize performance counters.**
start_perf_counters	**x'C0000300**	**'yes**	**0: Reserved**	**Set performance counters to zero and start them counting.**
stop_perf_counters	**x'C0000400**	**'yes**	**0: Reserved**	**Stop the performance counters counting.**
read_perf_counters	**x'C0010500**	**'no**	**0: GPU address**	**Write the vales of the performance counters, starting at the supplied GPU address.**
set_cond_val	**x'C0000600**	**'yes**	**0: Value**	**Set the value sent by the PEU to the Conditional Operation Unit when conditional execution is enabled.**
set_domain	**x'C0030700**	**'yes**	**0: i0 1: j0 2: i1 3: j1**	**Set the domain for program execution to be the rectangle (i0, j0) - (i1, j1) inclusive.**
start_program	**x'C0000800**	**'yes**	**0: Reserved**	**Instruct the Processor Execution Unit to start the program.**

	Opcode	Pipelined	Parameters	Description
wait_for_idle	x'C0000900	'yes	0: Reserved	Block reading of the command buffer until all processing in the GPU has completed. Immediately after processing has completed, continue to consume the command buffer.
Memory Controller Unit Commands				
set_inst_fmt	x'C0010A00	'no	0: Base Address 1: Format	Set the base address offset, pitch, tiling, and data format for the program instructions.
set_inp_fmt	x'C0030B00	'no	0: Input Index 1: Base Address 2: Format 3: Height	Set the base address offset, pitch, tiling, and data format for the given input.
set_out_fmt	x'C0030C00	'no	0: Output Index 1: Base Address 2: Format 3: Height	Set the base address offset, pitch, tiling, and data format for the given output.
set_cond_out_fmt	x'C0020D00	'no	0: Base Address 1: Format 2: Height	Set the base address offset, pitch, tiling, and data format for the conditional output.
set_constf_fmt	x'C0010E00	'no	0: Base Address 1: Format	Set the base address offset, pitch, tiling, and data format for the floating point constants.
set_consti_fmt	x'C0010F00	'no	0: Base Address 1: Format	Set the base address offset, pitch, tiling, and data format for the integer constants.
set_constb_fmt	x'C0011000	'no	0: Base Address 1: Format	Set the base address offset, pitch, tiling, and data format for the boolean constants.
inv_inst_cache	x'C0001100	'yes	0: Reserved	Invalidate the instruction cache.
inv_constf_cache	x'C0001200	'yes	0: Reserved	Invalidate the floating point constant cache.
inv_consti_cache	x'C0001300	'yes	0: Reserved	Invalidate the integer constant cache.
inv_constb_cache	x'C0001400	'yes	0: Reserved	Invalidate the boolean constant cache.
inv_cond_out_cache	x'C0001500	'yes	0: Reserved	Invalidate the conditional output cache.
inv_inp_cache	x'C0001600	'yes	0: Reserved	Invalidate the input cache.
flush_out_cache	x'C0001700	'yes	0: Reserved	Flush the output cache.
flush_cond_out_cache	x'C0001800	'yes	0: Reserved	Flush the conditional output cache.
set_out_mask	x'C0001900	'yes	0: Mask	Set the write mask for the output channels.
set_cond_out_mask	x'C0001A00	'yes	0: Mask	Set the write mask for the conditional output buffer.

	Opcode	Pipelined	Parameters	Description
Conditional Output Unit Commands				
set_cond_test	x'C0001B00	'yes	0: Condition	Set the test the conditional output unit will evaluate.
set_cond_loc	x'C0001C00	'yes	0: Location	Set location for the conditional test (PE or DPP).

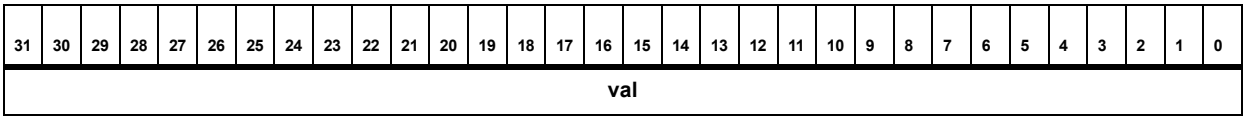
2.2.1 Processor Execution Unit Commands

This section contains all CTM commands for the Processor Execution Unit.

Set Conditional Execution Value Command

The **set_cond_val** command takes a single parameter, a 32-bit value to send to the Conditional Operation Unit when the conditional execution flag is set.

Parameter 0: value

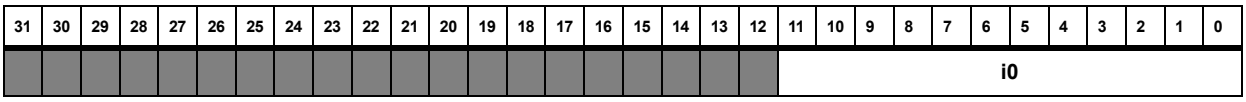


Bits	Field Name	Description
31:0	val	The conditional execution value for the PEU to pass to the Conditional Operation Unit

Set Domain Command

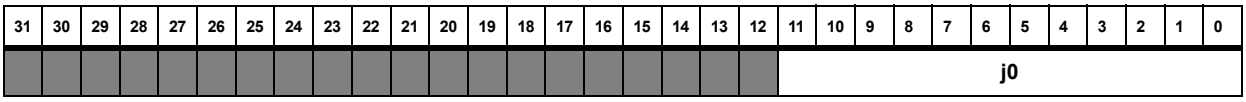
The **set_domain** command takes four parameters, which specify i0, j0, i1, and j1 values for the current domain, as described in Section 3.2. These are unsigned integers, of a range given by the bits in use.

Parameter 0: i0



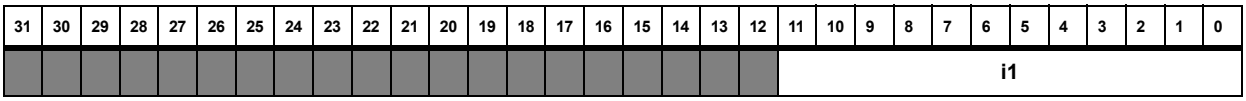
Bits	Field Name	Description
11:0	i0	The i0 domain parameter.
31:12	Reserved	Reserved

Parameter 1: j0



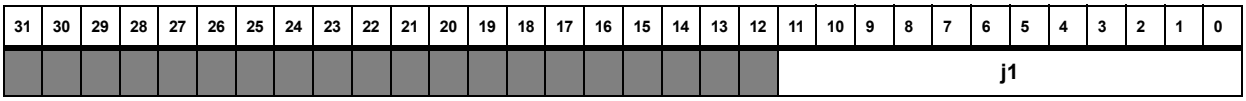
Bits	Field Name	Description
11:0	j0	The j0 domain parameter.
31:12	Reserved	Reserved

Parameter 2: i1



Bits	Field Name	Description
11:0	i1	The i1 domain parameter.
31:12	Reserved	Reserved

Parameter 3: j1

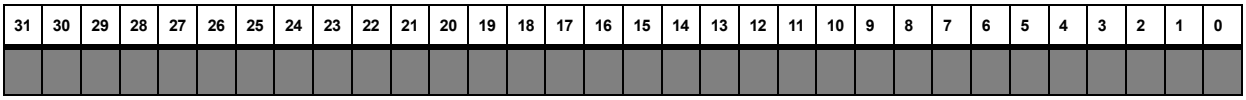


Bits	Field Name	Description
11:0	j1	The j1 domain parameter.
31:12	Reserved	Reserved

Start Program Command

The **start_program** command takes one, reserved parameter. Upon receiving this command, the PE distributes work to the processors as described in Section 3.2.

Parameter 0:

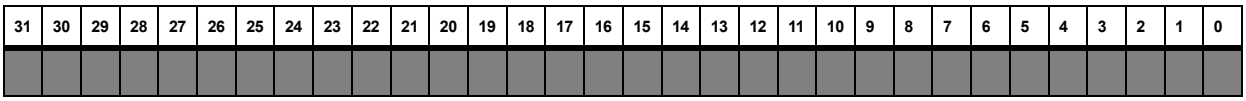


Bits	Field Name	Description
31:0	Reserved	Reserved

Wait For Idle Command

The **wait_for_idle** command takes one, reserved parameter. After receiving this command, the PE blocks all further command processing until all processors in the DPP are idle. Once all processors are idle, processing of subsequent commands resumes.

Parameter 0:

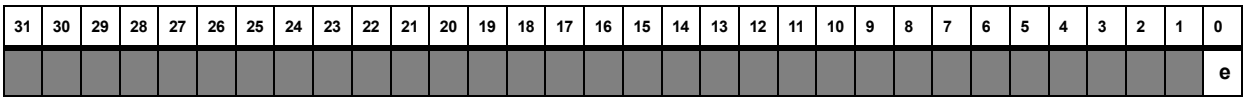


Bits	Field Name	Description
31:0	Reserved	Reserved

Initialize Performance Counters Command

The **init_perf_counters** command takes one parameter. This command executes any setup required for performance counter use. If bit 0 of the parameter is 1, performance counters are enabled. If it is 0, performance counters are disabled, and the other performance counter commands have no effect.

Parameter 0:

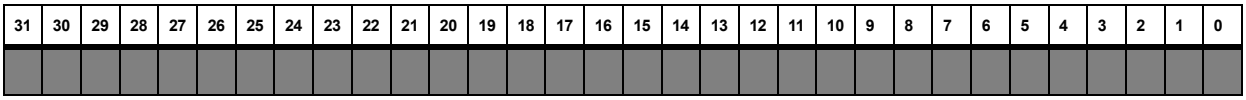


Bits	Field Name	Description
31:1	Reserved	Reserved
0	enable	Turn performance counters on or off.

Start Performance Counter Command

The **start_perf_counters** command takes one, reserved parameter. This command sets enabled performance counters to zero, and then starts the counters counting.

Parameter 0:

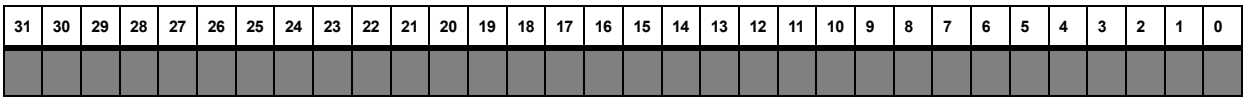


Bits	Field Name	Description
31:0	Reserved	Reserved

Stop Performance Counters Command

The **stop_perf_counters** command takes one, reserved parameter. This command stops any enabled counters counting.

Parameter 0:



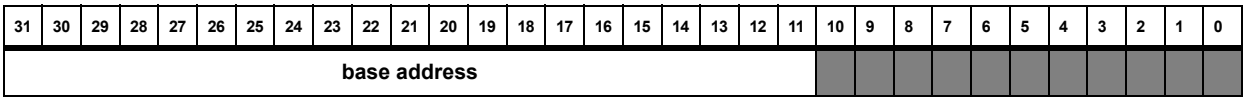
Bits	Field Name	Description
31:0	Reserved	Reserved

Read Performance Counters Command

The **read_perf_counters** command takes one parameter. This command transfers the performance counters to an area in memory beginning at the GPU address supplied in the parameter. The counters are written as 32-bit unsigned integers in the order given in Section 2.1.2 .

If performance counters are disabled, this command has no effect.

Parameter 0: base address

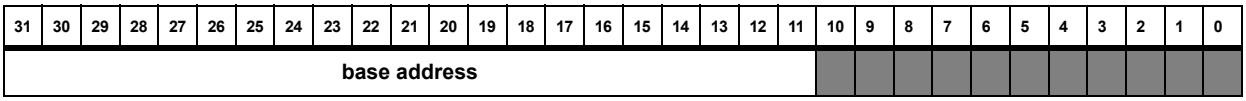


Bits	Field Name	Description
10:0	Reserved	Reserved
31:11	base address	The 2K-aligned address where the performance counter results will be placed.

2.2.2 Memory Controller Unit Commands

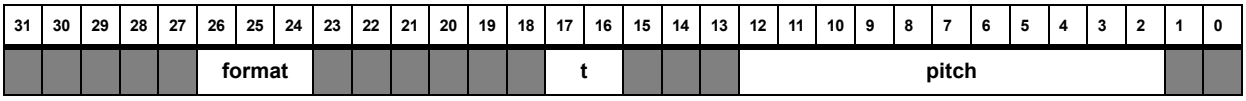
This section contains all of the commands for the memory controller unit.

All address values are provided in the same format: a 2K-aligned base format. The lower eleven bits are ignored by CTM, and the address is treated as if all of the bits were zero. These base addresses are used in the address calculation of the MC, as described in Section 2.1.4. The base address parameter is:



Bits	Field Name	Description
10:0	Reserved	Reserved
31:11	base address	The 2K-aligned address.

Similarly, all format parameters are expressed the same way. The format contains the pitch, tiling format, and data format of the information in memory to which it refers. The pitch is given in multiples of 4 (the lowest two bits of the pitch must be zero). The pitch, tiling, and data formats are used by the MC to calculate an address offset from an (x, y) index pair, as described in Section 2.1.4. The format parameter is:



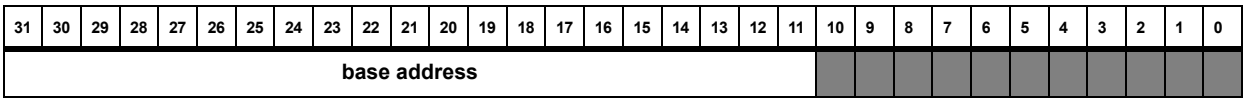
Bits	Field Name	Description
1:0	Reserved	Reserved
12:2	pitch	The pitch in multiples of 4
15:13	Reserved	Reserved
17:16	tiling	Tiling format (possible values): • 0 - LINEAR • 1 - TILED • 2 - LINEAR_INP_2X2 • 3 - TILED_INP_2X2

Bits	Field Name	Description
23:18	Reserved	Reserved
26:24	format	Data format (possible values): • 0 - UINT16_1 • 1 - UINT8_4 • 2 - FLOAT32_1 • 3 - FLOAT32_2 • 4 - FLOAT32_4 • > 5 - Reserved
31:27	Reserved	Reserved

Set Instruction Format Command

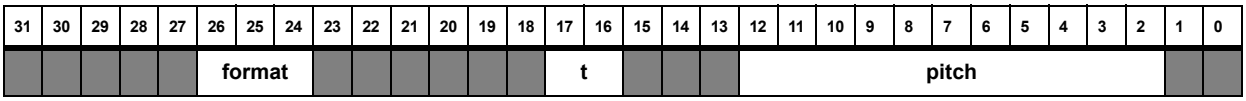
The **set_inst_fmt** command takes two parameters. The first parameter is the base address for the initial program instruction in memory. The second contains pitch, tiling, and format information for the program data.

Parameter 0: base address



Bits	Field Name	Description
10:0	Reserved	Reserved
31:11	base address	The 2K-aligned address at which the first program instruction is located.

Parameter 1: format

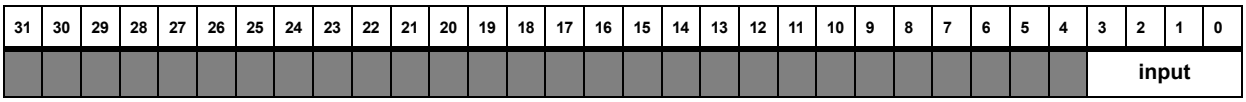


Bits	Field Name	Description
1:0	Reserved	Reserved
12:2	pitch	The pitch in multiples of 4
15:13	Reserved	Reserved
17:16	tiling	Tiling format (possible values): • 0 - LINEAR • 1 - TILED • 2 - LINEAR_INP_2X2 • 3 - TILED_INP_2X2
23:18	Reserved	Reserved
26:24	format	Data format (possible values): • 0 - UINT16_1 • 1 - UINT8_4 • 2 - FLOAT32_1 • 3 - FLOAT32_2 • 4 - FLOAT32_4 • > 5 - Reserved
31:27	Reserved	Reserved

Set Input Format Command

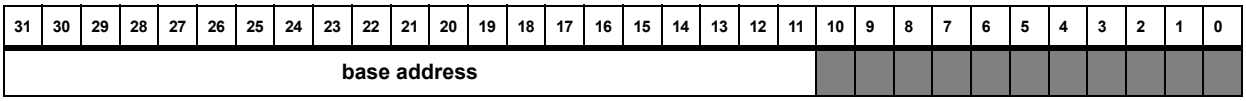
The **set_inp_fmt** command takes four parameters. The first parameter is the index of the input to which the following parameters apply; the second parameter is the base address for the given input in memory; the third contains its pitch, tiling, and format information; and the fourth is the height of the input.

Parameter 0:



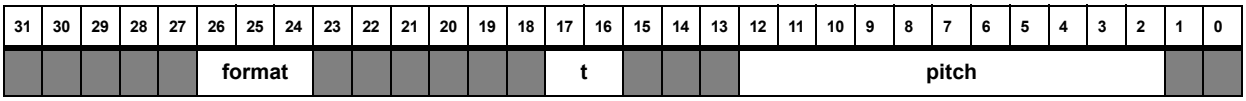
Bits	Field Name	Description
3:0	input	The input to which this command applies.
31:4	Reserved	Reserved

Parameter 1:



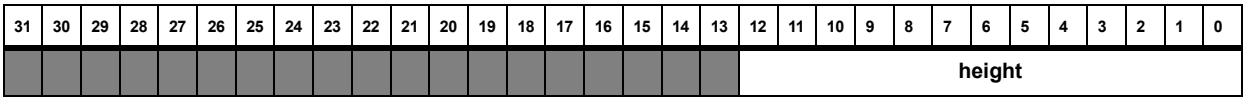
Bits	Field Name	Description
10:0	Reserved	Reserved
31:11	base address	The 2K-aligned address at which the given input is located.

Parameter 2:



Bits	Field Name	Description
1:0	Reserved	Reserved
12:2	pitch	The pitch in multiples of 4
15:13	Reserved	Reserved
17:16	tiling	Tiling format (possible values): • 0 - LINEAR • 1 - TILED • 2 - LINEAR_INP_2X2 • 3 - TILED_INP_2X2
23:18	Reserved	Reserved
26:24	format	Data format (possible values): • 0 - UINT16_1 • 1 - UINT8_4 • 2 - FLOAT32_1 • 3 - FLOAT32_2 • 4 - FLOAT32_4 • > 5 - Reserved
31:27	Reserved	Reserved

Parameter 3:

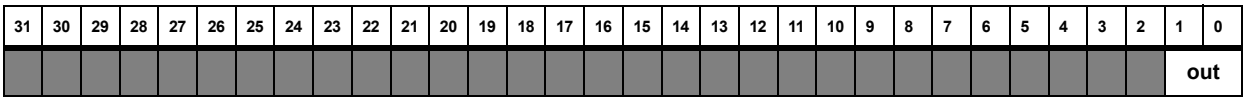


Bits	Field Name	Description
12:0	height	The height
31:13	Reserved	Reserved

Set Output Format Command

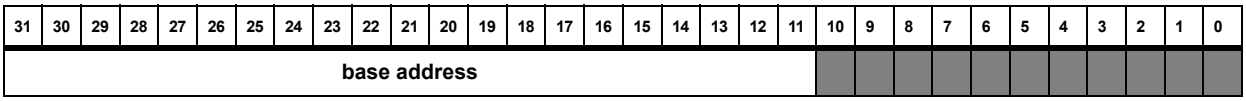
The **set_out_fmt** command takes four parameters. The first parameter is the index of the output to which the following parameters apply; the second parameter is the base address for the given output in memory; the third contains its pitch, tiling, and format information; and the fourth is the height of the given input.

Parameter 0:



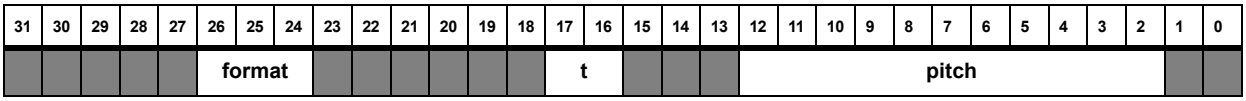
Bits	Field Name	Description
1:0	output	The output to which this command applies.
31:2	Reserved	Reserved

Parameter 1:



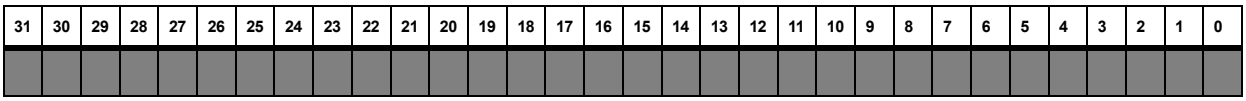
Bits	Field Name	Description
10:0	Reserved	Reserved
31:11	base address	The 2K-aligned address at which the given output is located.

Parameter 2:



Bits	Field Name	Description
1:0	Reserved	Reserved
12:2	pitch	The pitch in multiples of 4
15:13	Reserved	Reserved
17:16	tiling	Tiling format (possible values): • 0 - LINEAR • 1 - TILED • 2 - LINEAR_INP_2X2 • 3 - TILED_INP_2X2
23:18	Reserved	Reserved
26:24	format	Data format (possible values): • 0 - UINT16_1 • 1 - UINT8_4 • 2 - FLOAT32_1 • 3 - FLOAT32_2 • 4 - FLOAT32_4 • > 5 - Reserved
31:27	Reserved	Reserved

Parameter 3:

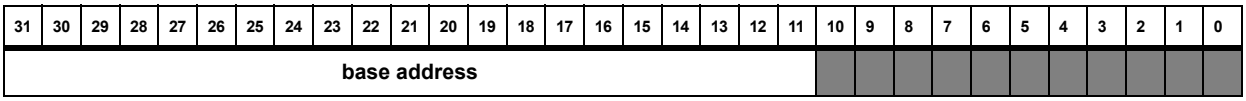


Bits	Field Name	Description
12:0	height	The height
31:13	Reserved	Reserved

Set Conditional Output Format Command

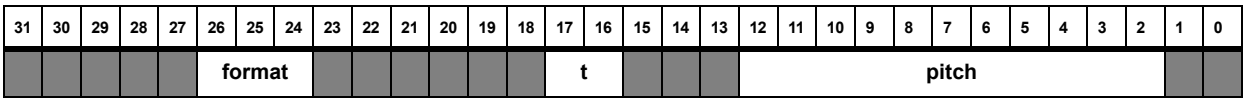
The **set_cond_out_fmt** command takes three parameters. The first parameter is the base address for the conditional output in memory; the second contains its pitch, tiling, and format information; and the third is the height of the conditional output buffer.

Parameter 0:



Bits	Field Name	Description
10:0	Reserved	Reserved
31:11	base address	The 2K-aligned address at which the conditional output is located.

Parameter 1:



Bits	Field Name	Description
1:0	Reserved	Reserved
12:2	pitch	The pitch in multiples of 4
15:13	Reserved	Reserved
17:16	tiling	Tiling format (possible values): • 0 - LINEAR • 1 - TILED • 2 - LINEAR_INP_2X2 • 3 - TILED_INP_2X2
23:18	Reserved	Reserved
26:24	format	Data format (possible values): • 0 - UINT16_1 • 1 - UINT8_4 • 2 - FLOAT32_1 • 3 - FLOAT32_2 • 4 - FLOAT32_4 • > 5 - Reserved
31:27	Reserved	Reserved

Parameter 2:

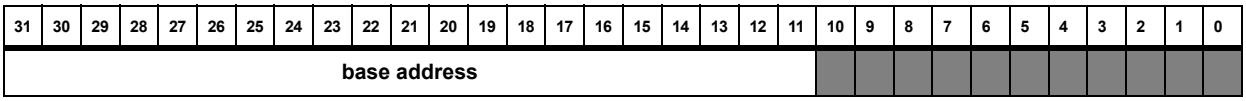
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																			height												

Bits	Field Name	Description
12:0	height	The height
31:13	Reserved	Reserved

Set Constant Format Commands

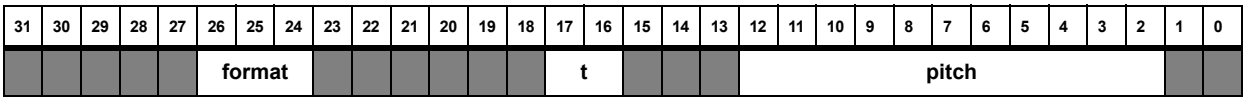
The **set_constf_fmt**, **set_consti_fmt**, and **set_constb_fmt** commands all have the same form. They take two parameters. The first parameter is the base address for the corresponding constants in memory. The second contains the corresponding pitch, tiling, and format information.

Parameter 0: base address



Bits	Field Name	Description
10:0	Reserved	Reserved
31:11	base address	The 2K-aligned address at which the constants are located.

Parameter 1: format

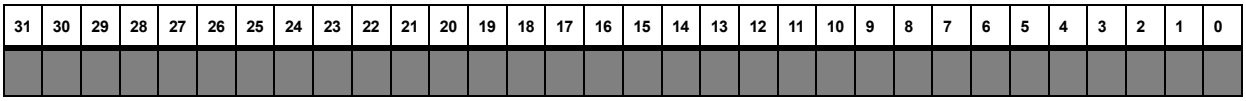


Bits	Field Name	Description
1:0	Reserved	Reserved
12:2	pitch	The pitch in multiples of 4
15:13	Reserved	Reserved
17:16	tiling	Tiling format (possible values): • 0 - LINEAR • 1 - TILED • 2 - LINEAR_INP_2X2 • 3 - TILED_INP_2X2
23:18	Reserved	Reserved
26:24	format	Data format (possible values): • 0 - UINT16_1 • 1 - UINT8_4 • 2 - FLOAT32_1 • 3 - FLOAT32_2 • 4 - FLOAT32_4 • > 5 - Reserved
31:27	Reserved	Reserved

Invalidate Cache Commands

The **inv_inst_cache**, **inv_constf_cache**, **inv_consti_cache**, **inv_constb_cache**, **inv_inp_cache**, and **inv_cond_out_cache** commands all take one, reserved parameter. Upon receiving one of these commands, the MC invalidates the corresponding cache as described in Section 2.1.4.

Parameter 0:



Bits	Field Name	Description
31:0	Reserved	Reserved

Flush Cache Commands

The **flush_out_cache** and **flush_cond_out_cache** commands all take one, reserved parameter. Upon receiving one of these commands, the MC flushes the corresponding cache as described in Section 2.1.4.

Parameter 0:

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

Bits	Field Name	Description
31:0	Reserved	Reserved

Set Output Write Mask Command

The **set_out_mask** command takes a single parameter, a bitfield specifying which channels in an output write request are passed through to memory. If the bit is zero, the corresponding channel of the output data is ignored. The specified mask is active until the next **set_out_mask** command is received.

Parameter 0: writemask

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
																												c3	c2	c1	c0

Bits	Field Name	Description
0	channel 0 writemask	If 1, then write channel 0 of output data upon request; if 0, do not write
1	channel 1 writemask	If 1, then write channel 1 of output data upon request; if 0, do not write
2	channel 2 writemask	If 1, then write channel 2 of output data upon request; if 0, do not write
3	channel 3 writemask	If 1, then write channel 3 of output data upon request; if 0, do not write
31:4	Reserved	Reserved

Set Conditional Output Mask Command

The **set_cond_out_mask** command takes a single parameter that specifies whether the conditional output result is written to the conditional output buffer. If the mask is 1, then the output value is written to the buffer. If it is 0, then the output value is not written to the buffer. The mask is maintained until the next **set_cond_out_mask** command.

Parameter 0: writemask

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
																															w

Bits	Field Name	Description
0	writemask	If 1, then write the conditional value to the conditional output buffer; if 0, do not write
31:1	Reserved	Reserved

2.2.3 Conditional Output Unit Commands

This section contains all CTM commands for the Conditional Output Unit (CO).

Set Conditional Test Command

The **set_cond_test** command takes a single parameter, which specifies the test condition that the CO will perform. This test is active until the next **set_cond_test** command is received.

Parameter 0: condition

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
																													cond		

Bits	Field Name	Description
2:0	condition	**Test Condition:** • 0 - Never (always false) • 1 - Less (true if client value is less than buffer value) • 2 - Less or Equal (true if client value is less or equal to buffer value) • 3 - Equal (true if client value equals buffer value) • 4 - Greater or Equal (true if client value is greater or equal to buffer value) • 5 - Greater (true if client value is greater than buffer value) • 6 - Not Equal (true if client value is not equal to buffer value) • 7 - Always (always true)
31:3	Reserved	Reserved

Set Condition Location Command

The **set_cond_loc** command takes a single parameter that controls the conditional test return value for requests either from the PE requests or the DPP. If the location is set to the DPP, the CO always returns true to the PE and performs the conditional test for any DPP request. Conversely, if the location is set to the PE, the CO always returns true to the DPP and performs the conditional test for any PE request. The location is maintained until the next **set_cond_loc** command is received.

Parameter 0: loc

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
																															l

Bits	Field Name	Description
0	loc	**Sets the location of the conditional test:** • 0 - DPP • 1 -- PE
31:1	Reserved	Reserved

Chapter 3
DPP Array Instruction Set Architecture

This chapter describes the functional behavior of the ATI X1K Fragment Processor (X1K FP). This is intended for low level-programmers who need to know what the hardware does, but not how it does it.

3.1 Instructions

There are 512 instruction slots. In the absence of flow control, programs will increment the program counter after each instruction.

Each instruction can be one of four types:

- INST_TYPE_ALU - Arithmetic and Logic Unit instruction
- INST_TYPE_OUTPUT - Output instruction (with ALU functionality)
- INST_TYPE_FC - Flow Control instruction
- INST_TYPE_TEX - Texture instruction

ALU and OUTPUT instructions both have full RGB and Alpha math functionality. The only functional difference between them is that ALU instructions can set the predicate bits, and OUTPUT instructions can write to the output registers. There is no way to do both in the same instruction. Internally, the sequencer must treat instructions that have potential outputs specially for scheduling. The last executed instruction of the X1K FP program must also be an OUTPUT instruction, even if it's not outputting anything interesting.

The first OUTPUT instruction will reserve space in the output register fifo. This space is limited, therefore issuing an OUTPUT earlier than necessary may cause threads to stall earlier than necessary. You should not set an ALU instruction as type OUTPUT unless it is actually writing to an output register, or it is the last instruction of the program.

Flow control instructions and texture instructions each have their own interpretation of the bits in the instruction word.

The last instruction must be an OUTPUT instruction (even if the output mask is zero), and should always wait for the texture unit semaphore by setting the TEX_SEM_WAIT bit (see below). At the time of termination, the contents of the output registers are sent to the render targets. Note that uncached writes (UMRT) occur immediately and do not wait for program termination.

Updates to X1K FP code outside the currently active program are safe, and do not stall the pipeline. If you intend to overwrite the active program, however, the pipe must be flushed so that fragments running the old program get out before the update.

3.2 Instruction Words

INST_TYPE_ALU / INST_TYPE_OUTPUT (6 words):

- CMN_INST_*
- ALU_RGB_ADDR_*
- ALU_ALPHA_ADDR_*
- ALU_RGB_INST_*
- ALU_ALPHA_INST_*
- ALU_RGBA_INST_*

INST_TYPE_FC (3 words):

- CMN_INST_*
- FC_INST_*
- FC_ADDR_*

INST_TYPE_TEX (4 words):

- CMN_INST_*
- TEX_INST_*
- TEX_ADDR_*
- TEX_ADDR_DXDY_*

The FC and TEX words overlap with the ALU/OUTPUT words in instruction memory. The unused memory locations for FC and TEX are ignored by X1K FP; they may be left uninitialized, or set to zero, with no ill effect.

Within CMN_INST_*, the fields effective for each instruction type are indicated by Xs:

	ALU	OUTPUT	FC	TEX
TYPE	X	X	X	X
TEX_SEM_WAIT	X	X	X	X
RGB_PRED_SEL	X	X	X	X
RGB_PRED_INV	X	X	X	X
ALPHA_PRED_SEL	X	X		X
ALPHA_PRED_INV	X	X		X
WRITE_INACTIVE	X	X		X
LAST	X	X		
NOP	X	X		
RGB_WMASK	X	X		X
ALPHA_WMASK	X	X		X
RGB_OMASK	X	X		
ALPHA_OMASK	X	X		
RGB_CLAMP	X	X		
ALPHA_CLAMP	X	X		
ALU_RESULT_SEL	X	X		
ALU_RESULT_OP	X	X		
ALU_WAIT			X	X

3.2.1 Synchronization of instruction streams

X1K FP allows you to freely intermix instructions of multiple types. It will process the three types (ALU/Output, Texture, and FC) in parallel whenever possible. Instructions need to be synchronized when an instruction of one type depends on the output of another type. The cases where explicit synchronization may be required are:

- TEX instruction dependent on ALU for source register or predicate. Synchronized with the ALU_WAIT bit.
- FC instruction dependent on ALU for predicate or ALU result. Synchronized with the ALU_WAIT bit.

- ALU instruction dependent on TEX for lookup result. Synchronized using the texture semaphore.

A texture or FC instruction that uses a result computed by a prior ALU instruction should set the ALU_WAIT bit. This forces processing for the thread to stall until pending ALU instructions are complete.

Note that a static FC instruction never needs to set ALU_WAIT, since it never depends on a result computed within the program. Also, an ALU instruction never needs to set ALU_WAIT—dependencies amongst ALU instructions are resolved internally.

The texture semaphore is used to synchronize the output of a texture instruction with a subsequent ALU or texture instruction that uses that result. It is also used to synchronize uncached writes. The texture semaphore is described in more detail below.

3.3 ALU Instructions

An ALU instruction actually consists of an RGB vector instruction and an Alpha scalar instruction.

There are only a few operations that only one or the other unit can compute, but in each case there is a special instruction the other engine can use to copy the result.

3.3.1 Sources

Each instruction can specify the addresses for 6 different sources -- 3 RGB vectors and 3 Alpha scalars. Each source can either come from one of 128 temporary registers (which can be modified during the program, and be different for each processor), or from one of 256 constant registers (which can only be changed when a program is not running). In addition, a source can be an inline constant. The loop variable (aL) may be added to any combination of source addresses, but may not be added to an inline constant.

Each color register (temporary and constant) consists of a 3-component RGB vector and a scalar Alpha value.

Inline constants are unsigned floating-point values with 4 bits of exponent (with bias 7) and 3 bits mantissa. Inline constants represent finite values only—there is no representation for NaN or infinity. Inline constants can express denormal values though. Also, the bit pattern 0x0 represents 2^-10, rather than zero. Example values are shown below:

	EXPONENT	MANTISSA
2^-10	0x0	0x0
2^-9	0x0	0x1
2^-8	0x0	0x2
2^-7	0x0	0x4
2^-6	0x1	0x0
1	0x7	0x0
256	0xf	0x0
480	0xf	0x7

You can obtain negative inline constants and the value zero using the input modifiers and swizzles, described below.

Each source is specified with three fields. Valid encodings of these fields are shown below (for source 0, in this example):

	ADDR0[7]	ADDR0[6:0]	ADDR0_CONST	ADDR0_REL
register N	**0**	**N**	**0**	**0**
register N + aL	**0**	**N**	**0**	**1**
constant N	**N / 128**	**N % 128**	**1**	**0**
constant N + aL	**N / 128**	**N % 128**	**1**	**1**
inline const X	**1**	**X**	**0**	**0**

Note that inline constants set the MSB of ADDR0 and clear ADDR0_CONST.

3.3.2 Presubtract

Each RGB and Alpha instruction has a presubtract operation, which does some extra math on incoming data from the first or from the first and second sources. The available operations are:

- SRCP_OP_BIAS - 1-2*src0
- SRCP_OP_SUB - src1-src0
- SRCP_OP_ADD - src1+src0
- SRCP_OP_INV - 1-src0

The RGB presubtract happens on all three components in parallel. The Alpha presubtract is scalar.

If any presubtract result is used in the instruction, and one of the sources being used in a presubtract is written in the previous instruction, and the previous instruction is an ALU or output instruction, a NOP needs to be inserted between the two instructions. Do this by setting the NOP flag in the previous instruction, so the NOP does not consume an instruction slot. This allows the HW the extra cycle necessary to resolve the dependencies involved in doing this extra math (there are additional cases where NOP may need to be set, noted below).

Note: In these cases, the assembler could hide it.

NOP is never required if the previous instruction is a texture lookup.

3.3.3 Inputs

Each math operation has zero to three inputs. Each input can be configured to select a source and swizzle its channels. There are fields to configure 6 inputs per instruction: 3 for RGB and 3 for Alpha. An instruction can read in at most 12 independent colour components (9 RGB components and 3 alpha components).

Select

Each input selects from src0, src1, src2, or the presubtract result ("srcp"). One can conceive of the selects assembling a 4-component vector as seen below. The swizzle selects (see next section) determine which of the four values are chosen to actually take part in the computations.

Input Select/ Presubtract	4-component Vector
src0 =	{ rgb_addr0->r { rgb_addr0->g { rgb_addr0->b { alpha_addr0->a
src1 =	{ rgb_addr1->r { rgb_addr1->g { rgb_addr1->b { alpha_addr1->a
src2 =	{ rgb_addr2->r { rgb_addr2->g { rgb_addr2->b { alpha_addr2->a
srcp =	{ rgb_srcp_result.r = rgb_srcp_op(rgb_addr0->r, rgb_addr1->r) { rgb_srcp_result.g = rgb_srcp_op(rgb_addr0->g, rgb_addr1->g) { rgb_srcp_result.b = rgb_srcp_op(rgb_addr0->b, rgb_addr1->b) { alpha_srcp_result.a = alpha_srcp_op(alpha_addr0->a, alpha_addr1->a)

The RGB and alpha units each take three operands, A, B, and C. These operands are selected with the RGB_SEL_x and ALPHA_SEL_x fields. Note that src0, src1 and src2 are fetched from a combination of the RGB and alpha source addresses. If the RGB unit swizzles in an alpha component, the alpha component will always come from alpha_addr*. Similarly, if the alpha unit swizzles in an RGB component, it will always come from rgb_addr*.

Swizzle

Each component of each input can specify one of seven values. Each component can select R, G, B, or A from the selected source, or it can choose 0, 0.5, or 1. The RGB unit has 3 components, so there are three swizzle select fields per input. The Alpha unit only has 1 swizzle select per input.

The RGB unit always uses the RGB selectors (RGB_SEL_x) and, except for one case noted below, the red (RED_SWIZ_x), green (GREEN_SWIZ_x), and blue (BLUE_SWIZ_x) swizzle selects. The alpha unit always uses the alpha selectors (ALPHA_SEL_x) and the alpha (ALPHA_SWIZ_x) swizzle selects.

DP4 is a special case in that it is an RGB operation which operates on 4 components instead of 3. The fourth input component is configured with the Alpha's select (ALPHA_SEL_x) and swizzle (ALPHA_SWIZ_x). This is the only case where the Alpha's swizzle has an effect on the RGB computation's input.

Input Modifier

Each input has a modifier applied to it. The modifier can be one of:

- IMOD_OFF - No modification
- IMOD_NEG - Negate
- IMOD_ABS - Take absolute value
- IMOD_NAB - Take negative of absolute value

3.3.4 The Operation

Following are the possible math operations the ALU can perform. The three inputs are denoted by A, B, and C.

OP_RGB_SOP / OP_ALPHA_DP

- Get results from the other unit's unique ops. In the case of RGB_SOP, the result is replicated to all three channels. RGB's unique ops all have scalar results, so ALPHA_DP simply copies that scalar result to its alpha destination.
- RGB_SOP is only valid if the alpha operation is a transcendental operation: EX2, LN2, RCP, RSQ, SIN, COS. ALPHA_DP is only valid if the RGB operation is a dot product: DP3, DP4, D2A.

OP_RGB_MAD / OP_ALPHA_MAD

- `A * B + C`

OP_RGB_MIN / OP_ALPHA_MIN

- `A < B ? A : B`
- Minimum of A and B.

OP_RGB_MAX / OP_ALPHA_MAX

- `A >= B ? A : B`
- Maximum of A and B.

OP_RGB_CND / OP_ALPHA_CND

- `C > 0.5 ? A : B`

OP_RGB_CMP / OP_ALPHA_CMP

- `C >= 0 ? A : B`

OP_RGB_FRC / OP_ALPHA_FRC

- `A - floor(A)`
- floor(A) is the largest integer value less than or equal to A.

OP_RGB_DP3

- `A.r*B.r + A.g*B.g + A.b*B.b`
- Results are broadcast to all 3 channels.
- Use OP_ALPHA_DP to get result into Alpha.

OP_RGB_DP4

- `A.r*B.r + A.g*B.g + A.b*B.b + A.a*B.a`
- Results are broadcast to all 3 channels.
- Use OP_ALPHA_DP to get result into Alpha.
- Note that ".a" actually comes from the alpha instruction's swizzle and select (see the section on swizzle above).

OP_RGB_D2A

- `A.r*B.r + A.g*B.g + C.b`
- Results are broadcast to all 3 channels.
- Use OP_ALPHA_DP to get result into Alpha.

OP_ALPHA_EX2

- `2 ^ A`
- Use OP_RGB_SOP to get result into RGB.

OP_ALPHA_LN2

- `log2(A)`
- Use OP_RGB_SOP to get result into RGB.

OP_ALPHA_RCP

- `1 / A`
- Use OP_RGB_SOP to get result into RGB.

OP_ALPHA_RSQ

- `1 / squareRoot(A)`
- Use OP_RGB_SOP to get result into RGB.

Note: The SM3 specification defines reciprocal square root as `1 / squareRoot(abs(A))`—this can be achieved by using the input modifier for A.

OP_ALPHA_SIN

- `sin(A * 2pi)`
- Use OP_RGB_SOP to get result into RGB.

OP_ALPHA_COS

- `cos(A * 2pi)`
- Use OP_RGB_SOP to get result into RGB.

3.3.5 Instruction Modifiers

Each instruction can have an output modifier applied to its result:

- OMOD_U1 - Multiply by 1
- OMOD_U2 - Multiply by 2
- OMOD_U4 - Multiply by 4
- OMOD_U8 - Multiply by 8
- OMOD_D2 - Divide by 2
- OMOD_D4 - Divide by 4
- OMOD_D8 - Divide by 8
- OMOD_DISABLED - No modification

Each instruction can also be optionally clamped to the range 0 to 1. This happens after the above output modifier.

Disabling the Output Modifier

The multiply/divide output modifiers all convert NaN values into a standardized NaN (0x7fffffff) and squash any denormal values to plus or minus zero. For most ALU operations this is acceptable, however a MOV instruction needs to preserve the source exactly. For this, you can disable the output modifier for the MIN, MAX, CMP and CND instructions. With OMOD_DISABLED, the result is not modified at all; the value is neither multipled nor divided, and clamping is not applied.

This allows a MOV to be implemented using any of the following instructions, with OMOD_DISABLED set:

- MIN(src, src)
- MAX(src, src)
- CND(src, src, 0)
- CMP(src, src, 0)

OMOD_DISABLED is not valid with any other ALU operation.

3.3.6 Writemasks

There are a number of writemasks for each instruction:

Writemask	Size	Description
RGB_WMASK	3 bits	Write R,G,B to register destination.
ALPHA_WMASK	1 bit	Write A to register destination.
RGB_OMASK	3 bits	Write R,G,B to output or to predicate bits.
ALPHA_OMASK	1 bit	Write A to output or to predicate bits.
W_OMASK	1 bit	Write A to W output.
WRITE_INACTIVE	1 bit	If set, ignores flow control processor mask when writing. Affects ALU and texture instructions. If in doubt, this bit should be cleared.
RGB_PRED_SEL	3 bits	Sets one of six modes that specify which of the 4 predicate bit(s) to AND with the RGB writemask (and output mask when applicable). One of: • NONE - no predication • RGBA - normal predication • RRRR - replicate R predicate bit • GGGG - replicate G predicate bit • BBBB - replicate B predicate bit • AAAA - replicate A predicate bit
RGB_PRED_INV	1 bit	Inverts selected RGB predicate bit(s). Should be zero if RGB_PRED_SEL is set to NONE.
ALPHA_PRED_SEL	3 bits	Like RGB_PRED_SEL, but used to control predication for the alpha unit's write mask.
ALPHA_PRED_INV	1 bit	Inverts selected alpha unit predicate bit. Should be zero if ALPHA_PRED_SEL is set to NONE.
IGNORE_UNCOVERED	1 bit	If set, excludes uncovered processors (outside triangle or killed via TEXKILL) from TEX lookups and flow control decisions. Affects texture and flow control instructions. If in doubt, this bit should be cleared.
ALU_WMASK	1 bit	If set, update the ALU result. Similar to the predicate write mask.

Flow control instructions only have one predicate select, using the RGB_PRED_SEL and RGB_PRED_INV fields. ALU/Output instructions can use different predicate selects for the RGB (vector) computation and the alpha (scalar) computation. For texture instructions, the RGB results from the texture unit will be influenced by RGB_PRED_SEL/ RGB_PRED_INV, and the alpha result from the texture unit will be influenced by the ALPHA_PRED_SEL/ ALPHA_PRED_INV fields.

3.3.7 Destination

The destination address refers to a temporary register. The loop variable (aL) may optionally be added to the address before writing. The predicate select in RGB_PRED_SEL, RGB_PRED_INV, ALPHA_PRED_SEL, and ALPHA_PRED_INV will be applied when writing to the destination.

3.3.8 Output

With OUTPUT instructions, the TARGET field indicates where the result of the instruction should be written. When in cached write mode (the default mode), the following options are available:

- RNDR_TGT_A - Write to render target A register
- RNDR_TGT_B - Write to render target B register
- RNDR_TGT_C - Write to render target C register

- RNDR_TGT_D - Write to render target D register

The OUT_FMT_* registers describe render targets A through D. The results are stored and the final value is sent out when the program terminates. If a channel in an output target is written more than once, the final value written is what will be sent out. The RGB and alpha unit may write to different targets in the same instruction.

The output may be predicated using PRED_SEL and PRED_INV.

You may also perform uncached output (or UMRT writes), which are sent out immediately, but you cannot perform cached and uncached writes in the same program. The selection of the kind of program (a program that uses cached writes or one that uses uncached writes) is selected by CTM based on information supplied with the program itself.

An output instruction in a UMRT program must either write all channels, or no channels. The green and blue channels will be interpreted as the (X,Y) coordinates, the alpha channel must be zero, and the red channel contains the 32-bit floating-point value to write. A UMRT program writes only one 32-bit value per output instruction.

The type of UMRT write performed is controlled by the RGB TARGET field (the alpha unit's TARGET field is ignored). To perform a UMRT write, set the RGB TARGET field to one of the following values:

- RNDR_TGT_UMRT_SAME_IDX - Write using a shared index for all processors.
- RNDR_TGT_UMRT_SAME_IDX_TEX_SEM_ACQUIRE - and acquire texture semaphore to synchronize reads.
- RNDR_TGT_UMRT_PER_PROCESSOR_IDX - Write using distinct index for each processor (at 1/4 speed).
- RNDR_TGT_UMRT_PER_PROCESSOR_IDX_TEX_SEM_ACQUIRE - and acquire texture semaphore to synchronize reads.

If the program is reading back results written with uncached writes, then the program should use one of the TEX_SEM_ACQUIRE choices to synchronize uncached writes and reads. Additional information on the texture semaphore is given in the texture instruction overview, below.

3.3.9 Setting Predicate Bits

Each instruction may optionally set one or more predicate bits. ALU instructions (as opposed to OUTPUT instructions) interpret the OMASK fields as a predicate writemask. The TARGET field determines when to set the bits associated with each channel:

- PRED_OP_EQUAL - Set when channel is zero.
- PRED_OP_LESS - Set when channel is negative.
- PRED_OP_GREATER_EQUAL - Set when channel is non-negative.
- PRED_OP_NOT_EQUAL - Set when channel is non-zero.

The enumeration's names are based on the assumption that they will be primarily used after a subtraction of two values. That's not the only possible use, of course. The RGB and alpha units may use different functions to set the predicate in the same instruction.

In order to achieve the remaining common comparisons, <= and >, one can simply reverse the order of the values being subtracted, or reverse both signs, and use the >= and < operations respectively.

You can simultaneously write to the predicate register and a temporary register, and you can perform a predicated temporary register write if you are also writing the predicate register. However, the old value of the predicate will only be applied to the temporary register's write mask; it will not be applied to the predicate write mask. In other words, if the predicate is 0x7, your temporary write mask is 0xf and your predicate write mask is 0xf, you will write only RGB components to the temporary register, but you will write to all 4 predicate bits.

If the instruction result is clamped, the comparison happens on the post-clamped result. If output modifier is disabled, denormals may be compared -- denormals are equivalent to zero.

3.3.10 ALU Result

Every instruction has an "ALU result." In order to use it, an ALU instruction must write an ALU result, and it must be consumed by the next flow control instruction. The ALU result is preserved across other ALU/texture instructions that do not write a new ALU result, but is NOT preserved across flow control instructions; therefore the ALU result must be consumed by the first flow control statement after it is written.

The ALU result is a single bit. The channel source for the ALU result is selected by the ALU_RESULT_SEL field:

- ALU_RESULT_SEL_RED
- ALU_RESULT_SEL_ALPHA

How to interpret the floating point result to set the ALU result bit is specified by the ALU_RESULT_OP field, which is similar to the interpretation of the TARGET field for setting the predicate bits:

- ALU_RESULT_OP_EQUAL - Set when channel is zero
- ALU_RESULT_OP_LESS - Set when channel is negative
- ALU_RESULT_OP_GREATER_EQUAL - Set when channel is non-negative
- ALU_RESULT_OP_NOT_EQUAL - Set when channel is non-zero

The ALU instruction that updates the ALU result must set the ALU_WMASK bit.

If the instruction result is clamped, the comparison happens on the post-clamped result. If output modifier is disabled, denormals may be compared—denormals are equivalent to zero.

3.4 Texture Instructions

Texture instructions are simpler than ALU or flow control instructions. Texture instructions have one destination temporary address, 1 to 3 source temporary addresses, a sampler ID, and an opcode and control bits specifying how to lookup the texture.

As with ALU temporary addresses, the loop variable (aL) may be added to any texture temporary address (source and destination). Texture source addresses allow arbitrary swizzles from RGBA to STRQ coordinate space, and the RGBA result from the texture unit may also be swizzled. Unlike with ALU instructions, the texture swizzles cannot be used to select constant inputs (0, 0.5, 1). Texture source addresses always read from the temporary registers; they cannot read from the constant bank.

Texture instructions feature a texture semaphore mechanism to synchronize texture lookup with instructions using the result of the lookup. See below for more information.

You may choose to limit which channels of a texture lookup are written by using the write masks RGB_WMASK and ALPHA_WMASK. These write masks may be predicated; the RGB results from the texture unit are predicated with RGB_PRED_SEL and RGB_PRED_INV, while the alpha result from the texture unit is predicated with ALPHA_PRED_SEL and ALPHA_PRED_INV.

Texture instructions have an UNSCALED bit that control whether the texture coordinates are scaled by the texture dimensions before lookup. In typical usage, this bit is cleared for normal texture lookups which supply coordinates in the range [0.0, 1.0], and set for texture lookups which supply coordinates that are prescaled to the texture dimensions. Uncached reads (used for UMRT) should set the UNSCALED bit: the coordinates supplied to the program are prescaled, and the index coordinate should always be an integer value.

3.4.1 Operations

There are five texture operations available.

Texture Option	Description
TEX_INST_NOP	Perform no operation. The source addresses are ignored, and nothing is written to the destination address. A texture NOP may acquire the texture semaphore, so NOP can be used for synchronization purposes.
TEX_INST_LOOKUP	A standard texture lookup. Reads the coordinates from SRC_ADDR and writes the results of the lookup to DST_ADDR.
TEX_INST_KILL_LT_0	Kill the processor if any components in SRC_ADDR are less than zero. Note that the source swizzles are ignored in this case; if you want to limit which channels are examined, you may use the write masks in WMASK_RGB, WMASK_ALPHA, and/or predication. Nothing is written to the destination address.
TEX_INST_LOOKUP_PROJ	Lookup a projected texture. Q is used for the projective divide.
TEX_INST_LOOKUP_UNCACHED	Perform an uncached read. This mechanism is used to retrieve data that was written using uncached writes, either in this program or by a previous program. If reading data written by this program, the texture semaphore mechanism must be used on the uncached write instructions. If reading data written by a previous program, it may be necessary to flush the pipe. Typically the UNSCALED bit should be set for an uncached read.

3.4.2 Semaphore

The semaphore is used to synchronize texture lookups with their subsequent use in the program. It also serves a second purpose in uncached write mode, where it synchronizes uncached writes with subsequent uncached reads through the texture unit.

Each texture instruction has a bit, TEX_SEM_ACQUIRE, specifying whether it should hold the texture semaphore until the looked-up data comes back and is written to the destination temporary register. All program instructions have another semaphore bit, TEX_SEM_WAIT, that specifies whether to wait on the semaphore so its (dependent) source data is up to date. You may take advantage of the texture semaphore to perform various independent computations while waiting on the texture operation to complete.

Hardware disallows more than one ACQUIRE operation at a time, so if you set TEX_SEM_ACQUIRE on a lookup or uncached write, you must also set TEX_SEM_WAIT for that instruction. WAIT has no cost if there are no outstanding ACQUIRE operations. For an instruction with TEX_SEM_WAIT and TEX_SEM_ACQUIRE both set, the wait happens first.

There is only one texture semaphore, however you may use it to protect multiple texture lookups, as long as the lookups are themselves independent. When a texture instruction sets TEX_SEM_ACQUIRE, the texture unit ensures that that particular lookup, and all prior lookups, have completed before releasing the semaphore. Therefore, to protect several texture lookups, you may set TEX_SEM_ACQUIRE only on the last texture lookup, and set TEX_SEM_WAIT on the first instruction that uses any of the results. This example illustrates the usage:

	INSTRUCTION	TEX_SEM_WAIT	TEX_SEM_ACQUIRE
0:	r4 = TEXLD(s0, r1)	0	0
1:	r5 = TEXLD(s1, r2)	0	0
2:	r6 = TEXLD(s2, r3)	1	1
3:	r1 = r1 + 1	0	

	INSTRUCTION	TEX_SEM_WAIT	TEX_SEM_ACQUIRE
4:	r2 = r2 + 1	0	
5:	r3 = r3 + 1	0	
6:	r4 = r4 + 1	1	

In the above example, note that instruction 2 waits for the semaphore to ensure the semaphore is available before acquiring it.

Using TEX_SEM_ACQUIRE on an uncached write instruction is very similar to using TEX_SEM_ACQUIRE on a texture lookup. If you are performing uncached reads and writes in the same program, you should use one of the TEX_SEM_ACQUIRE options on the last uncached write before an uncached read, and you should use TEX_SEM_WAIT on the uncached read. The uncached write mechanism will not release the semaphore until that particular uncached write, and all prior uncached writes, are complete.

Remember that the last instruction of the program must set TEX_SEM_WAIT, to ensure that the texture unit is ready to process the next fragment. It is invalid to terminate the program while holding the texture semaphore, either from a texture lookup or from an uncached write.

Warning: see errata section at the end for a known issue with texture semaphores.

3.5 Flow Control

Each flow control instruction is essentially a conditional jump. Various optional stack operations allow all the different kinds of traditional flow control statements. In particular, flow control instructions allow branch statements (if/else/endif blocks), loop statements (with an optional loop register, aL), and subroutine calls. Optimizers may be able to combine these basic types of instructions, and utilize more esoteric flow control modes.

HW supports two flow control modes, "partial" and "full". Partial flow control mode enables twice as many contexts as full mode, but partial flow control mode has a limited nesting depth of branch statements, and does not support loops or subroutine calls. Partial flow control mode should be used unless the program requires branch statements nested more than 6 deep, or the program requires loops or subroutines.

In CTM, partial or full flow control mode is selected based on information supplied as part of the program itself. The application has no control over the selection (except via the program it loads).

See the Fields section below for descriptions of fields that affect the jump condition and the various flow control stacks. Following that are the values of those fields for the most common types of flow control operations.

3.5.1 Dynamic Flow Control

As the X1K DPP is a SIMD engine, applying the same instruction to a group of processors, dynamic flow control must be implemented with processor masks. If a processor wants to take a jump because it failed an IF condition, but its neighbors in the processor group don't want to jump, the processor must be masked off for a time until that branch of the IF statement is completed. Only if all processors fail the IF condition would the program counter actually be changed. Conversely, if some processors don't want to jump to a subroutine, they must be masked off for the entire subroutine. Only if none of the processors want to jump would the call be skipped. A break statement within a loop masks off passing processors until the loop is complete, and the program counter is only changed if all processors want to jump.

These processor masks are organized into stacks so flow control blocks may be nested. The operations on these stacks are encoded in the flow control instructions as flags, instead of having one set of opcodes which hard-wire the stack behavior. This orthogonality allows for more creative control of the program behavior, and provides opportunity for optimizations in programs that use a lot of flow control.

Jump conditions can be based off of a boolean constant, the result of the previous ALU operation, and/or a predicate bit. Booleans are constant across all processors, so dynamic flow control is only achieved with predicates and conditionals (ALU result). Any ALU instruction can specify whether to write the ALU result and what channel supplies the data for the result. The ALU result is only valid until another ALU instruction writes to the result, or a flow control instruction is encountered. The predicate bits can be set anywhere and are preserved across flow control instructions, but there are only 4 of them.

Flow control predication cannot be per-channel. One of the replicate swizzles must be used for predication of flow control instructions (all other types of instructions can be predicated per channel). Flow control instructions use the RGB_PRED_SEL and RGB_PRED_INV fields to compute the predicate.

3.5.2 Stacks and Branch Counters

The HW maintains two separate stacks for flow control:

- **Address Stack** - Purely an address stack. No other state is maintained. Popping the address stack overrides the instruction address field of the flow control instruction. The address stack will only be modified if the flow control instruction decides to jump.
- **Loop Stack** - Stores an internal iteration count, loop variable (aL), and a processor mask per frame. The only way to access the iteration count is with the LOOP/ENDLOOP and REP/ENDREP operations. The only way to alter the aL variable is with the LOOP/ENDLOOP ops. The only way to read the aL variable is with relative addressing. The only way to alter the processor mask is with the BREAK or CONTINUE instruction.

Each stack's size is dependent on whether the program is in partial or full flow control mode. Stack overflows and underflows produce undefined behaviour in the hardware. The stack sizes are:

	PARTIAL	FULL
Loop stack	**n/a**	**4**
Address stack	**n/a**	**4**

The loop stack is maintained in such a way that an inner REP block will continue to see the loop variable from an outer LOOP block. Nested LOOP blocks will shadow the loop variable. The loop variable is not valid if you are not in at least one LOOP block.

In addition to the two stacks, hardware maintains an Active Bit and a Branch Counter for each processor that indicate whether the processor is active and, if it was disabled by a conditional statement (if, else), how long before it can be reactivated. If the active bit is unset, the processor is inactive and the branch counter indicates the number of conditional blocks we must exit before the processor can be activated again. The maximum value of this counter is dependent on whether the program is in partial or full flow control mode. The limits (which determine maximum safe nesting depth) are:

	PARTIAL	FULL
Branch counter	**0..3**	**0..31**
Maximum depth	**4**	**32**

The branch counter can be incremented and decremented directly by any flow control instruction based on whether the processor agrees with the jump decision. Manipulating the branch counter may affect the active bit. Incrementing the counter on an active processor will disable the processor by clearing the active bit, and set the branch counter to zero. Decrementing the counter of an inactive processor to a negative value will set the active bit, reactivating the processor. The branch counter is ignored in hardware while the active bit is set.

Processors disabled by looping statements (BREAKLOOP, BREAKREP, and CONTINUE) are also tracked with "loop inactive" counters, however unlike the branch counter, the loop counters cannot be manipulated directly.

Since only conditional (if, else) and loop statements maintain active processor masks, to call a function based on a condition requires the program to use the branch counters on CALL and RETURN so the processor active mask will be updated on the conditional call. If you know ahead of time that *all* calls to a particular subroutine will be unconditional calls, you can omit the branch counter manipulation on that subroutine's return and on any calls to that subroutine. The benefit of this is unclear, unless you are nearing the upper limit on the branch counter.

Returns within dynamic branches and/or loops (nested in the subroutine) are not supported. A return can be made conditional (by incremeneting the branch stack counter on stay), but the hardware does not support returning within other conditional blocks that might partially mask it. If a branch is entirely static (based on a constant boolean), you may put a return within a branch (just get the branch counter decrement right). This cannot be done inside loops, however.

3.5.3 Fields

Fields Controlling Conditions on the Jump

JUMP_FUNC - 2x2x2 table indicating when to jump.

- Bit 0 = Jump when (!alu_result && !predicate && !boolean).
- Bit 1 = Jump when (!alu_result && !predicate && boolean).
- Bit 2 = Jump when (!alu_result && predicate && !boolean).
- Bit 3 = Jump when (!alu_result && predicate && boolean).
- Bit 4 = Jump when (alu_result && !predicate && !boolean).
- Bit 5 = Jump when (alu_result && !predicate && boolean).
- Bit 6 = Jump when (alu_result && predicate && !boolean).
- Bit 7 = Jump when (alu_result && predicate && boolean).

Common JUMP_FUNC values:

- 0x00 = Never jump
- 0x0f = Jump iff alu_result is false.
- 0x33 = Jump iff predicate is false.
- 0x55 = Jump iff boolean is false.
- 0xaa = Jump iff boolean is true.
- 0xcc = Jump iff predicate is true.
- 0xf0 = Jump iff alu_result is true.
- 0xff = Always jump

JUMP_ANY - How to treat partially passing groups of processors.

- false = Don't jump unless all processors want to jump
- true = Jump if at least one active processor wants to jump

When JUMP_ANY is false, the instruction behaves like a universal quantifier, and will decide to jump if there are no active processors. When JUMP_ANY is true, the instruction behaves like an existential quantifier, and will never decide to jump if there are no active processors. Looping statements may override the jump decision made by the processors based on the loop counter.

Fields Controlling Optional Stack Operation

OP - Loop Stack Operations.

- FC_OP_JUMP = None
- FC_OP_LOOP = Initialize counter and aL, and push loop stack if stay

- FC_OP_ENDLOOP = Increment counter and aL if jump, else pop loop stack
- FC_OP_REP = Initialize counter, and push loop stack if stay
- FC_OP_ENDREP = Increment counter if jump, else pop loop stack
- FC_OP_BREAKLOOP = Pop loop stack if jump
- FC_OP_BREAKREP = Pop loop stack if jump
- FC_OP_CONTINUE = Disable processors until end of current loop

You should use FC_OP_BREAKLOOP if the innermost looping construct is LOOP, and FC_OP_BREAKREP if the innermost looping construct is REP.

A_OP - Address Stack Operations.

- FC_A_OP_NONE = None
- FC_A_OP_POP = Pop address stack if jump (overrides JUMP_ADDR given in instruction)
- FC_A_OP_PUSH = Push address stack if jump

B_OP0 - Branch stack Operations if stay.

- FC_B_OP_NONE = None
- FC_B_OP_DECR = Decrement branch counter for inactive processors by amount in B_POP_CNT. Activate processors which go negative.
- FC_B_OP_INCR = Increment branch counter for inactive processors by 1. Deactivate processors which disagree with the jump decision (by deciding to jump) and set their branch counter to 0.

B_OP1 - Branch stack Operations if jump.

- FC_B_OP_NONE = None
- FC_B_OP_DECR = Decrement branch counter for inactive processors by amount in B_POP_CNT. Activate processors which go negative.
- FC_B_OP_INCR = Increment branch counter for inactive processors by 1. Deactivate processors which disagree with the jump decision (by deciding not to jump) and set their branch counter to 0.

B_POP_CNT - Branch Stack Pop Count.

How much to decrement the branch counters by when appropriate B_OP* field says to decrement.

B_ELSE - Branch Stack Else

- false = None
- true = Activate processors whose branch count is zero (processors deactivated by the innermost cond itional block), and deactivate all processors that were active.

Special Cases

When the iteration count is zero, LOOP/REP ignore JUMP_FUNC and jump.

When the iteration count is zero, ENDLOOP/ENDREP ignore JUMP_FUNC and don't jump.

Any processors deactivated by B_ELSE "want to jump" regardless of JUMP_FUNC.

Any processors deactivated by a branching statement (if, else) will inhibit a decision to jump by a BREAK or CONTINUE statement.

Any processors deactivated by a CONTINUE statement will inhibit a decision to jump by a BREAK statement; they will not inhibit a decision to jump by another CONTINUE statement.

Processors deactivated by other flow control are indifferent to the decision to jump by a BREAK or CONTINUE statement.

Address Fields

BOOL_ADDR - Which of 32 constant booleans to use for jump condition.

INT_ADDR - Which of 32 constant integers to use for loop initialization (the red channel is used for iteration count, green for aL initialization, and blue for aL increment).

JUMP_ADDR - Which instruction to jump to if conditions pass.

JUMP_GLOBAL -- Whether JUMP_ADDR is global, or if OFFSET_ADDR should be added to JUMP_ADDR.

3.5.4 Common Flow Control Statements

```
          JUMP_FUNC          A_OP                    B_POP_CNT
          |      JUMP_ANY    |       B_OP0           |   B_ELSE
          |      |   OP       |       |       B_OP1   |   |   JUMP_ADDR
          |      |   |        |       |       |       |   |   |

IF b      0x55   0   JUMP     NONE    NONE    NONE    0   0   ELSE+1
ELSE      0xff   0   JUMP     NONE    NONE    NONE    0   0   ENDIF
ENDIF

IF p      0x33   0   JUMP     NONE    INCR    INCR    0   0   ELSE+1
ELSE      0x00   0   JUMP     NONE    NONE    DECR    1   1   ENDIF+1
ENDIF     0x00   1   JUMP     NONE    DECR    NONE    1   0   0

IF c      0x0f   0   JUMP     NONE    INCR    INCR    0   0   ELSE+1
ELSE      0x00   0   JUMP     NONE    NONE    DECR    1   1   ENDIF+1
ENDIF     0x00   1   JUMP     NONE    DECR    NONE    1   0   0

IF b      0x55   0   JUMP     NONE    NONE    NONE    0   0   ENDIF
ENDIF

IF p      0x33   0   JUMP     NONE    INCR    NONE    0   0   ENDIF+1
ENDIF     0x00   1   JUMP     NONE    DECR    NONE    1   0   0

IF c      0x0f   0   JUMP     NONE    INCR    NONE    0   0   ENDIF+1
ENDIF     0x00   1   JUMP     NONE    DECR    NONE    1   0   0

LOOP      0x00   0   LOOP     NONE    NONE    NONE    0   0   ENDLOOP+1
ENDLOOP   0xff   1   ENDLOOP  NONE    NONE    NONE    0   0   LOOP+1

REP       0x00   0   REP      NONE    NONE    NONE    0   0   ENDREP+1
ENDREP    0xff   1   ENDREP   NONE    NONE    NONE    0   0   REP+1

BREAK     0xff   0   BREAK    NONE    NONE    DECR    n   0   END+1
BREAK b   0xaa   0   BREAK    NONE    NONE    DECR    n   0   END+1
BREAK p   0xcc   0   BREAK    NONE    NONE    DECR    n   0   END+1
BREAK c   0xf0   0   BREAK    NONE    NONE    DECR    n   0   END+1

CONTINUE  0xff   0   CONTINUE NONE    NONE    DECR    n   0   END

CALL      0xff   1   JUMP     PUSH    NONE    INCR    0   0   Subroutine
CALL b    0xaa   1   JUMP     PUSH    NONE    INCR    0   0   Subroutine
CALL p    0xcc   1   JUMP     PUSH    NONE    INCR    0   0   Subroutine
CALL c    0xf0   1   JUMP     PUSH    NONE    INCR    0   0   Subroutine
RETURN    0xff   0   JUMP     POP     NONE    DECR    1   0   0
```

- n indicates how many branch stack frames the BREAK is inside within the current loop.
- Lines with no fields filled out indicate no FC instruction is necessary in that spot.

3.5.5 Optimizations

Clearly, not all the possible combinations are explored above. The flexibility of the flow control instruction allows for more creative flow control operations, or (more likely) optimizations.

One of the easiest optimizations makes use of the B_POP_CNT to merge consecutive ENDIF statements:

```
          JUMP_FUNC          A_OP                    B_POP_CNT
          |       JUMP_ANY    |       B_OP0           |   B_ELSE
          |       |   OP       |       |       B_OP1   |   |   JUMP_ADDR
          |       |   |        |       |       |       |   |   |
IF c      0x0f    0   JUMP     NONE    INCR    NONE    0   0   ENDIF_0+1
[...]
IF c      0x0f    0   JUMP     NONE    INCR    NONE    0   0   ENDIF_1+1
[...]
IF c      0x0f    0   JUMP     NONE    INCR    NONE    0   0   ENDIF_2+1
[...]
ENDIF     0x00    1   JUMP     NONE    DECR    NONE    1   0   0
ENDIF     0x00    1   JUMP     NONE    DECR    NONE    1   0   0
ENDIF     0x00    1   JUMP     NONE    DECR    NONE    1   0   0
```

Becomes

```
IF c      0x0f    0   JUMP     NONE    INCR    NONE    0   0   ENDIF+1
[...]
IF c      0x0f    0   JUMP     NONE    INCR    DECR    1   0   ENDIF+1
[...]
IF c      0x0f    0   JUMP     NONE    INCR    DECR    2   0   ENDIF+1
[...]
ENDIF
ENDIF
ENDIF     0x00    1   JUMP     NONE    DECR    NONE    3   0   0
```

3.6 Note on Floating Point

X1K FP is designed to be compliant with the Shader Model 3, which does not officially support IEEE special values (denormal, infinity, NaN), and allows for leniency in various corner cases.

X1K FP strives to provide a more complete IEEE floating point implementation. X1K FP supports the IEEE 32-bit floating point format, with 23 bits mantissa, 8 bits biased exponent (bias 127), and 1 bit sign. The X1K FP also supports the special IEEE values (denormal, infinity, NaN), but there are some important caveats in the implementation which are noted below. There is no distinction between an sNaN and a qNaN.

3.6.1 Pervasive Deviations from IEEE

The most pervasive caveat is that denormals are flushed to an appropriately signed zero throughout X1K FP. There is no gradual underflow, and identities are not preserved for denormal values. This will be apparent in comparison operations where a denormal is treated as equivalent to zero.

Also pervasive, the internal rounding mode is not configurable and is not exact to the IEEE standard. It could best be said that rounding is random; operations in and near X1K FP round with differing standards. Most ALU operations are accurate to within one bit on each input; transcendental functions have larger tolerances.

The lack of separable multiply and add instructions has consequences on rounding and sign preservation; when using MAD to perform only a multiply or addition, keep in mind that the other operation may influence the result despite apparent identities. For example, the obvious instructions to use for moving a value from one register to another both utilize MAD, either with the additive identity "0 * 0 + r1", or a combination of additive and multiplicative identities, "r0 * 1 + 0". Neither these instructions will correctly copy -0.0, because the adder cannot generate -0.0 except with two negative inputs. In this case, a more accurate move instruction would be "-0 * 0 + r1". (the ideal MOV instruction is described below).

X1K FP only supports comparisons against zero (predication, ALU result, and CMP) and +0.5 (CND), and this has consequences for implementing a general compare function with special values. It is tempting to implement a general comparison between values A and B by subtracting the results, but this will not have the desired effect for special values. In IEEE, an infinite value is equivalent to itself, but NaN is never equivalent to NaN. Yet (infinity - infinity) = (NaN - NaN) = NaN, and the results are indistinguishable. The limited operator set further complicates issues, since (A > B) is not equivalent to !(A <= B) when either input is NaN.

The behaviour for CMP and CND is described below. When using the predicate comparison operators, the following hold for special values:

VALUE	X<0	X>=0	X==0	X!=0
+0.0	0	1	1	0
-0.0	0	1	1	0
+Inf	0	1	0	1
-Inf	1	0	0	1
NaN	0	0	0	1

Denormals compare as equivalent to zero. Note that the only way a denormal may be involved in a comparison for predicate/alu result is if the output modifier is disabled with OMOD_DISABLED.

3.6.2 ALU Non-Transcendental Floating Point

Non-transcendental ALU operations maintain extra precision to represent computations where an intermediate result exceeds IEEE's finite range. For example, if a MAD generates a result outside the finite range, but the output modifier brings the value back into range, the ALU will generate a finite value, not infinity.

The ALU accepts denormal values, but denormals are flushed to zero, preserving sign. It is possible for a multiplicative output modifier to bring a denormal intermediate result into the normal range; in this case, the ALU will generate a normal nonzero value.

The ALU MAD operation, which many ALU operations are based on, follows standard IEEE rules when handling special input values, for example:

OPERATION	RESULT	NOTES
x * NaN	NaN	x is any value.
0.0 * Inf	NaN	
Inf * Inf	Inf	
Inf * -Inf	-Inf	
0.0 * -0.0	-0.0	

OPERATION	RESULT	NOTES
x + NaN	NaN	x is any value.
Inf + -Inf	NaN	
Inf + Inf	Inf	
Inf + -1.0	Inf	
0.0 + -0.0	0.0	
-0.0 + -0.0	-0.0	

Dot products may lose precision in cases where the values to be added differ greatly in magnitude. For example, if the two largest values to be added cancel exactly, and the next-largest value has a magnitude smaller by a factor of 2^25 or more, X1K FP will emit +0.0 rather than the sum of the two remaining components. IEEE is silent on the behaviour of such fused operations, and it seems unlikely that this condition will manifest very often.

MIN and MAX operations return the second argument if either input is NaN (this is consistent with IEEE and SM3 specifications); infinite values compare as usual. If both inputs are +-0.0, MIN and MAX will return the second input (consistent with IEEE and the SM3 spec)—as a result, MIN(+0, -0) == -0, and MIN(-0, +0) == +0.

CND and CMP operations return the second argument if either input is NaN; infinite values compare as usual. As with the predicate compare operators, +0.0 and -0.0 are both "equal" to 0.

MIN, MAX, CND, and CMP are guaranteed to return one of their first two arguments. If you use OMOD_DISABLED as well, then you will get a bit-exact representation of one of the first two arguments.

ALU operations usually enable the output modifier, which in turn standardizes NaN values and flushes denormal results to zero. A MOV instruction which preserves the source bits may be implemented by setting OMOD_DISABLED for the instruction and using the MAX(src, src) instruction. The output modifier cannot be disabled for a saturated MOV (MOV with clamping enabled).

3.6.3 ALU Transcendental Floating Point

In X1K FP, transcendental operations are EX2, LN2, RCP, RSQ, SIN, and COS (mathematically speaking, one of these functions does not belong). Transcendentals do not maintain extra internal precision; as a result, if the result of the transcendental operation exceeds the IEEE finite range, the ALU will generate infinity even if the output modifier would bring the result back into range. Similarly if the result is denormal, the ALU will generate a pure zero (preserving sign) even if the output modifier would bring the result back into the normal range.

Special values are computed as shown in the following table:

INPUT	EX2	LN2	RCP	RSQ	SIN	COS
+0.0	+1.0	-Inf	+Inf	+Inf	+0.0	+1.0
-0.0	+1.0	-Inf	-Inf	+Inf *	-0.0	+1.0
+Inf	+Inf	+Inf	+0.0	+0.0	NaN	NaN
-Inf	+0.0	NaN	-0.0	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

Note:* For RSQ, recall that the square root occurs first. IEEE specifies sqrt(-0.0) -> -0.0; the X1K FP deviates from this.

3.6.4 Texture Floating Point

Projected texture coordinates are processed in the X1K FP block before being sent to the texture unit. The texture unit does not accept NaN, so NaN coordinates are converted to +infinity before being sent to the texture unit. As with the ALU, denormal inputs and denormal results are converted to pure zero, preserving sign.

The multiplier used for projection and cubemapping does not follow IEEE rules when handling special values. This will become apparent only when you attempt to project or cubemap a coordinate that contains an infinite or NaN component.

You should use caution when generating very large values for use as coordinates in a texture lookup. These values may generate infinite values when scaled by the texture dimensions or projected.

3.7 Errata

There is a problem with texture semaphores: if an instruction that does a wait on a texture semaphore is preceeded by a flow-control instruction, that second instruction may execute before the texture semaphore has returned.

As a workaround, always do a tex sem wait on a flow-control instruction if any possible next instruction contains a tex sem wait (including branch and call/return). You should also remove the tex sem wait on the other instructions (unless you are in full-flow control mode) or you risk waiting on a younger thread, which shouldn't cause a deadlock but could be bad for performance. In full-flow control mode, you can leave the tex sem wait bit set for those other instructions.

Chapter 4
DPP Application Binary Interface

The ATI Data Parallel Processor (DPP) Array is the collection of floating-point processors found in ATI graphics hardware. The processors execute programs encoded in an ISA described in the previous chapter. This chapter specifies how executable programs must be packaged in order to run unmodified on the DPP Array, via the program loader embedded in CTM.

4.1 Executable Files

The DPP executable file format must be the Executable and Linking Format (ELF). ELF is specified as a component of the Tool Interface Standard (TIS), and its documentation is publicly available. This document assumes familiarity with ELF, and details only the portions of the ELF file that are specific to loading programs for the DPP.

4.1.1 File Format

The top-level layout of an executable file object (the "Execution View" of the ELF format) is given in the following figure:

ELF Header
Program Header
Segment 1
...
Segment n
Section Header Table (Optional)

4.1.2 ELF Header

DPP-specific fields in the ELF Header are:

Field	Assigned Value	Description
e_ident[EI_CLASS]	ELFCLASS32	32-bit executable object.
e_ident[EI_DATA]	ELFDATA2LSB	Data is little endian.
e_ident[EI_OSABI]	ELFOSABI_ATI	The ABI for ATI DPP devices (current value is 98).
e_ident[EI_ABIVERSION]	1	ABI version number.
e_type	ET_EXEC	The object contains an executable file.
e_machine	EM_ATI_R5XX	Specifies the ATI X1k series encoding (current value is 122).
e_entry	0	No defined entry point virtual address.
e_flags	EF_ATI_DPP	Required processor specific flag (current value is 1).

4.1.3 Program Code Sections

The program executable code is contained within a single .text section of the ELF file with a program header element of type PT_LOAD:

Name	Type	Flags	Section Contents
text	SHT_PROGBITS	SHF_ALLOC SHF_EXECINSTR	The executable instructions of a program.

4.1.4 Program Loading

A binary object to be loaded for execution contains information in addition to the instruction code found in the .text section of its binary. This information is encoded in seven notes encapsulated in one or more sections of type SHT_NOTE with program header elements of type PT_NOTE. This information is used to ensure the validity of the execution environment in CTM with respect to the program being loaded.

The notes are summarized in the following table, and described in detail in the subsections below:

Note	Description
Program Information	Ancillary information used by the loader to set up program.
Inputs	Which inputs are referenced in the program.
Outputs	Which outputs are referenced in the program.
Conditional Output	Whether a conditional output value is written.
Float32 Constants	Which FLOAT32 constants are referenced in the program.
Int32 Constants	Which INT32 constants are referenced in the program.
Early Program Exit	Whether the program contains an early exit command.

Program Information Note

The Program Information Note is summarized in the following tables. If the object file does not contain a program information note, the object is ill conditioned and will fail to load:

Field	Size in Bytes	Value
namesz	4	8
descsz	4	sizeof(dpp_proginfo)
type	4	ELF_NOTE_ATI_PROGINFO (current value is 1)
name	8	ELF_NOTE_ATI (current value is "ATI DPP")
desc	sizeof(dpp_proginfo)	The DPP Program Information structure (see below).

The DPP Program Information structure (dpp_proginfo) found in the Program Information Note is:

Field	Type	Value	Description
Reserved	Elf32_Word	0x00000001	Reserved
FG_DEPTH_SRC	Elf32_Word	per-program	A 1 in bit 0 specifies conditional value is exported.
US_CONFIG	Elf32_Word	per-program	A 1 in bit 0 specifies that output writes are uncached.
US_PIXSIZE	Elf32_Word	per-program	Largest register used in the program
US_FC_CTRL	Elf32_Word	per-program	A 1 in bit 31 specifies full flow control functionality

Field	Type	Value	Description
US_CODE_ADDR	Elf32_Word	per-program	Lower 16 bits is instruction where program execution begins. Upper 16 bits is instruction after which to halt
US_CODE_RANGE	Elf32_Word	per-program	Upper 16 bits is (INST_CNT - 1); lower 16 bits is 0.
Reserved	Elf32_Word	0x000000c0	Reserved
Reserved	Elf32_Word	0x00020000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
Reserved	Elf32_Word	0x00000000	Reserved
INSTR_CNT	Elf32_Word	per-program	Number of instructions in program

Inputs Note

The Inputs Note is summarized in the following table. If an object file does not contain an inputs note, then the program loader acts as if the program has no inputs:

Field	Size in Bytes	Value
namesz	4	8
descsz	4	4 * ninputs
type	4	ELF_NOTE_ATI_INPUTS (current value is 2)
name	8	ELF_NOTE_ATI (current value is "ATI DPP")
desc	4 * ninputs	List of the input indices used in the program.

Outputs Note

The Outputs Note is summarized in the following table. If an object file does not contain an outputs note, then the program loader acts as if the program has no outputs:

Field	Size in Bytes	Value
namesz	4	8
descsz	4	4 * noutputs
type	4	ELF_NOTE_ATI_OUTPUTS (current value is 3)
name	8	ELF_NOTE_ATI (current value is "ATI DPP")
desc	4 * noutputs	List of the output indices used in the program.

Conditional Output Note

The Conditional Output Note is summarized in the following table. If an object file does not contain a conditional output note, then the program loader acts as if the program has no conditional output:

Field	Size in Bytes	Value
namesz	4	8
descsz	4	4
type	4	ELF_NOTE_ATI_CONDOUT (current value is 4)
name	8	ELF_NOTE_ATI (current value is "ATI DPP")
desc	4	1 if conditional output is written by program; otherwise 0.

Float32 Constants Note

The Float32 Note is summarized in the following table. If an object file does not contain a Float32 constants note, then the program loader acts as if the program has no float32 constant references:

Field	Size in Bytes	Value
namesz	4	8
descsz	4	4 * nfloat32consts
type	4	ELF_NOTE_ATI_FLOAT32CONSTS (current value is 5)
name	8	ELF_NOTE_ATI (current value is "ATI DPP")
desc	4 * nfloat32consts	List of the float32 constant indices used in the program.

Int32 Constants Note

The Int32 Constants Note is summarized in the following table. If an object file does not contain an Int32 constants note, then the program loader acts as if the program has no int32 constant references:

Field	Size in Bytes	Value
namesz	4	8
descsz	4	4 * nint32consts

Field	Size in Bytes	Value
type	4	ELF_NOTE_ATI_INT32CONSTS (current value is 6)
name	8	ELF_NOTE_ATI (current value is "ATI DPP")
desc	4 * nint32consts	List of the int32 constant indices used in the program.

Early Program Exit Note

The Early Exit Note is summarized in the following table. If an object file does not contain an early program exit note, then the program loader acts as if the program has no early program exit command:

Field	Size in Bytes	Value
namesz	4	8
descsz	4	4
type	4	ELF_NOTE_ATI_EARLYEXIT (current value is 7)
name	8	ELF_NOTE_ATI (current value is "ATI DPP")
desc	4	1 if the program includes an early exit; otherwise 0.

Chapter 5
Device Interface

This chapter describes the interfaces exported by the ATI CTM Library, as detailed in the **amDeviceManaged.h** file. For more information, refer to the CTM Device Interface HTML files.

5.1 Functions

5.1.1 amCloseManagedConnection

```
amCloseManagedConnection  ( AMmanagedDevice  dev )
```

Close a managed device.

This function shuts down an instantiation of a managed device. The handle is undefined after the function call.

Parameters

dev: (in) Handle to the managed device.

Returns

No return value.

5.1.2 amCommandBufferConsumed

```
amCommandBufferConsumed  (  AMmanagedDevice  dev,
                            AMuint32  buf
                          )
```

Submit a command buffer to a managed device.

This function submits a command buffer to a managed device. The command buffer location (as a GPU address) and size are passed in. It returns a unique 32-bit unsigned integer identifying this command buffer submission request. This return value can be used to query whether the command buffer has been consumed.

Parameters

dev: (in) Handle to the managed device.

buf: (in) Command buffer identifier that was returned by amSubmitCommandBuffer.

Returns

Returns 1 if command buffer has been consumed; otherwise returns 0.

5.1.3 amOpenManagedConnection

```
amOpenManagedConnection ( const AMchar8 * dev,
                          AMmanagedDeviceInfo * info
                        )
```

Instantiate a managed device.

This function must be used to create a managed device instantiation. All CTM functions take a handle to a managed device created here.

Parameters

dev: (in) Name of device to open a connection to.

info: (out) Information structure populated by this function.

Returns

Returns an opaque managed device handle, or NULL upon failure.

5.1.4 amSubmitCommandBuffer

```
amSubmitCommandBuffer ( AMmanagedDevice  dev,
                        AMuint32  addr,
                        AMuint32  size
                       )
```

Submit a command buffer to a managed device.

This function submits a command buffer to a managed device. The command buffer location (as a GPU address) and size are passed in. It returns a unique 32-bit unsigned integer identifying this command buffer submission request. This return value can be used to query whether the command buffer has been consumed.

Parameters

dev: (in) Handle to the managed device.

addr: (in) 32-bit unsigned integer specifying GPU address of command buffer.

size: (in) size of the command buffer in units of 32-bit unsigned integers.

Returns

Returns a unique 32-bit unsigned integer identifying this command buffer submission request, or zero if the submission has failed.