	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	17 January, 2002	[date \@ "d MMMM, yyyy"]	GEN-CXXXXX-REVA	1 of 49

Author: Andrew Gruber, Andi Skende, Tom Frisinger

Issue To:	**Copy No:**

Shader Processor

Rev 2.02

Overview: This document describes the overall architecture of the Shaders, interfaces, partitioning into functional blocks as well as the timing of the shader pipeline. It's intended for use by hardware designers.

AUTOMATICALLY UPDATED FIELDS:
Document Location : //ma_andi_mobile/..../doc_lib/parts/sp
Current Intranet Search Title: Shader Processor

APPROVALS	
Name/Dept	Signature/Date

Remarks

THIS DOCUMENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATI TECHNOLOGIES INC. THROUGH UNAUTHORIZED USE OR DISCLOSURE.

"Copyright 2001, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished work created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this unpublished work. The copyright notice is not an admission that publication has occurred. This work contains confidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or transmitted in any form or by any means without the prior written permission of ATI Technologies Inc."

Table Of Contents

Revision Changes:

Rev 0.0 (Steve Morein)
Date: April, 2001
Initial revision.

Document started

Rev 0.1 (Andi Skende)
Date: May 09, 2001

Updated, added the instruction formant, initial block diagrams and preliminary interface description

Rev 0.2 (Andi Skende)
Date: May 21, 2001

A more detailed description of the SP<->TEX, RE/Sequencer <->SP interfaces.

Rev 0.3 (Andi Skende)
Date: June 19, 2001

Added the paragraph related to shader functional limitations that the compiler needs to be aware of.
A new updated and compressed version of ALU instruction format.

Rev 0.4 (Andi Skende)
Date: June 20, 2001

Updated the Introduction of this document. A new Pipeline Timing Diagram was inserted.

Rev 0.5 (Andi Skende)
Date: July 31, 2001

Merged in the Shader Hardware Spec. A more detailed description of the interfaces with the other blocks was added. Updated some of the diagrams to a more correct representation of the datapaths.

Rev 0.6 (Andi Skende)
Date: August 17,2001

A more detailed description/definition of Shader interfaces with the other blocks.
A more detailed description of the instruction supported by Shader Processor and it's relation to instruction set exposed at API level.

Rev 0.7 (Andi Skende)
Date: November 8, 2001

Updated the Alu instruction word definition and the list of the alu instruction opcodes supported by the shader pipe ALU unit.

Rev 0.8 (Andi Skende)
Date: November 27, 2001

Updated the definition of the External Interfaces

Rev 0.9 (Andi Skende)
Date: December 10, 2001

Updated the definition and naming of some of the external interfaces, rearranged the ALU instruction word definition such that the fields are dword aligned.
The instruction opcode definition was updated and expanded.

Rev 1.0 (Andi Skende)
Date: January 15, 2002

Updated most of the diagrams. Updated the External Interface definitions. Added a description of the Parameter Interpolation Units. Added a diagram description of the GPR write data paths.

Rev 1.1 (Andi Skende)
Date: January 21, 2002

Updated some of the external interface definitions.
Specified the expected behavior of hardware implementation of some shader opcode with some corner case values as input arguments. The MS Reference Rasterizer shader was used as guideline.

Rev 1.2 (Andi Skende)
Date: January 22, 2002

Updated some of the external interface definitions.

Rev 1.3 (Andi Skende)
Date: January 30, 2002

1. Changed the order of the swizzle bits per channel in the instruction word such that the LSBs belong to the red channel …MSBs for the alpha channel.
2. Extended the SQ_SP_Instuct bus width to 21 bits to account for the Scalar Opcode being 6 bits instead of 5.

Rev 1.4 (Andi Skende)
Date: August 12, 2002

1. Modified the instruction word format.
2. Added new opcodes (DST and MUL_PREV2) and redefined the opcoded values for both vector and scalar instructions.
3. Updated the SQ_SP Instruction interface definition.

Introduction

Shader Pipe (SP) serves as the central Arithmetic and Logic Unit (ALU) for the R400 Graphics Processor. There are four identical Shader pipelines in the R400 architecture. Differently from previous ATI architectures, the R400 Shader Pipe truly represents a Unified Shader Architecture (USA). In R400, both vertex and pixel shading operations are implemented through the shader units. The R400 Shader Pipe represents a Single Instruction Multiple Data (SIMD) architecture. All the shader units of each and every pipe execute the same ALU instruction on different sets of vertex parameters/pixel values. The building blocks of the R400 shader units execute operations on single precision IEEE floating-point values.

State

1.1 Shader State

1.1.1 *GPRs (General Purpose Registers)*

The general-purpose registers are 128 bits wide, composed of four 32-bit values. Depending on the operation these values can be interpreted, among others, as ABGR, WZYX, QRTS, QWVU, or AUYV respectively; to simplify matters the only the alias WZYX will be used throughout this document.

To hide the latency of memory accesses the shader pipe will switch between different vectors. This is the same as the idea of "microthreading" that some advanced CPU's are investigating. The large register file is split between the vectors executing in the shader pipe. The management of the shader register file is automatic, and not visible to a program executing on a vector, except that a program is required to declare the number of GPRs it needs to execute. The hardware will not start a vector until the required number of registers is available. There is a direct tradeoff between the number of registers each program/vector needs and the number of vectors than can be simultaneously resident. If there are too few vectors resident, then the latency of memory accesses can no longer be hidden and performance suffers.

There are a total of 128 general purpose registers. A given shader can request at most 64 GPRs. Requesting a very large number of GPRs will make it difficult to hide memory latency, but the program will still execute and generate the correct result.

Most pixel programs are expected to have less than eight registers; vertex programs are expected to have less than sixteen registers.

The number of registers a program needs is the maximum number of registers it needs at any instruction. If a program needs only 3 general purpose registers nearly all of the time, except for a short period when it needs 8, it still needs to allocate eight. A significant performance optimization is for the compiler to reorder the instructions to minimize the number of needed registers.

127	95	63	31 0	GPR
W/A	Z/B	Y/G	X/R	R0
				R1
				R127

Notation: R0.W refers to the bits 96 to 127 of register one (so does R0.A).

1.1.2 *Constants*

There are also (512) constant registers available to vertex and pixel shaders in the primary command stream.

127	95	63	31 0	Constant
W/A	Z/B	Y/G	X/R	C0
				C1
				C511

Real-time has it's own 256. Constants are physically part of the Sequencer unit. As it will become clear by reading the rest of this document, the content of the constants can be made available to the ALU units of the shader pipes in the form of one of the possible ALU operation arguments. The ALU instruction word provides for that.

The constant file is shared between vertex shaders and pixel shaders, it is the drivers job to allocate one section to pixel shaders and another to vertex shaders to match the D3D programming model; other API's may allow more freedom.

To be able to support multiple textures easily, and to save hardware area, the texture state registers are stored in separate pool of constant registers. Each texture constant holds 192 bits of texture state. Rather than have four or six sets of texture registers as we do in the R100, R200, and R300, by storing them in constant memory we can save area by reusing the logic already needed to update the constant registers in order. Since any single texture instruction will only fetch from one texture we do not need the simultaneous access we would get with implementing this as "normal" registers.

1.1.3 *Previous Instruction Result*

(This section is no longer accurate and needs updating)
Within an ALU clause the result of the previous operation is explicitly available, without requiring a register read (due to an exposed pipeline delay, the result of the previous operation can not be read from the register file without a one-instruction delay slot). There are two distinct previous instructions, one scalar and one vector.
This register is not preserved between the end of one ALU clause and the beginning of another.
It can be used to avoid using another GPR if the result is not needed. Also, the output modifiers, which do affect the result of an instruction written into GPRs, do not affect the Previous Result content.

1.2 Initial state

1.2.1 *Vertex Shader*

A vertex shader initially has the X value of R0 set to the vertex index. No other registers are filled. The vertex shader must use the index to fetch the vertex data from the vertex array(s). The pointers to the vertex arrays should be placed in texture constant registers by the driver.

1.2.2 *Pixel Shader*

The pixel shader has the interpolated values generated from the values exported by the vertex shader.

If the vertex shader exports to parameter 0, and the R400 is appropriately programmed, then GPR 0 in the pixel shader contains the interpolated values for that parameter at that pixel.

2. Program Format

(This section is no longer accurate and needs updating)
A pixel or vertex shader program consists of 16 clauses, eight texture clauses and eight alu clauses.
The instructions in a clause will be executed sequentially. If a given instruction is implementing, for example,
T * S + D (T = texture for SRC A, S = Specular for Source B, D = Diffuse for Source C), it's the Sequencer's task to resolve the dependencies between the ALU clause and the respective texture clause. In other words, the sequencer will not issue the ALU instruction using texture data as input to the shader pipe, until the texture request has been issued to and serviced by the texture pipe. In general, the Shader is not aware of the origin of the SRC A, SRC B and SRC C data (texture, diffuse, specular, vertex parameters etc). Three address pointers into the register files (one for each operand) are all the shaders need to fetch these operands. In reality, as it will become more evident later in this document, there is no need for the pointer values to be passed to the shader units. This is related to the GPR's read/write mechanism we have chosen to implement.

3. ALU

3.1 ALU structure

ALU consist of two distinct units: the 'Vector' ALU and the 'Scalar' ALU. The Vector ALU performs operations in parallel across a 4-component vector, while the Scalar ALU performs operations on a single component of a vector which is then replicated across all components. A single instruction will 'co-issue' both a Vector and a Scalar instruction. Almost all scalar instructions require SrcC as an operand. When the Vector operation is only using SrcA and SrcB as operands (such as in a MUL (Multiply) instruction), the scalar pipe is free to use SrcC as it wishes. When the vector pipe is also consuming SrcC, such as in a three operand instruction like MULADD (Multiply and ADD), SrcC is fixed for the scalar pipe. It's important to understand that the given scalar operation still occurs on SrcC. Under most circumstances this will result in undesirable behavior unless the scalar operation is benign and has masked its destination writes.

For more details on the overall structure of the Shader ALU, refer to the figures in Section 5 of this document.

3.2 ALU instruction format

There are two opcodes present in the ALU instruction, one for the Vector operation and one for Scalar operation. The idea is that we can allow a 4-component vector operation (if the compiler permits) coissued with a Scalar Operation. The Scalar unit may use SRC C, depending on whether this source is being used by the vector operation. Please refer to Section 8 of this document on the limitations of a Vector or Scalar instruction issuing.

Field	Bits	Size	Description
SRC A Select	95	1	Select bit for selecting Constant vs. Register 0: Constant 1: Register
SRC B Select	94	1	Select bit for selecting Constant vs. Register 0: Constant 1: Register
SRC C Select	93	1	Select bit for selecting Constant vs. Register 0: Constant 1: Register
Vector Opcode	92:88	5	Opcode for Vector instruction
SRC A Register/Constant Pointer	87:80	8	Location of SRC A in the Register or Constant file If Register, **Bits [87]-[80]** denote: **Bit [87]** 0: Do not execute ABS on input register 1: Execute ABS on input register **Bit [86]** 0: Logical register addressing 1: Current Loop Index relative register addressing **Bits [85]-[80]** location of SRC A in the register file If Constant, **Bits [87]-[80]** denote: **Bits [87]-[80]** location of SRC A in the constant file
SRC B Register/Constant Pointer	79:72	8	Location of SRC B in the Register or Constant file Refer to **SRC A Register/Constant Pointer**
SRC C Register/Constant Pointer	71:64	8	Location of SRC C in the Register or Constant file Refer to **SRC A Register/Constant Pointer**
Constant0 Logical/Relative	63	1	The address pointer into the Constant file is relative to some base address register (works in conjunction with **Relative Address Register Select**) 0: Logical constant addressing 1: Relative constant addressing
Constant1 Logical/Relative	62	1	The address pointer into the Constant file is relative to some base address register (works in conjunction with **Relative Address Register Select**) 0: Logical constant addressing 1: Relative constant addressing
Relative Address Register Select	61	1	This bit determines the address register used as base register when Constant indexing is relative. It is used in conjunction with **Constant0 Logical/Relative** and **Constant1 Logical/Relative** fields. 0: Current Loop Index relative 1: Address Register relative
Predicate Select	60:59	2	**Bits [60][59]** 0X: No predication 10: Predicated – 1 means skip, 0 means execute 11: Predicated – 0 means skip, 1 means execute
SRC A Modifier	58	1	0: No modification 1: Negate
SRC B Modifier	57	1	0: No modification 1: Negate
SRC C Modifier	56	1	0: No modification 1: Negate

SRC A Swizzle	55:48	8	2 bits for each component **Bits [52][51]** – W channel swizzle 00: leave W 01: X 10: Y 11: Z **Bits [50][49]** – Z channel swizzle 00: leave Z 01: W 10: X 11: Y **Bits [48][47]** – Y channel swizzle 00: leave Y 01: Z 10: W 11: X **Bits [46][45]** – X channel swizzle 00: leave X 01: Y 10: Z 11: W
SRC B Swizzle	47:40	8	2 bits for each component (refer to **SRC A Swizzle**)
SRC C Swizzle	39:32	8	2 bits for each component (refer to **SRC A Swizzle**)
Scalar Opcode	31:26	6	Opcode for the Scalar instruction
Scalar Clamp	25	1	0: No clamp 1: Clamp to [+0.0f, 1.0f] range
Vector Clamp	24	1	0: No clamp 1: Clamp to [+0.0f, 1.0f] range
Scalar Write Mask	23:20	4	Defines which out of 32 bit words (four of them) in the scalar result is written back in the Register file. There's one bit per channel. **Bit [23]** 0: Leave the current value 1: Write Scalar W **Bit [22]** 0: Leave the current value 1: Write Scalar Z **Bit [21]** 0: Leave the current value 1: Write Scalar Y **Bit [20]** 0: Leave the current value 1: Write Scalar X
Vector Write Mask	19:16	4	Defines which out of 32 bit words (four of them) in the vector result is written back in the Register file. There's one bit per channel. **Bit [19]** 0: Leave the current value 1: Write Vector W **Bit [18]** 0: Leave the current value 1: Write Vector Z **Bit [17]** 0: Leave the current value 1: Write Vector Y **Bit [16]** 0: Leave the current value 1: Write Vector X

Scalar Destination Pointer	15:8	8	**Bit [15]** denotes the destination of the ALU results 0: Register file (Scalar and Vector) 1: Export file (Scalar and Vector) If Register file destination, **Bits [14]-[8]** denote: **Bit [14]** determines whether the scalar destination address into Register file is logical or relative. 0: Logical register addressing 1: Current Loop Index relative register addressing **Bits [13]-[8]** specifies the address into the Register file for the result of scalar operation. If Export file destination, **Bits [14]-[8]** denote: **Bit [14]** determines parameter export masking behavior. See table in 3.2.1.4 **Bits [13]-[8]** are unused. Must Be Zero.
Vector Destination Pointer	7:0	8	**Bit [7]** determines the ABS input modifier on all constants in instruction 0: Do not execute ABS on all constants 1: Execute ABS on all constants **Bit [6]** determines whether the vector destination address into Register or Export file is logical or relative. 0: Logical register addressing (Must Be Zero for exports on r400) 1: Current Loop Index relative register addressing If Register file destination, **Bits [5]-[0]** denote: **Bits [5]-[0]** specifies the address into the Register file for the result of vector operation. If Export file destination, **Bits [5]-[0]** denote: **Bits [5]-[0]** specifies the address into the Export file for the result of vector and scalar operations.

There's a total of 96 bits per instruction. The bit allocation and assignment for the different fields of the instruction word was done with under the limitations that they should be DWORD (32 bit) aligned.

3.2.1 *ALU Instruction Word Interpretation*

3.2.1.1 Argument Selection and Pointers

There can be a maximum of three sources (operands) required for an ALU operation of a vector type.
The R400 ALU instruction word definition provides location pointers into the Register file (GPRs) or Constant file for each of the three sources (**SRC A Register/Constant Pointer, SRC B Register/Constant Pointer**, **SRC B Register/Constant Pointer**).

3.2.1.1.1 Logical vs. Relative Registers

SrcA, SrcB and SrcC GPR locations denoted by SRC A (B, C) Register/Constant Pointer fields of the ALU instruction word, can be logical as well as relative addresses. If relative, they are relative to the Current Loop Index present in the Sequencer state.

3.2.1.1.2 Logical vs. Relative vs. Absolute Constants

Constants can also be addressed in logical and relative fashions along with an absolute mode. When relative, they can be relative to either the Current Loop Index (CLI) or Address Register (AR) in the sequence state. The truth table below shows the instruction fields that are used to decode the nature of the constant values.

Constant0 Logical/Relative 0: Logical 1: Relative	**Constant1 Logical/Relative** 0: Logical 1: Relative	**Relative Address Register Select** 0: Current Loop Index 1: Address Register	**Constant0**	**Constant1**	**Constant2**
0	0	0	Logical	Logical	Same as Constant1
0	0	1	Absolute	Absolute	Same as Constant1
1	0	0	Relative, CLI	Logical	Same as Constant1
0	1	0	Logical	Relative, CLI	Same as Constant1
1	1	0	Relative, CLI	Relative, CLI	Same as Constant1
1	0	1	Relative, AR	Logical	Same as Constant1
0	1	1	Logical	Relative, AR	Same as Constant1
1	1	1	Relative, AR	Relative, AR	Same as Constant1

Note from the table that if both Constants are relative, they are relative to the same value, either the Current Loop Index (CLI) or Address Register (AR).

Constant0 refers to the first constant in the instruction; Constant1 and Constant2 refer to the second and third constants in the instruction respectively.

3.2.1.2 Input and Output Modifiers

The R400 ALU Instruction word definition provides for only three input modifiers for each of the three sources, **Negate**, **ABS** and **Swizzle**. When the source is a Constant value, input modifier **ABS** applies to all constants in instruction. In all situations, **ABS** is always applied before **Negate**. Input modifiers do not apply to PreviousScalar.

The R400 ALU Instruction word provides for two output (result) modifiers: **Write Mask** which only affects the results going into GPRs but not the PreviousScalar and **Clamp**.

3.2.1.3 GPR write-backs

The table below describes the precedence order for "mixed" use of **Scalar Write Mask** and **Vector Write Mask** for GPR write-backs when the **Scalar Destination Pointer** and **Vector Destination Pointer** in the instruction word specify the same GPR. This is done per component (each MASK field is 4 bits wide, one bit per component/channel).

Scalar Write Mask	**Vector Write Mask**	Result of GPR write-back
0	0	Don't write (mask)
1	0	Write Scalar Component
0	1	Write Vector Component
1	1	Write Scalar Component

3.2.1.4 Export and Predicate related decoding

Exports are allowed from either Scalar or Vector Pipe. Similar to the GPR write-backs, masking of export data is permitted. The mask is present in the ALU instruction word. When exporting, the export address used is the **Vector Destination Pointer** present in the instruction word. The **Scalar Destination Pointer** in this case is always ignored. The table below describes the "mixed" use of **Scalar Write Mask** and **Vector Write Mask** per component when exports are coissued. The ability to generate 0.0f or 1.0f during export provides one method for defaults.

Scalar Write Mask	**Vector Write Mask**	Result of Export
0	0	**Bit [14]** 0: Don't write (mask) 1: Write 0.0f
1	0	Write Scalar Component
0	1	Write Vector Component
1	1	Write 1.0f

A few other export related definitions and restrictions:

1) Exporting of 'Color/Fog' is a special case.
    a) When exporting Fog, Color must be exported at the same time. Fog will be exported in the Scalar pipe and Color in the Vector pipe.
    b) The SP produces a final export Color by obeying the vector/scalar mask rules for exports. The SP does not see bit[14] so when the vector and scalar masks for a given channel are 0 a 0.0f is generated. The SP then merges Fog (always from the scalar pipe) into the final export color. Finally, channel masking is applied in (SQ/SX) only when the scalar and vector masks for that channel are 0 and bit[14] is 0.
    c) Note for Fog to work correctly, SW should always output the same Fog factor from the scalar pipe for all masked writes and all channels of Color should be written before the shader exits.

3.2.1.5 Export Types and Addresses

The location where the data should be put in the event of an export is specified by in the destination pointer field of the ALU instruction word. Following is a list of the possible types of exports and the range of addresses.

Vertex Shading

0:15	- 16 parameter cache
16:31	- Empty (Reserved?)
32	- Export Address
33:37	- 5 vertex exports to the frame buffer and index
38:46	- Empty

47	- Debug Address
48:52	- 5 debug export (interpret as normal memory export)
53:59	- Empty
60	- export addressing mode
61	- Empty
62	- position
63	- sprite size export that goes with position export (X = point size, Y = edge flag is bit 0, Z = VtxKill is bitwise OR of bits 30:0 (any bit other than sign means VtxKill).)

Pixel Shading

0	- Color for buffer 0 (primary)
1	- Color for buffer 1
2	- Color for buffer 2
3	- Color for buffer 3
4:15	- Empty
16	- Buffer 0 Color/Fog (primary)
17	- Buffer 1 Color/Fog
18	- Buffer 2 Color/Fog
19	- Buffer 3 Color/Fog
20:31	- Empty
32	- Export Address
33:37	- 5 exports for multipass pixel shaders.
38:46	- Empty
47	- Debug Address
48:52	- 5 debug exports (interpret as normal memory export)
60	- export addressing mode
61	- Z for primary buffer (Z exported to 'X' component)
62:63	- Empty

3.3 ALU Opcodes

The following table represents the ALU operations/opcodes supported by the Vector unit.

Name	Opcode	Function	Notes
ADD	0x00	Result = SrcA + SrcB;	Per component add. 2 operand; possible coissue.
MUL	0x01	Result = SrcA * SrcB;	Per component multiply. 2 operand; possible coissue.

MAX	0x02	If (SrcA >= SrcB)
		  Result = SrcA;
		Else
		  Result = SrcB;
			Per component maximum.
			2 operand; possible coissue.

MIN	0x03	If (SrcA < SrcB)
		  Result = SrcA;
		Else
		  Result = SrcB;
			Per component minimum.
			2 operand; possible coissue.

SETE	0x04	If (SrcA = SrcB)
		  Result = 1.0f;
		Else
		  Result = 0.0f;
			Per component set equal.
			2 operand; possible coissue.

SETGT	0x05	If (SrcA > SrcB)
		  Result = 1.0f;
		Else
		  Result = 0.0f;
			Per component set greater than.
			2 operand; possible coissue.

SETGE	0x06	If (SrcA >= SrcB)
		  Result = 1.0f;
		Else
		  Result = 0.0f;
			Per-component set greater than equal.
			2 operand; possible coissue.

SETNE	0x07	If (SrcA != SrcB)
		  Result = 1.0f;
		Else
		  Result = 0.0f;
			Per component set no equal.
			2 operand; possible coissue.

FRACT	0x08	Result = SrcA + -FLOOR(SrcA);	Per component 'fractional' part of SrcA. 1 operand; possible coissue.
TRUNC	0x09	Result = trunc(SrcA);	Per component 'integer' part of SrcA. 1 operand; possible coissue.

FLOOR	0x0a	Result = TRUNC(SrcA);
		If ((SrcA < 0.0f) && (SrcA != Result))
		  Result += -1.0f;
			Per component floor function.
			1 operand; possible coissue.

MULADD	0x0b	Result = SrcA * SrcB + SrcC;	Per component multiply-add (MAD). 3 operand; no coissue.

CNDE	0x0c	If (SrcA == 0.0f)
		  Result = SrcB;
		Else
		  Result = SrcC;
			Per component conditional move equal.
			3 operand; no coissue.

CNDGE	0x0d	If (SrcA >= 0.0f)
		  Result = SrcB;
		Else
		  Result = SrcC;
			Per component conditional move greater than equal.
			3 operand; no coissue.

CNDGT	0x0e	If (SrcA > 0.0f)
		  Result = SrcB;
		Else
		  Result = SrcC;
			Per component conditional move greater than.
			3 operand; no coissue.

DOT4	0x0f	Result = SrcA.W * SrcB.W +
		         SrcA.Z * SrcB.Z +
		         SrcA.Y * SrcB.Y +
		         SrcA.X * SrcB.X;
			4 component dot product.
			Result replicated in all four channels.
			2 operand; possible coissue.

DOT3	0x10	Result = SrcA.Z * SrcB.Z +
		         SrcA.Y * SrcB.Y +
		         SrcA.X * SrcB.X;
			3 component dot product.
			Result replicated in all four channels.
			2 operand; possible coissue.

DOT2ADD	0x11	Result = SrcA.Y * SrcB.Y +
		         SrcA.X * SrcB.X +
		         SrcC.X;
			2 component dot product with add.
			Result replicated in all four channels.
			3 operand; no coissue.

CUBE	0x12	Result.W = FaceID;
		Result.Z = 2.0f * MajorAxis;
		Result.Y = S cube coordinate;
		Result.X = T cube coordinate;
			Cubemap instruction.
			2 operand (SrcA = Rn.zzxy, SrcB = Rn.yxzz); possible coissue.

MAX4	0x13	Result = max(SrcA.W, SrcA.Z, SrcA,Y, SrcA.X);	4 component maximum. Result replicated in all four channels. 1 operand; possible coissue.

PRED_SETE_PUSH	0x14	If ((SrcB.W == 0.0f) && (SrcA.W == 0.0f)) {
		  Result = 0.0f;
		  SetPredicateReg(Execute);
		}
		Else {
		  Result = SrcA.W + 1.0f;
		  SetPredicateReg(Skip);
		}
			Predicate counter increment equal; Update predicate register.
			Result replicated in all four channels.
			2 operand; possible coissue.
			See note below.

PRED_SETNE_PUSH	0x15	If ((SrcB.W != 0.0f) && (SrcA.W == 0.0f)) {
		  Result = 0.0f;
		  SetPredicateReg(Execute);
		}
		Else {
		  Result = SrcA.W + 1.0f;
		  SetPredicateReg(Skip);
		}
			Predicate counter increment not equal; Update predicate register.
			Result replicated in all four channels.
			2 operand; possible coissue.
			See note below.

PRED_SETGT_PUSH	0x16	If ((SrcB.W > 0.0f) && (SrcA.W == 0.0f)) {
		  Result = 0.0f;
		  SetPredicateReg(Execute);
		}
		Else {
		  Result = SrcA.W + 1.0f;
		  SetPredicateReg(Skip);
		}
			Predicate counter increment greater than; Update predicate register.
			Result replicated in all four channels.
			2 operand; possible coissue.
			See note below.

PRED_SETGE_PUSH	0x17	If ((SrcB.W >= 0.0f) && (SrcA.W == 0.0f)) {
		  Result = 0.0f;
		  SetPredicateReg(Execute);
		}
		Else {
		  Result = SrcA.W + 1.0f;
		  SetPredicateReg(Skip);
		}
			Predicate counter increment greater than equal; Update predicate register.
			Result replicated in all four channels.
			2 operand; possible coissue.
			See note below.

KILLE	0x18	If (SrcA == SrcB) {
		  Result = 1.0f;
		  Killed = TRUE;
		}
		Else
		  Result = 0.0f;
			Per component pixel kill equal; Set kill bit.
			2 operand; possible coissue.
			See note below.

KILLGT	0x19	If (SrcA > SrcB) {
		  Result = 1.0f;
		  Killed = TRUE;
		}
		Else
		  Result = 0.0f;
			Per component pixel kill greater than; Set kill bit.
			2 operand; possible coissue.
			See note below.

KILLGE	0x1a	If (SrcA >= SrcB) {
		  Result = 1.0f;
		  Killed = TRUE;
		}
		Else
		  Result = 0.0f;
			Per component pixel kill greater than equal; Set kill bit.
			2 operand; possible coissue.
			See note below.

KILLNE	0x1b	If (SrcA != SrcB) {
		  Result = 1.0f;
		  Killed = TRUE;
		}
		Else
		  Result = 0.0f;
			Per component pixel kill equal; Set kill bit.
			2 operand; possible coissue.
			See note below.

DST	0x1c	Result.W = SrcB.W;
		Result.Z = SrcA.Z;
		Result.Y = SrcA.Y * SrcB.Y;
		Result.X = 1.0f;
			Computes distance vector.
			2 operand; possible coissue.

MOVA	0x1d	Result = MAX(SrcA, SrcB);
		SQResultF = FLOOR(SrcA.W + 0.5f);
		If (SQResultF >= -256.0f) {
		  SQResultF = SQResultF;
		}
		Else {
		  SQResultF = -256.0f;
		}
		If (SQResultF > 255.0f) {
		  SQResultF = -256.0f;
		}
		SQResultI = truncate_to_int(SQResultF);
		ExportToSQ(SQResultI); // signed 9-bit integer
			Round to the nearest 'integer'; Export to SQ address register.
			2 operand; possible coissue.
			See note below.

	0x1e	Not Used	Not Used
	0x1f	Not Used	Not Used

The following table represents the ALU operations/opcodes supported by the Scalar unit.

Name	Opcode	Function	Notes
ADD	0x00	Result = SrcC.W + SrcC.X;	Scalar addition. Result replicated in all four channels.
ADD_PREV	0x01	Result = SrcC.W + PreviousScalar;	Scalar addition with PreviousScalar. Result replicated in all four channels.
MUL	0x02	Result = SrcC.W * SrcC.X;	Scalar multiply. Result replicated in all four channels.
MUL_PREV	0x03	Result = SrcC.W * PreviousScalar;	Scalar multiply with PreviousScalar. Result replicated in all four channels.

MUL_PREV2	0x04	If ((PreviousScalar == -MAX_FLOAT) ||
		    (PreviousScalar == -INFINITY) ||
		    (PreviousScalar is NaN) ||
		    (SrcC.X <= 0.0f) ||
		    (SrcC.X is NaN)) {
		  Result = -MAX_FLOAT;
		}
		Else {
		  Result = SrcC.W * PreviousScalar;
		}
			Scalar multiply (2) with PreviousScalar.
			Result replicated in all four channels.
			It is mostly/only used when emulating LIT instruction.

MAX	0x05	If (SrcC.W >= SrcC.X)
		  Result = SrcC.W;
		Else
		  Result = SrcC.X;
			Scalar maximum.
			Result replicated in all four channels.

MIN	0x06	If (SrcC.W < SrcC.X)
		  Result = SrcC.W;
		Else
		  Result = SrcC.X;
			Scalar minimum.
			Result replicated in all four channels.

SETE	0x07	If (SrcC.W == 0.0f)
		  Result = 1.0f;
		Else
		  Result = 0.0f;
			Scalar set equal.
			Result replicated in all four channels.

SETGT	0x08	If (SrcC.W > 0.0f)
		  Result = 1.0f;
		Else
		  Result = 0.0f;
			Scalar set greater than.
			Result replicated in all four channels.

SETGE	0x09	If (SrcC.W >= 0.0f)
		  Result = 1.0f;
		Else
		  Result = 0.0f;
			Scalar set greater than equal.
			Result replicated in all four channels.

SETNE	0x0a	If (SrcC.W != 0.0f)
		  Result = 1.0f;
		Else
		  Result = 0.0f;
			Scalar set not equal.
			Result replicated in all four channels.

FRACT	0x0b	Result = SrcC.W + -FLOOR(SrcC.W);	Scalar 'fractional' part of SrcC.W. Result replicated in all four channels.
TRUNC	0x0c	Result = trunc(SrcC.W);	Scalar 'integer' part of SrcC.W. Result replicated in all four channels.

FLOOR	0x0d	Result = TRUNC(SrcC.W);
		If ((SrcC.W < 0.0f) && (SrcC.W != Result))
		  Result += -1.0f;
			Scalar floor function.
			Result replicated in all four channels.

EXP_IEEE	0x0e	If (SrcC.W == 0.0f) {
		  Result = 1.0f;
		}
		Else {
		  Result = Approximate2ToX(SrcC.W);
		}
			Scalar Base2 exponent function.
			Result replicated in all four channels.

LOG_CLAMPED	0x0f	If (SrcC.W == 1.0f) {
		  Result = 0.0f;
		}
		Else {
		  Result = LOG_IEEE(SrcC.W);
		}
		// clamp result
		if (Result == -INFINITY) {
		  Result = -MAX_FLOAT;
		}
			Scalar Base2 log function.
			Result replicated in all four channels.

LOG_IEEE	0x10	If (SrcC.W == 1.0f) {
		  Result = 0.0f;
		}
		Else {
		  Result = ApproximateLog2(SrcC.W);
		}
			Scalar Base2 log function.
			Result replicated in all four channels.

RECIP_CLAMPED	0x11	If (SrcC.W == 1.0f) {
		  Result = 1.0f;
		}
		Else {
		  Result = RECIP_IEEE(SrcC.W);
		}
		// clamp result
		if (Result == -INFINITY) {
		  Result = -MAX_FLOAT;
		}
		if (Result == +INFINITY) {
		  Result = +MAX_FLOAT;
		}
			Scalar reciprocal.
			Result replicated in all four channels.

RECIP_FF	0x12	If (SrcC.W == 1.0f) {
		  Result = 1.0f;
		}
		Else {
		  Result = RECIP_IEEE(SrcC.W);
		}
		// clamp result
		if (Result == -INFINITY) {
		  Result = -ZERO;
		}
		if (Result == +INFINITY) {
		  Result = +ZERO;
		}
			Scalar reciprocal.
			Result replicated in all four channels.

RECIP_IEEE	0x13	If (SrcC.W == 1.0f) {
		  Result = 1.0f;
		}
		Else {
		  Result = ApproximateRecip(SrcC.W);
		}
			Scalar reciprocal.
			Result replicated in all four channels.

RECIPSQRT_CLAMPED	0x14	If (SrcC.W == 1.0f) {
		  Result = 1.0f;
		}
		Else {
		  Result = RECIPSQRT_IEEE(SrcC.W);
		}
		// clamp result
		if (Result == -INFINITY) {
		  Result = -MAX_FLOAT;
		}
		if (Result == +INFINITY) {
		  Result = +MAX_FLOAT;
		}
			Scalar reciprocal square root.
			Result replicated in all four channels.

RECIPSQRT_FF	0x15	If (SrcC.W == 1.0f) {
		  Result = 1.0f;
		}
		Else {
		  Result = RECIPSQRT_IEEE(SrcC.W);
		}
		// clamp result
		if (Result == -INFINITY) {
		  Result = -ZERO;
		}
		if (Result == +INFINITY) {
		  Result = +ZERO;
		}
			Scalar reciprocal square root.
			Result replicated in all four channels.

RECIPSQRT_IEEE	0x16	If (SrcC.W == 1.0f) {
		  Result = 1.0f;
		}
		Else {
		  Result = ApproximateRecipSqrt(SrcC.W);
		}
			Scalar reciprocal square root.
			Result replicated in all four channels.

MOVA	0x17	Result = MAX(SrcC);
		SQResultF = FLOOR(SrcC.W + 0.5f);
		If (SQResultF >= -256.0f) {
		SQResultF = SQResultF;
		}
		Else {
		SQResultF = -256.0f;
		}
		If (SQResultF > 255.0f) {
		SQResultF = -256.0f;
		}
		SQResultI = truncate_to_int(SQResultF);
		ExportToSQ(SQResultI); // signed 9-bit integer
			Round to the nearest 'integer'; Export to SQ address register. Result replicated in all four channels.
			See note below.

MOVA_FLOOR	0x18	Result = MAX(SrcC);
		SQResultF = FLOOR(SrcC.W);
		If (SQResultF >= -256.0f) {
		SQResultF = SQResultF;
		}
		Else {
		SQResultF = -256.0f;
		}
		If (SQResultF > 255.0f) {
		SQResultF = -256.0f;
		}
		SQResultI = truncate_to_int(SQResultF);
		ExportToSQ(SQResultI); // signed 9-bit integer
			Floor; Export to SQ address register.
			Result replicated in all four channels.
			See note below.

SUB	0x19	Result = SrcC.W + -SrcC.X;	Scalar subtract. Result replicated in all four channels. This instruction is needed since NEG argument modifier applies to all channels of the source.
SUB_PREV	0x1a	Result = SrcC.W + -PreviousScalar;	Scalar subtract with PreviousScalar. Result replicated in all four channels. This instruction is needed since NEG argument modifier applies to all channels of the source.

PRED_SETE	0x1b	If (SrcC.W == 0.0f) {
		Result = 0.0f;
		SetPredicateReg(Execute);
		}
		Else {
		Result = 1.0f;
		SetPredicateReg(Skip);
		}
			Scalar predicate set equal; Update predicate register.
			Result replicated in all four channels.
			See note below.

PRED_SETNE	0x1c	If (SrcC.W != 0.0f) {
		Result = 0.0f;
		SetPredicateReg(Execute);
		}
		Else {
		Result = 1.0f;
		SetPredicateReg(Skip);
		}
			Scalar predicate set not equal; Update predicate register.
			Result replicated in all four channels.
			See note below.

PRED_SETGT	0x1d	If (SrcC.W > 0.0f) {
		Result = 0.0f;
		SetPredicateReg(Execute);
		}
		Else {
		Result = 1.0f;
		SetPredicateReg(Skip);
		}
			Scalar predicate set greater than; Update predicate register.
			Result replicated in all four channels.
			See note below.

PRED_SETGE	0x1e	If (SrcC.W >= 0.0f) {
		Result = 0.0f;
		SetPredicateReg(Execute);
		}
		Else {
		Result = 1.0f;
		SetPredicateReg(Skip);
		}
			Scalar predicate set greater than equal; Update predicate register.
			Result replicated in all four channels.
			See note below.

PRED_SET_INV	0x1f	If (SrcC.W == 1.0f) {
		Result = 0.0f;
		SetPredicateReg(Execute);
		}
		Else {
		If (SrcC.W == 0.0f) {
		Result = 1.0f;
		}
		Else {
		Result = SrcC.W;
		}
		SetPredicateReg(Skip);
		}
			Scalar predicate counter invert; Update predicate register. Result replicated in all four channels.
			See note below.

PRED_SET_POP	0x20	Result = SrcC.W + -1.0f;
		
		If (Result <= 0.0f) {
		Result = 0.0f;
		SetPredicateReg(Execute);
		}
		Else {
		Result = Result;
		SetPredicateReg(Skip);
		}
			Scalar predicate counter pop; Update predicate register. Result replicated in all four channels.
			See note below.

PRED_SET_CLR	0x21	Result = +MAX_FLOAT;
		SetPredicateReg(Skip);
			Scalar predicate counter clear; Update predicate register. Result replicated in all four channels.
			See note below.

PRED_SET_RESTORE	0x22	If (SrcC.W == 0.0f) {
		Result = 0.0f;
		SetPredicateReg(Execute);
		}
		Else {
		Result = SrcC.W;
		SetPredicateReg(SKIP);
		}
			Scalar predicate counter restore; Update predicate register. Result replicated in all four channels.
			See note below.

KILLE	0x23	If (SrcC.W == 0.0f) {
		Result = 1.0f;
		Killed = TRUE;
		}
		Else
		Result= 0.0f;
			Scalar pixel kill equal; Set kill bit.
			Result replicated in all four channels.
			See note below.

KILLGT	0x24	If (SrcC.W > 0.0f) {
		Result = 1.0f;
		Killed = TRUE;
		}
		Else
		Result= 0.0f;
			Scalar pixel kill greater than; Set kill bit.
			Result replicated in all four channels.
			See note below.

KILLGE	0x25	If (SrcC.W >= 0.0f) {
		Result = 1.0f;
		Killed = TRUE;
		}
		Else
		Result= 0.0f;
			Scalar pixel kill greater than equal; Set kill bit.
			Result replicated in all four channels.
			See note below.

KILLNE	0x26	If (SrcC.W != 0.0f) {
		Result = 1.0f;
		Killed = TRUE;
		}
		Else
		Result= 0.0f;
			Scalar pixel kill not equal; Set kill bit.
			Result replicated in all four channels.
			See note below.

KILLONE	0x27	If (SrcC.W == 1.0f) {
		Result = 1.0f;
		Killed = TRUE;
		}
		Else
		Result= 0.0f;
			Scalar pixel kill one; Set kill bit.
			Result replicated in all four channels.
			See note below.

SQRT_IEEE	0x28	If (SrcC.W == 1.0f) {
		Result = 1.0f;
		}
		Else {
		Result = ApproximateSqrt(SrcC.W);
		}
			Scalar square root.
			Result replicated in all four channels.
			Useful for normal compression.

	0x29	Not Used	Not Used
MUL_CONST_0 *	0x2a	Result = SrcC_Const.W * SrcGPR.X	Scalar multiply with Constant, SrcGPR lsb 0. Result replicated in all four channels. See note below.

MUL_CONST_1 *	0x2b	Result = SrcC_Const.W * SrcGPR.X	Scalar multiply with Constant, SrcGPR lsb 1. Result replicated in all four channels. See note below.
ADD_CONST_0 *	0x2c	Result = SrcC_Const.W + SrcGPR.X	Scalar addition with Constant, SrcGPR lsb 0. Result replicated in all four channels. See note below.
ADD_CONST_1 *	0x2d	Result = SrcC_Const.W + SrcGPR.X	Scalar addition with Constant, SrcGPR lsb 1. Result replicated in all four channels. See note below.
SUB_CONST_0 *	0x2e	Result = SrcC_Const.W + -SrcGPR.X	Scalar subtraction with Constant, SrcGPR lsb 0. Result replicated in all four channels. See note below.
SUB_CONST_1 *	0x2f	Result = SrcC_Const.W + -SrcGPR.X	Scalar subtraction with Constant, SrcGPR lsb 1. Result replicated in all four channels. See note below.
SIN	0x30	Result = ApproximateSin(SrcC.W);	Scalar sin function. Valid input domain [-PI, +PI] Result replicated in all four channels.
COS	0x31	Result = ApproximateCos(SrcC.W);	Scalar cos function. Valid input domain [-PI, +PI] Result replicated in all four channels.
	0x32 - 0x3f	14 Not Used	14 Not Used

Predicate instructions (PRED_*)
In the case of a predicate instruction, the predicate register update always happens and cannot be destination masked. For more specifics on this, please refer to the Sequencer's architecture specification document.

The vector predicate instructions expect the predicate counter to be in SrcA.W and the predicate condition to be in SrcB.W with Result representing and updated predicate counter.

The scalar predicate instructions expect (depending on the instruction) the predicate counter to be in SrcC.W and the predicate condition to be in SrcC.W with Result representing and updated predicate counter.

Kill instructions (KILL*)
The result is used to determine the visibility of the pixel. A 1.0f indicates that the pixel is made not visible (i.e. it's killed). Once a pixel is killed, subsequent instructions cannot 'unkill' it. Note that vector pipe kill instruction compare is a per channel compare. As such, if any of the compares yields a result of 1.0f then the kill bit is set. Updating of the kill bit always happens and cannot be destination masked.

Mova instructions (MOVA*)
In the case of a mova instruction, the address register update always happens and cannot be destination masked. For more specifics on this, please refer to the Sequencer's architecture specification document.

Predicate, Kill and Mova coissue rules
Predicate and kill instructions should not be coissued together in any fashion because they share an update path. Doing so will result in undefined behavior, but not a chip hang.

Mova should not be coissue with another mova. Doing so will result in undefined behavior, but not a chip hang. Mova may be coissued with predicate or kill instructions.

Because the internal register updates cannot be destination masked, these instructions typically make poor candidates for benign vector or scalar instruction slot fillers.

Scalar operations with constant instructions (*_CONST_*)
These instructions are primarily provided to allow for more coissue opportunities in the scalar pipe. Certain vector instructions may be able to be decomposed in a number of scalar operations using these instructions.

These instructions operate and are programmed slightly differently than other scalar operations.

The follow code show how SrcGPR is selected:

```
SrcGPR =        (((SRC C Swizzle >> 4) & 0x3) << 4) |
                (((SRC C Swizzle >> 2) & 0x3) << 2) |
                (((SRC C Select) & 0x1) << 1) |
                (((Scalar Opcode) & 0x1) << 0);
```

Note that **SRC C Select** is used to select SrcGPR. This means SrcC is assumed to be a constant by the HW when using these opcodes

Like PreviousScalar, SrcGPR does not have input modifiers applied to it nor is indexed addressing permitted. No such restrictions apply to SrcC_Const.

Similar to other scalar instructions, the W channel swizzle of **SRC C Swizzle** selects the source channel for SrcC_Const and the X channel of **SRC C Swizzle** selects the source channel for SrcGPR.

R400 numerics and instruction details

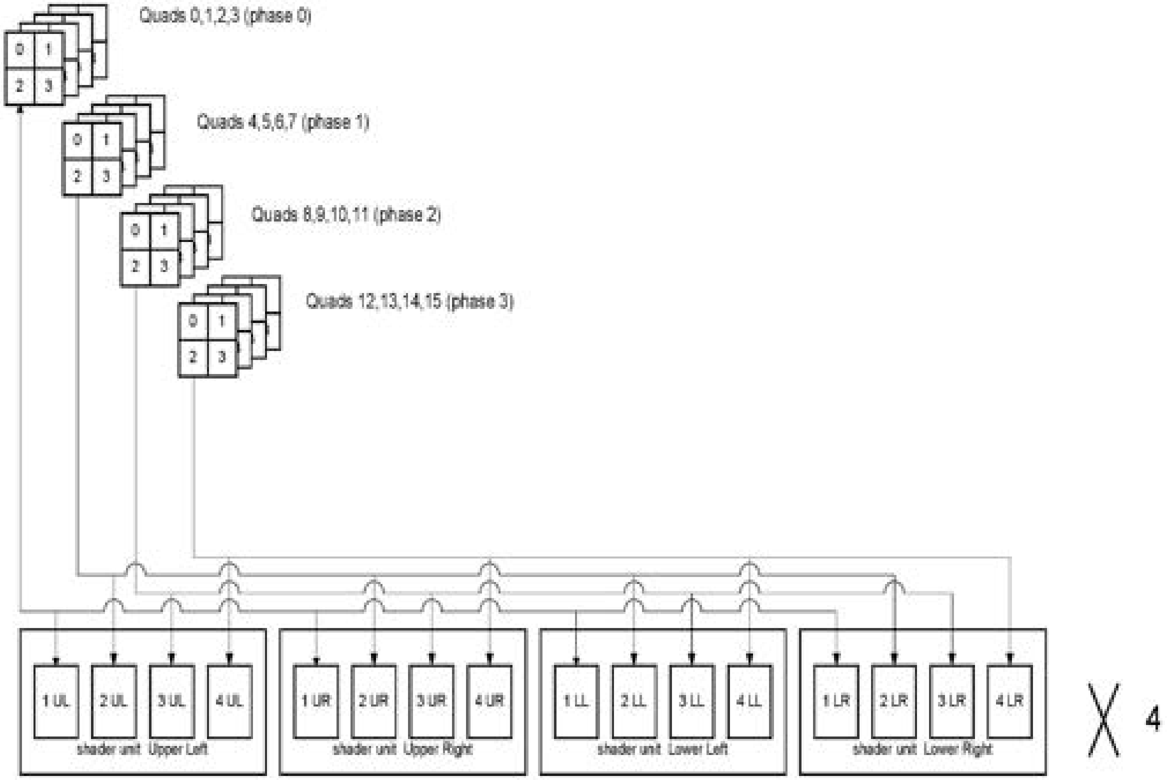
The R400 Floating Point Numerics document (R400numerics.doc) details how the R400 handles NaNs, INFINITYs, denorms along with explaining how out of domain inputs should be handled for instructions like LOG*. Also documented are conversion rules and rounding conventions. Notation, arithmetic and logical truths for the R400 are as also established here.

In addition, a more complete pseudo-coded implementation of the R400 shader pipe and its instructions are provided along with important macro instruction expansions.

4. Shader Block Diagrams

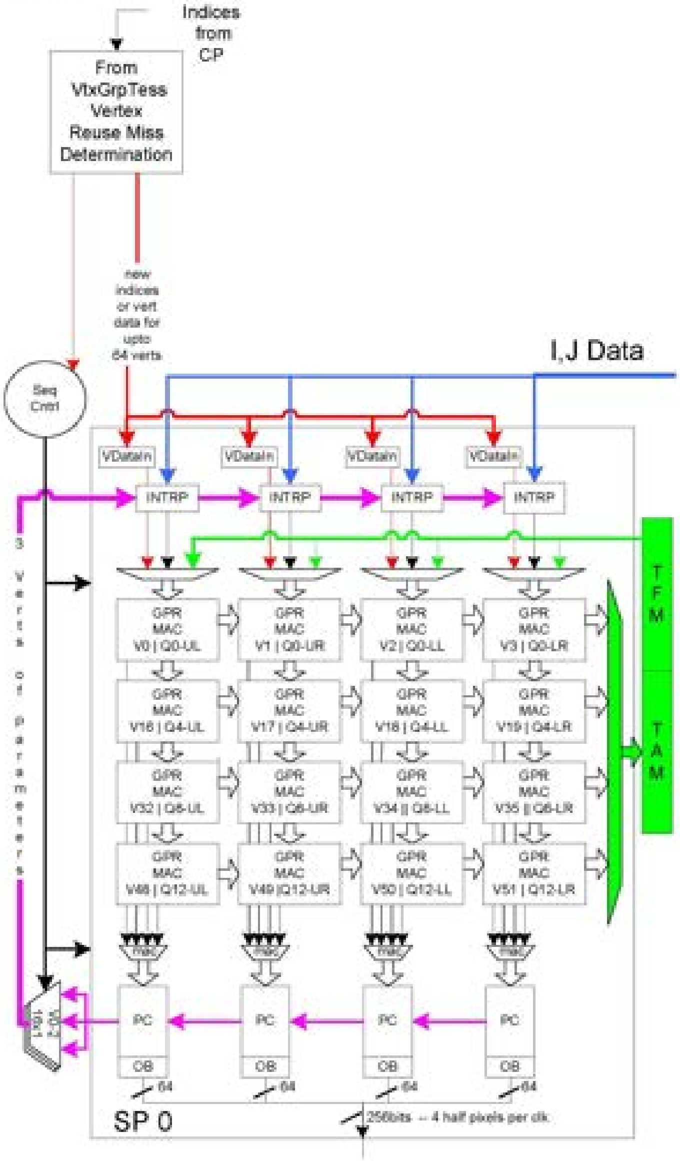
4.1 Shader as an SIMD architecture

As shown in the diagram below, four identical processing units comprise a shader unit. There are four shader units in one shader pipeline. R400 has four shader pipelines. The full R400 shader pipe represents an example of a SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata streams) architecture: the four shader pipelines, each with 4 identical shader units, executing the same pair of Vector/Scalar Instruction on different data streams, in this case different pixel positions within the same quad, over four different quads of pixels in parallel. In figure, the shader units are named as Upper Left, Upper Right, Lower Left and Lower Right based on the relative position within the quad of the pixels they process. Within one shader pipe, only 4 processing units, one from each shader unit, are in the same execution sequence (phase) at a given time. So, through the whole chip we have 4 sets of 16 processing units being at different execution phases from one set to the other, but in the same execution phase within the same set.



4.2 Top-Level Diagram of a Shader Pipeline

The diagram below represents a high level description of a shader pipe. The major data streams coming into or going out of it are clearly shown.



5. Interfaces

5.1 External Interfaces

5.1.1 *Naming Convention*

TP -> stands for Texture Pipe.
SP-> stands for Shader Pipe
SQ->stands for Sequencer Unit
SX->stands for Shader Export
SC->stands for Scan Converter

When an X is used as a postfix in the Direction colomn of the interface definition tables, it means that the bus is a broadcast bus to all units with the same name. For example, if the direction of the bus is defined as SQ->SPx, that means that Sequencer is broadcasting the same information to all Shader Pipe instances.

5.1.2 Shader Engine to Texture Fetch Unit Bus

Four quad's worth of addresses is transferred to Fetch Unit every clock. These are sourced from a different pixel within each of the sub-engines repeating every 4 clocks. The register file index to read must precede the data by 2 clocks. The Read address associated with Quad 0 must be sent 1 clock after the Instruction Start signal is sent, so that data is read 3 clocks after the Instruction Start.

Four Quad's worth of Fetch Data may be written to the Register file every clock. These are directed to a different pixel of the sub-engines repeating every 4 clocks. The register file index to write must accompany the data. Data and Index associated with the Quad 0 must be sent 3 clocks after the Instruction Start signal is sent.

Name	Direction	Bits	Description
SP0_TP0_fetch_addr	SP0->TP0	384	3 Fetch Addresses read from the Register file
TP0_SP0_data	TP0→SP0	512	4 texture results
SP1_TP1_fetch_addr	SP1->TP1	384	3 Fetch Addresses read from the Register file
TP1_SP1_data	TP1→SP1	512	4 texture results
SP2_TP2_fetch_addr	SP2->TP2	384	3 Fetch Addresses read from the Register file
TP2_SP2_data	TP2→SP2	512	4 texture results
SP3_TP3_fetch_addr	SP3->TP3	384	3 Fetch Addresses read from the Register file
TP3_SP3_data	TP3→SP3	512	4 texture results
TPx_SPx_gpr_dst	TPx→SPx	7	Write address into the gprs
TPx_SPx_gpr_cmask	TPx→SPx	4	Channel mask. Supports the ability to mask any of the 32 bit channel of the fetch return data

5.1.3 Sequencer to Shader Pipe(s): Texture stall

Texture pipe signals the Sequencer that its input buffer is full. The Sequencer asserts SQ_SPx_fetch_stall so that Shader Pipe does not send new requests to the Texture Pipe.

Name	Direction	Bits	Description
SQ_SPx_fetch_stall	SQ→SPx	1	Do not send more texture requests if asserted

5.1.4 ScanConverter to Shader Pipe: IJ bus

This is a bus that sends the IJ information to the IJ fifos on the top of each shader pipe. At the same time the control information goes to the sequencer. There are 4 of these buses over the whole chip (SP0 thru 3)

Name	Direction	Bits	Description
SC_SP0_data	SC→SP0	64	IJ information sent over 2 clocks (or XY info sent over 1 clock in the lower 24 LSBs of the interface)
SC_SP0_q_wr_mask	SC→SP0	1	Write Mask
SC_SP0_dest	SC→SP0	1	Controls the write destination (XY buffer, IJ buffer)
SC_SP1_data	SC→SP1	64	IJ information sent over 2 clocks (or XY info sent over 1 clock in the lower 24 LSBs of the interface)
SC_SP1_q_wr_mask	SC→SP1	1	Write Mask
SC_SP1_dest	SC→SP1	1	Controls the write destination (XY buffer, IJ buffer)
SC_SP2_data	SC→SP2	64	IJ information sent over 2 clocks (or XY info sent over 1 clock in the lower 24 LSBs of the interface)
SC_SP2_q_wr_mask	SC→SP2	1	Write Mask
SC_SP2_dest	SC→SP2	1	Controls the write destination (XY buffer, IJ buffer)
SC_SP3_data	SC→SP3	64	IJ information sent over 2 clocks (or XY info sent over 1 clock in the lower 24 LSBs of the interface)
SC_SP3_q_wr_mask	SC→SP3	1	Write Mask
SC_SP3_dest	SC→SP3	1	Controls the write destination (XY buffer, IJ buffer)
SC_SQ_RTS	SC→SQ	1	SC ready to send data

5.1.5 Sequencer to Shader Pipe(s) - broadcast: Interpolator bus

This bus interface defines all the signals needed to perform the interpolation of the primitive parameters coming from the Parameter Caches via the SX blocks.

Name	Direction	Bits	Description
SQ_SPx_interp_prim_type	SQ→SPx	3	Type of the primitive 000 : Normal 011 : Real Time 100 : Line AA 101 : Point AA 110 : Sprite
SQ_SPx_interp_flat_vtx	SQ→SPx	2	Provoking vertex for flat shading
SQ_SPx_interp_flat_gouraud	SQ→SPx	1	Flat or gouraud shaded interpolation
SQ_SPx_interp_cyl_wrap	SQ→SPx	4	Which channel of the parameter being interpolated needs to be wrapped
SQ_SPx_interp_ijline	SQ→SPx	2	Line in the IJ/XY buffer to use to interpolate
SQ_SPx_interp_buff_swap	SQ→SPx	1	Swap the IJ/XY buffers at the end of the interpolation
SQ_SPx_interp_gen_I0	SQ→SPx	1	Generate I0 or not. This tells the interpolators not to use the parameter cache but rather overwrite the data with interpolated 1 and 0. Overwrite if gen_I0 is high.

5.1.6 Sequencer to Shader Pipe(s)-broadcast: Parameter Cache Read control bus

This interface provides three different pointers specifying the location of the parameter values in the Parameter Caches. Depending on the way the vertices get mapped into primitives, it might happen that the parameter values come from different relative offsets in the parameter caches from one parameter cache to the other across a shader pipe. This is the reason why three different read pointers are specified. This is not true for the write path.
This is a broadcast interface.

Name	Direction	Bits	Description
SQ_SPx_ptr0	SQ→SPx	7	Parameter Pointer into PC
SQ_SPx_ptr1	SQ→SPx	7	Parameter Pointer into PC
SQ_SPx_ptr2	SQ→SPx	7	Parameter Pointer into Parameter Cache
SQ_SPx_pc0_addr_sel	SQ→SPx	2	Selection one of the pointers for parameter cache 0
SQ_SPx_pc1_addr_sel	SQ→SPx	2	Selection one of the pointers for parameter cache 1
SQ_SPx_pc2_addr_sel	SQ→SPx	2	Selection one of the pointers for parameter cache 2
SQ_SPx_pc3_addr_sel	SQ→SPx	2	Selection one of the pointers for parameter cache 3
SQ_SP0_read_ena	SQ→SP0	4	Read enables for the 4 memories in the SP0
SQ_SP1_read_ena	SQ→SP1	4	Read enables for the 4 memories in the SP1
SQ_SP2_read_ena	SQ→SP2	4	Read enables for the 4 memories in the SP2
SQ_SP3_read_ena	SQ→SP3	4	Read enables for the 4 memories in the SP3

5.1.7 Sequencer to Shader Pipe: GPR, Parameter Cache control and auto counter

This interface defines the control mechanism for the GPR read/write paths as well as the Parameter Cache write path.

Name	Direction	Bits	Description
SQ_SPx_wr_addr	SQ→SPx	7	Write address (the same bus used for writing into GPRs or Parameter Cache)
SQ_SPx_gpr_rd_addr	SQ→SPx	7	Read address
SQ_SPx_gpr_re_addr	SQ→SPx	1	Read Enable
SQ_SPx_gpr_we_addr	SQ→SPx	1	Write Enable for the GPRs
SQ_SPx_gpr_phase_mux	SQ→SPx	2	The phase mux (arbitrates between inputs, ALU source reads and writes)

SQ_SPx_channel_mask	SQ→SPx	4	The channel mask
SQ_SP0_pixel_mask	SQ→SP0	4	The pixel mask
SQ_SP1_pixel_mask	SQ→SP1	4	The pixel mask
SQ_SP2_pixel_mask	SQ→SP2	4	The pixel mask
SQ_SP3_pixel_mask	SQ→SP3	4	The pixel mask
SQ_SPx_pc_we_addr	SQ→SPx	1	Write Enable for the parameter caches
SQ_SPx_gpr_input_mux	SQ→SPx	2	When the phase mux selects the inputs this tells from which source to read from: Interpolated data, VTX0, VTX1, autogen counter.
SQ_SPx_index_count	SQ→SPx	12?	Index count, common for all shader pipes

5.1.8 Shader Pipe to Shader Export (SX): Parameter data out of Parameter Cache

There is 512-bit of data (4 x128) coming out of each shader pipe for each read out of the parameter caches. These data gets routed into the interpolation units by the SX blocks.

Name	Direction	Bits	Description
SP0_SX0_data0	SP0→SX0	128	Parameter data 0
SP0_SX0_data1	SP0→SX0	128	Parameter data 1
SP0_SX0_data2	SP0→SX0	128	Parameter data 2
SP0_SX0_data3	SP0→SX0	128	Parameter data 3
SP1_SX1_data0	SP1→SX1	128	Parameter data 0
SP1_SX1_data1	SP1→SX1	128	Parameter data 1
SP1_SX1_data2	SP1→SX1	128	Parameter data 2
SP1_SX1_data3	SP1→SX1	128	Parameter data 3
SP2_SX0_data0	SP2→SX0	128	Parameter data 0
SP2_SX0_data1	SP2→SX0	128	Parameter data 1
SP2_SX0_data2	SP2→SX0	128	Parameter data 2
SP2_SX0_data3	SP2→SX0	128	Parameter data 3
SP3_SX1_data0	SP3→SX1	128	Parameter data 0
SP3_SX1_data1	SP3→SX1	128	Parameter data 1
SP3_SX1_data2	SP3→SX1	128	Parameter data 2
SP3_SX1_data3	SP3→SX1	128	Parameter data 3

5.1.9 Shader Export (SX) to Interpolators: Parameter Cache Return bus

This bus represents the values of a given parameter at the three vertices of the primitive.
Note: The nature of this bus might change in the future depending on where the Parameter Difference engine physically resides (see Section of this document titled "Open Issues").

Name	Direction	Bits	Description
SXx_SPx_vtx_data_0	SXx→SPx	128	Vertex data to interpolate
SXx_SPx_vtx_data_1	SXx→SPx	128	Vertex data to interpolate
SXx_SPx_vtx_data_2	SXx→SPx	128	Vertex data to interpolate

5.1.10 Shader Pipe to Shader Export (SX): Pixel/Vertex write to SX

Name	Direction	Bits	Description
SP0_SX0_export_data	SP0→SX0	256	4 pairs of 32 bits channel values
SP0_SX0_export_dst	SP0→SX0	4	Specifies one of the of up to 12 export destinations
SP1_SX1_export_data	SP1→SX1	256	4 pairs of 32 bits channel values
SP1_SX1_export_dst	SP1→SX1	4	Specifies one of the of up to 12 export destinations
SP2_SX0_export_data	SP2→SX0	256	4 pairs of 32 bits channel values

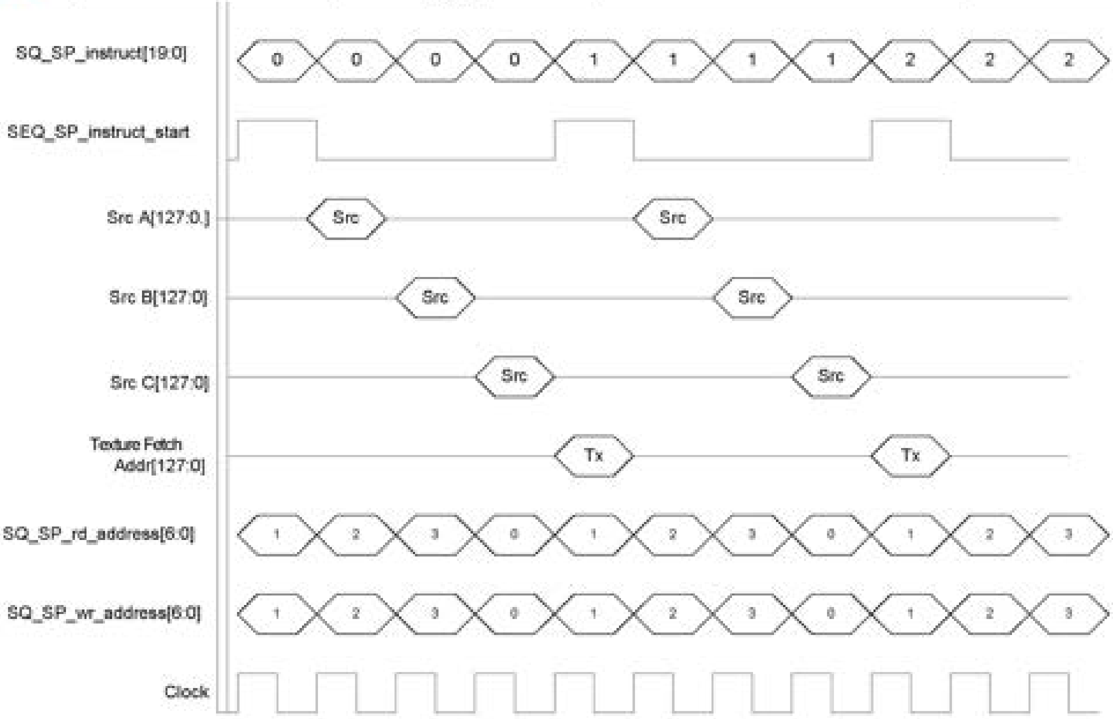
SP2_SX0_export_dst	SP2→SX0	4	Specifies one of the of up to 12 export destinations
SP3_SX1_export_data	SP3→SX1	256	4 pairs of 32 bits channel values
SP3_SX1_export_dst	SP3→SX1	4	Specifies one of the of up to 12 export destinations
SPx_SXx_ export _count	SP0→SX0	3	Each set of four pixels or vectors is exported over eight clocks. This field specifies where the SP is in that sequence.
SPx_SXx_ export _last	SP0→SX0	1	Asserted on the first shader count of the last export of the clause
SP0_SX0_ export _pvalid	SP0→SX0	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SP0_SX0_ export _wvalid	SP0→SX0	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SP1_SX1_ export _pvalid	SP1→SX1	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SP1_SX1_ export_wvalid	SP1→SX1	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SP2_SX0_ export _pvalid	SP2→SX0	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SP2_SX0_ export _wvalid	SP2→SX0	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SP3_SX1_ export _pvalid	SP3→SX1	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SP3_SX1_ export _wvalid	SP3→SX1	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors

5.1.11 *Sequencer to SPx: Instruction Interface*

This is the bus that sends the instruction and constant data to all four Shader pipe instances. Because a new instruction is needed only every 4 clocks, the width of "SQ_SPx_instruct" sub-bus is divided by 4 and both constants and instruction are sent over those 4 clocks. **SRC A (B or C) Select** of SQ_SP_Instruction interface is derived by Sequencer from the **SRC A (B,C) Select** and **SRC A (B,C) Register/Constant Pointer** (upper two bits) of the ALU Instruction word. All the other bit-fields in the SQ_SP_instruct interface bus are explicitly present in the ALU Instruction word. Please refer to Section 3.2 of this document for more details on the R400 ALU instruction word format.

Name	Direction	Bits	Description
SQ_SPx_instruct_start	SQ→SPx	1	Instruction start
SQ_SPx_instruct	SQ→SPx	21	Transferred over 4 cycles 0: SRC A Select 1:0 00: Constant 01: GPR input 10: Previous Vector Result 11: Previous Scalar Result SRC A Negate Argument Modifier 2:2 SRC A Abs Argument Modifier 3:3 SRC A swizzle 11:4 VectorDst 17:12 Unused 20:18 -- 1: SRC B Select 1:0 SRC B Negate Argument Modifier 2:2 SRC B Abs Argument Modifier 3:3

			SRC B swizzle 11:4 ScalarDst 17:12 Unused 20:18 2: SRC C Select 1:0 SRC C Negate Argument Modifier 2:2 SRC C Abs Argument Modifier 3:3 SRC C swizzle 11:4 Unused 20:12 3: Vector Opcode 4:0 Scalar Opcode 10:5 Vector Clamp 11:11 Scalar Clamp 12:12 Vector Write Mask 16:13 Scalar Write Mask 20:17
SQ_SPx_stall	SQ→SPx	1	Stall signal (ALU executes a Max PV,PV and Max PS,PS instruction)
SQ_SPx_export_count	SQ→SPx	3	Each set of four pixels or vectors is exported over eight clocks. This field specifies where the SP is in that sequence.
SQ_SPx_export_last	SQ→SPx	1	Asserted on the first shader count of the last export of the clause
SQ_SP0_export_pvalid	SQ→SP0	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP0_export_wvalid	SQ→SP0	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SQ_SP1_ export_pvalid	SQ→SP1	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP1_ export_wvalid	SQ→SP1	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SQ_SP2_ export_pvalid	SQ→SP2	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP2_ export_wvalid	SQ→SP2	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SQ_SP3_ export_pvalid	SQ→SP3	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP3_ export_wvalid	SQ→SP3	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors



The above diagram attempts to describe the timing relation between the signals at SQ-SP instruction interface. Each instruction (SQ_SP_instruct [19:0]) is broadcasted over four cycles. SQ_SP_instruct_start is asserted at the first cycle of the instruction broadcast. Cycle 0 of SQ_SP_rd_address is "dedicated" to reading SRC A out of the Register File, Cycle 1 to reading SRC B, Cycle 2 to reading SRC C and Cycle 3 to reading Texture Address for a texture fetch request. Cycle 0 of SQ_SP_wr_address is "dedicated" to writing into Register File of Interpolated data from Interpolation units, Cycle 1 to writing of return data from a previously issued texture fetch and Cycle 3 to Previous Vector result and Cycle 3 to Previous Scalar result.

5.1.12 *Shader Pipe to Sequencer: Constant address load*

Name	Direction	Bits	Description
SP0_SQ_const_addr	SP0→SQ	36	Constant address load to the sequencer
SP0_SQ_valid	SP0→SQ	1	Data valid
SP1_SQ_const_addr	SP1→SQ	36	Constant address load to the sequencer
SP1_SQ_valid	SP1→SQ	1	Data valid
SP2_SQ_const_addr	SP2→SQ	36	Constant address load to the sequencer
SP2_SQ_valid	SP2→SQ	1	Data valid
SP3_SQ_const_addr	SP3→SQ	36	Constant address load to the sequencer
SP3_SQ_valid	SP3→SQ	1	Data valid

5.1.13 *Sequencer to SPx: constant broadcast*

The interface represents the constant values interface coming from the Sequencer unit. Constant values can be selected as operands for in a given shader instruction.

Name	**Direction**	**Bits**	**Description**
SQ_SPx_constant	SQ→SPx	128	Constant broadcast

6. Parameter Interpolation

This section was partially copied from Section 15 "IJ Format" of the "R400 Sequencer Specification" document.
There are two key featues in R400 interpolation scheme:

a. Barycentric coordinates are used in the interpolation of all parameters.
b. The interpolation is done at a different precision across a 2x2 colletion of pixels. The parameters of the upper left pixel of the quad are interpolated at full 20x24 mantissa precision. Then the result along with the difference in IJ barycentric coords is used to interpolate the parameters for the remaining pixels of the 2x2.

Assuming P0 is the interpolated parameter at Pixel 0 having the barycentric coordinates I(0), J(0) and so on for P1,P2 and P3. Also assuming that A is the parameter value at V0 (interpolated with I), B is the parameter value at V1 (interpolated with J) and C is the parameter value at V2 (interpolated with (1-I-J).

$$\Delta 01I = I(1) - I(0)$$
$$\Delta 01J = J(1) - J(0)$$
$$\Delta 02I = I(2) - I(0)$$
$$\Delta 02J = J(2) - J(0)$$
$$\Delta 03I = I(3) - I(0)$$
$$\Delta 03J = J(3) - J(0)$$

P0	P1
P2	P3

$$P0 = C + I(0) * (A - C) + J(0) * (B - C)$$
$$P1 = P0 + \Delta 01I * (A - C) + \Delta 01J * (B - C)$$
$$P2 = P0 + \Delta 02I * (A - C) + \Delta 02J * (B - C)$$
$$P3 = P0 + \Delta 03I * (A - C) + \Delta 03J * (B - C)$$

P0 is computed at 20x24 mantissa precision and P1 to P3 are computed at 8x24 mantissa precision. So far no visual degradation of the image was seen using this scheme.

Multiplies (Full Precision): 2
Multiplies (Reduced precision): 6
Subtracts 19x24 (Parameters): 2
Adds: 8

FORMAT OF P0's IJ: Mantissa 20 Exp 4 for I + Sign
Mantissa 20 Exp 4 for J + Sign

FORMAT of Deltas (x3): Mantissa 8 Exp 4 for I + Sign
Mantissa 8 Exp 4 for J + Sign

Total number of bits for one quad worth of IJ data: 20*2 + 8*6 + 4*8 + 4*2 = 128
All numbers are kept using the un-normalized floating point convention: if exponent is different than 0 the number is normalized if not, then the number is un-normalized. The maximum range for the IJs (Full precision) is +/- 63 and the range for the Deltas is +/- 127.

7. Shader Limitations

The sequencer unit and compiler need to be aware of a series of limitations in the shader functionality. These are limitations on the pixel shader functionality as well as on the vertex shader functionality. In reality, the compiler does not need to pay attention to these limitations since sequencer will detect them and react accordingly. However, for compiler optimization reasons, we describe here the various latency issues revolving around our shader pipe implementation.

1) The use of Previous Vector result (PV) and Previous Scalar result (PS) values.
The following sequence is being executed:
ADD R0 = R3, R4.
MUL R2 = R0, R1 and the desired R0 values are the ones coming from the ADD instruction (ie a dependant instruction).
Because of the pipeline latencies involved, the R0 value from the ADD instruction won't be available in GPRs until one instruction later from the moment the MUL instruction enters the execution pipeline. The sequencer will write the same sequence as follows:
ADD R0 = R3, R4.
MUL R2 = PV, R1

Alternatively, if wanted, the compiler can force the use of the PV register by instead using the following instruction:
MUL R2 = PV, R1 (instead of MUL R2 = R0, R1).

The following sequence is being executed:
ADD R0.x = R3, R4.
MUL R2 = R0, R1 and the desired R0 values is the one coming from the ADD instruction.
Because of the pipeline latencies involved, the R0.x value from the ADD instruction won't be available in GPRs until one cycle later from the moment the MUL instruction enters the execution pipeline. The compiler can introduce a NOP instruction in between ADD and MUL, but it does not have to. The sequencer will detect this dependency case and insert a MOV PV, PV on the vector side and a MOV PS, PS on the scalar side as well (MOV PV,PV is the HW translation of a NOP).

As a conclusion: If the Previous Vector Result is used explicitly in the code, then the instruction will be executed as is. If a dependant use of a masked register is done instead, the sequencer is going to introduce a NOP between the two instructions in order to achieve the right behavior.

2) There is a one-instruction load-use delay between a MOVA instruction and use of a 'indexed' constant. If this delay cannot be honored, the MOVA instruction should be followed by a MOV instruction.

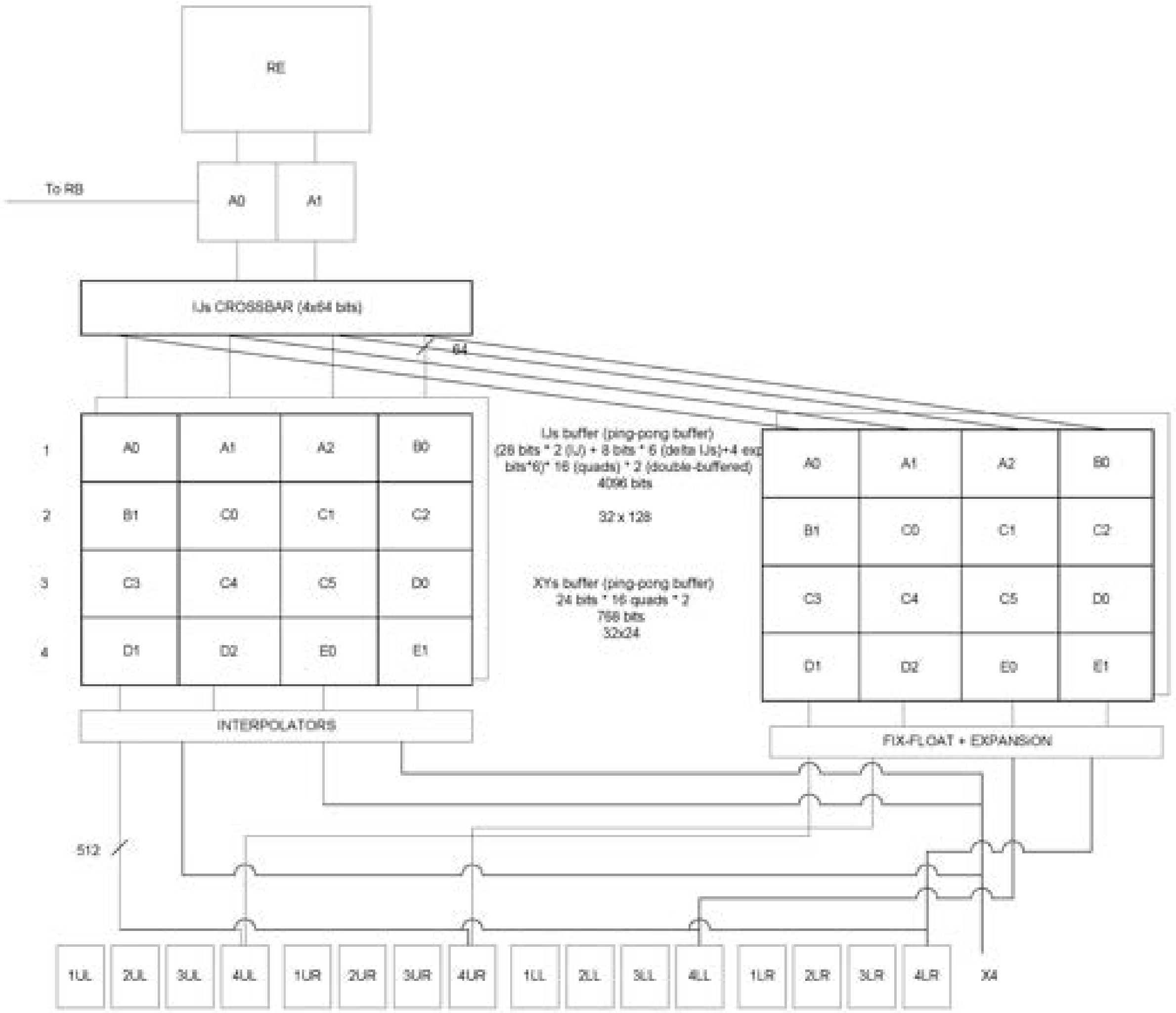
3) General Coissue limitations. A scalar instruction may not be coissued with a 3-argument vector instruction.

8. Hardware Implementation Specifics

8.1 General Information on the Shader Floating Point arithmetic

Two special cases of floating-point numbers are recognized: zero is defined as any number with a zero exponent and infinity (or NaN) is defined as any number with an exponent of 255. The mantissa is cleared for both input and output values of zero and infinity while the sign bit is cleared for input and output zeros. The result of a multiplication involving zero is always defined as zero. The result of any addition involving infinity is always defined as infinity, with the added stipulation that any addition involving negative infinity always results in negative infinity. The Vector Engine/Scalar Engine works with standard IEEE floating-point numbers although all math operations are performed without rounding.

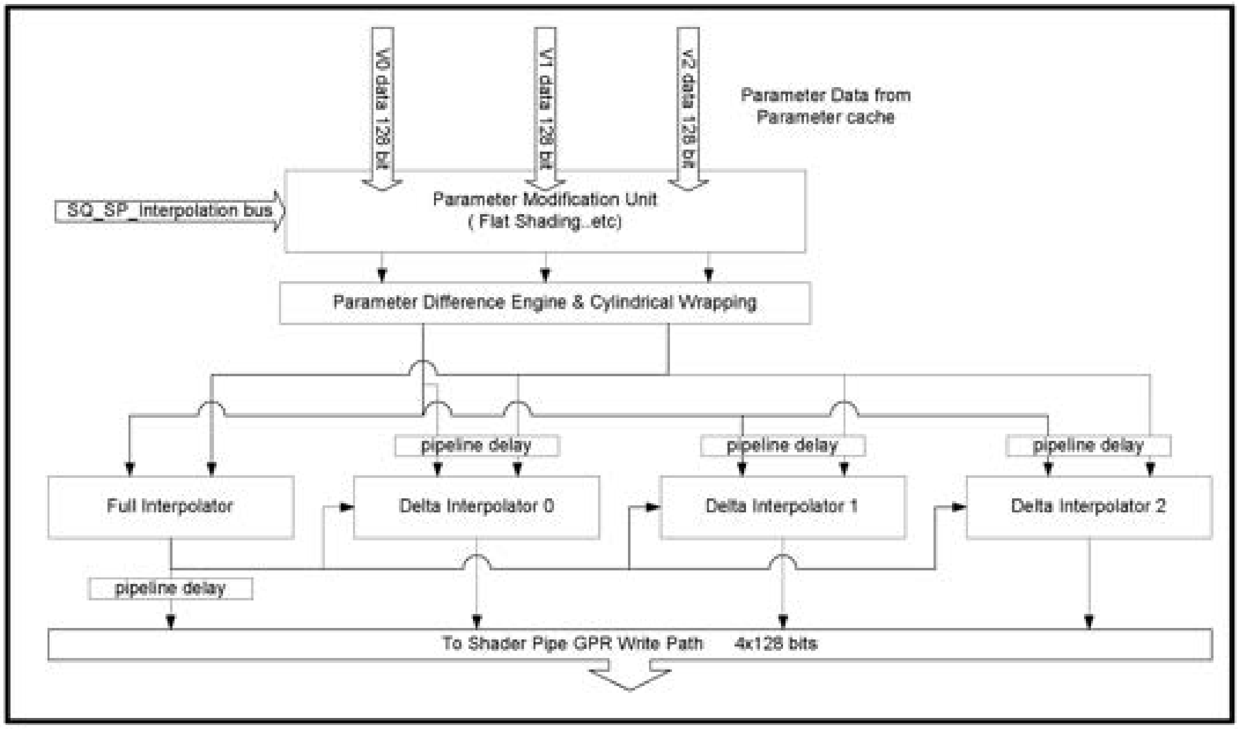
8.2 Interpolators and IJ/XY Buffers



The above diagram represents a high level description of IJ and XY fifos, as well as Parameter Interpolation unit (Interpolators). The double buffer on the left represents the IJ fifo. Each of the cells in that buffer represents a quad of pixels worth of IJ interpolation data. The double buffer on the right represents the XY Address fifo. There is 512 bits of data transferred from the interpolators into each shader pipe (GPRs) per cycle, a totall of 2048 bits across the whole chip. The reading and writing of IJ/XY fifos is controlled by the sequencer via the "SQ to SP: Intepolator bus" interface described in the Section 5.1.4 of this document.

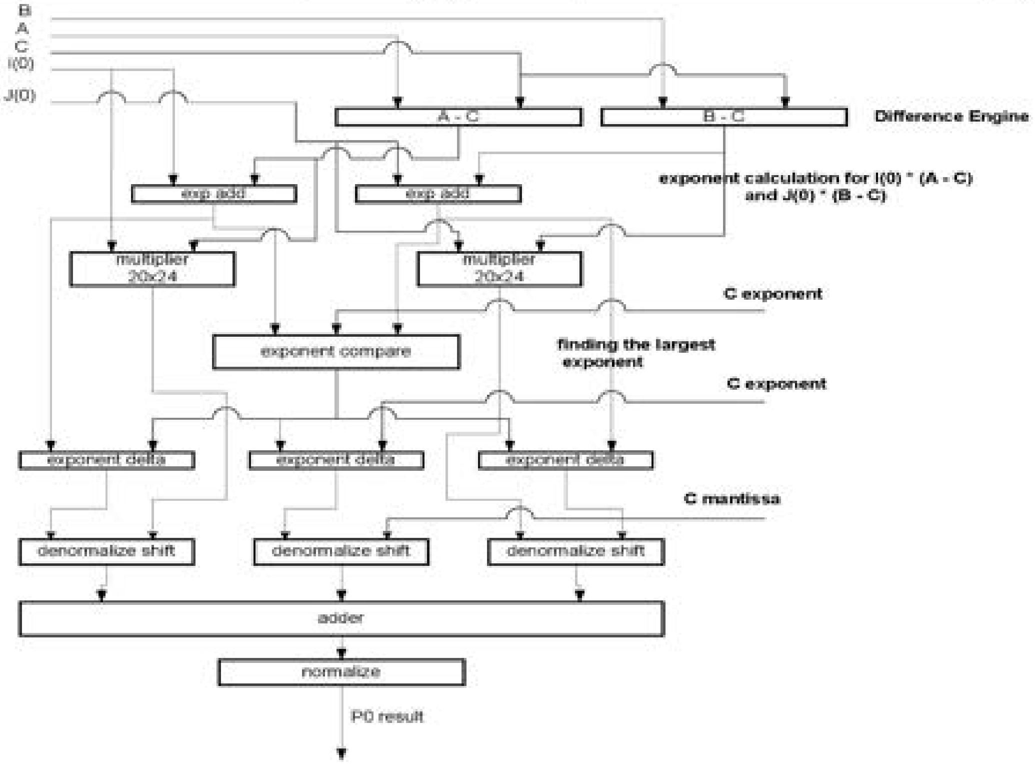
8.2.1 *Interpolators*

The following diagram describes the interpolation unit for one shader pipe. There are four instances of the same in the R400 architecture. Parameter Difference Engine calculates the difference between the values of a given parameter at the vertices. These deltas are inputs into the four interpolators shown below. IJ intepolation data from the IJ fifos would be the other required input into the interpolators. The first interpolation unit (High Precision Interpolator) is used to calculate the interpolated values for Pixel 0, the upper left pixel of the quad. The result of this interpolator is then fed into the lower precision interpolator (Delta Interpolators) used to calculate the parameter values for the other three pixels of the quad. In Section 6 of this document you will find the mathematical description of the barycentric interpolation equations.

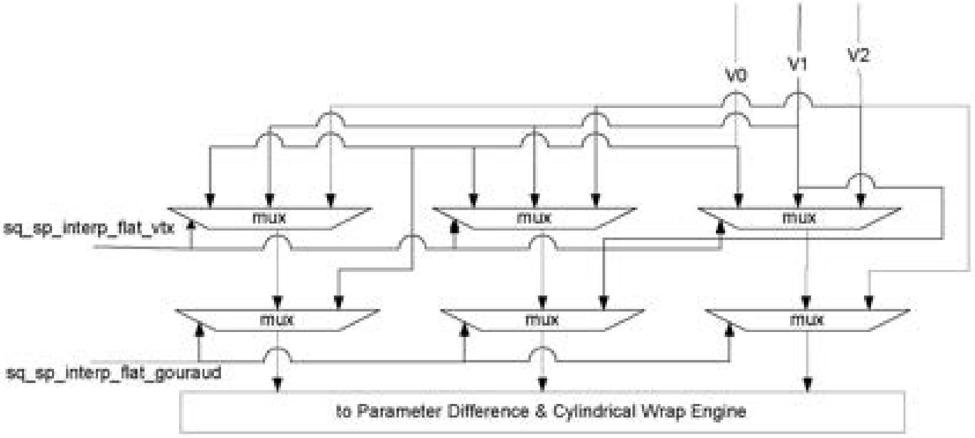


8.2.1.1 Interpolation Units

The following diagram describes the data path for a high precision interpolation unit for one 32-bit channel of the parameter data. The diagram does not exactly describe the hardware implementation of the interpolator. In order to simplify the diagram, three denormalize shifters are shown. In reality, only two out of three mantissa terms about to be added, need to be denormalized to the third mantissa term with the largest exponent. The "exponent compare" unit finds the largest exponent of the following three terms: C, I(0) (A-C) and J(0)*(B-C).



8.2.1.2 Parameter Selection Unit



V0
V1
V2
mux
mux
mux
sq_sp_interp_flat_vtx
mux
mux
mux
sq_sp_interp_flat_gouraud
to Parameter Difference & Cylindrical Wrap Engine

The Parameter Modification unit described in the above diagram is used to preprocess the vertex parameter data coming from the Parameter Cache unit before they enter the Difference engine. It's in this block that the selection of the provoking vertex parameter value, for flat shading, is done. The pertaining control signals are part of the "SQ to SP: Interpolator bus" interface described in Section 5.1.5 of this document.

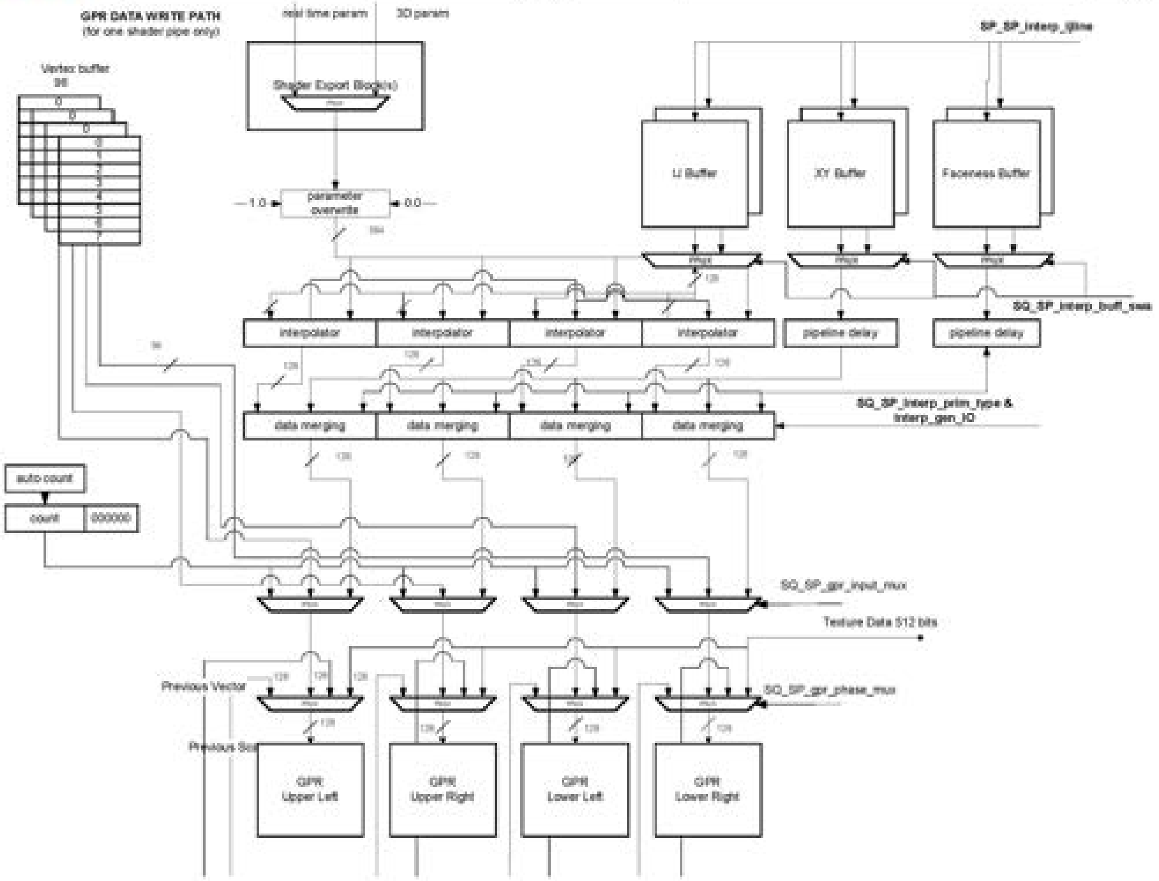
8.2.1.3 Parameter Difference & Cylindrical Wrap Engine

The purpose of the unit is to calculate the delta differences between the parameter values at vertices. These deltas are then used as input values into the interpolator units. Also, it's in this unit that cylindrical wrap adjustment of the parameter values is done. All the functions are implemented on per channel basis. One alternative solution would be for the delta differences between the parameter values to be done in the SX blocks. This way we can cut down from 4 channels * 2 substracts/channel * 4 pipes = 32 subtractors to 8 if placed in SX blocks. This may complicate the Cylindrical Wrap implementation, which in itself requires a bunch of subtracts. The control/state signals are part of the "SQ_SP: Interpolator Bus" interface. Please refer to Section 5.1.5 of this document for a detailed definition of this control interface.

8.2.2 GPR Write Path

The diagram below shows all the possible data paths going into the GPR write paths, their selection and routing.
The drawing shows four GPR units representing any four GPR units of a shader pipeline that are in the same phase.
From the pipeline-timing point of view, interpolated parameters of pixels belonging to the same quad are flowing through the data paths described in the drawing at any given time. As it was mentioned before, the interpolators have two sets of inputs, the IJ data coming out of the IJ buffers and vertex parameter data coming from the parameter caches via the SX (Shader Export) blocks. The SX blocks are responsible for multiplexing between the real time parameters and vertex parameter data coming from the parameter caches. The outputs from the interpolators get merged into 128 vectors with data coming from XY and Faceness buffers. The next level of muxing controlled by SQ_SP_gpr_input_mux part of the "SQ_SP: Interpolation bus" interface is used to route between vertex data/indices and interpolated pixel parameters. The next and last level of multiplexing is used to multiplex the writing into GPRs of texture fetch return data, interpolated pixel data, previous scalar and previous vector result.
The truth table below describes all the possible "merge" combinations between the interpolated, XY and Faceness data based on the **SQ_SP_interp_prim_type** and **SQ_SP_interp_gen_IO** signals found in "SQ_SP: Interpolation bus" interface. The above signals are also used to control the overwrite of the parameter values coming from the parameter cache with constants 0.0 and 1.0 when doing expansion for point sprite primitive types.

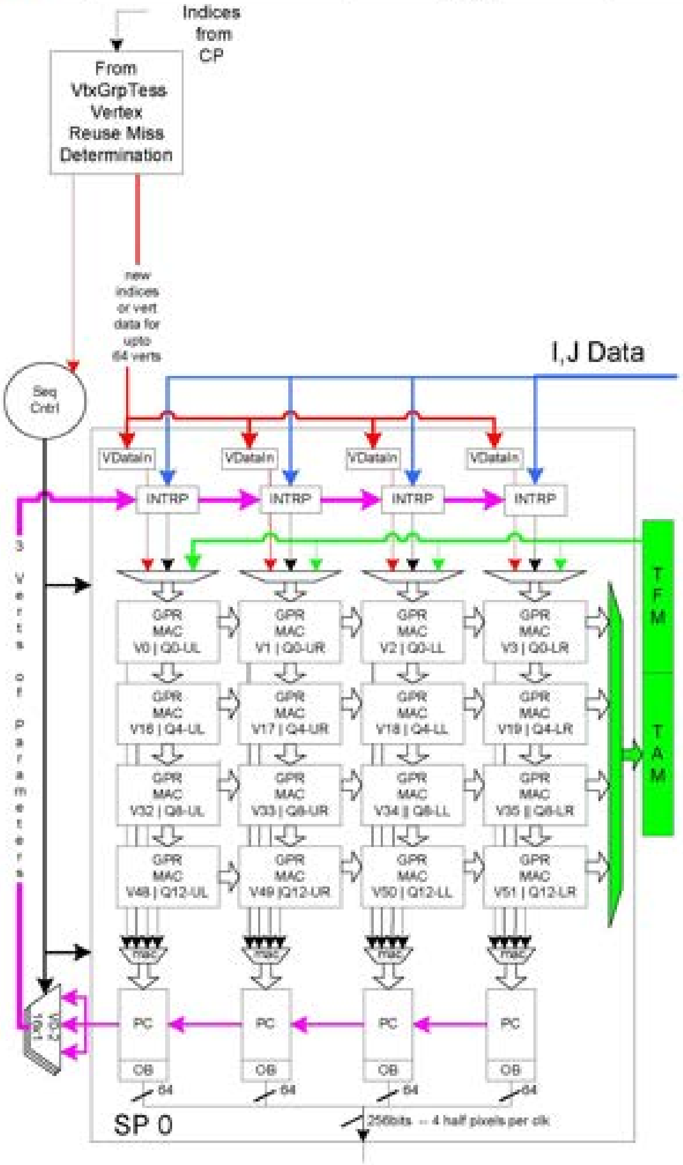
SQ_SP_interp_gen_IO	**SQ_SP_param_type[2]**	**Merge Logic Result**
0	0	Interpolated data
0	1	Interpolated data
1	0	X, Y, don't care, faceness
1	1	X, Y, S, T



8.3 Vector Unit

8.3.1 *Vector Unit Pipeline*

The diagram below describes a complete set of the Vector Units present in a single shader pipeline with interpolator units at the top of the pipeline and the parameter caches at the bottom of it.

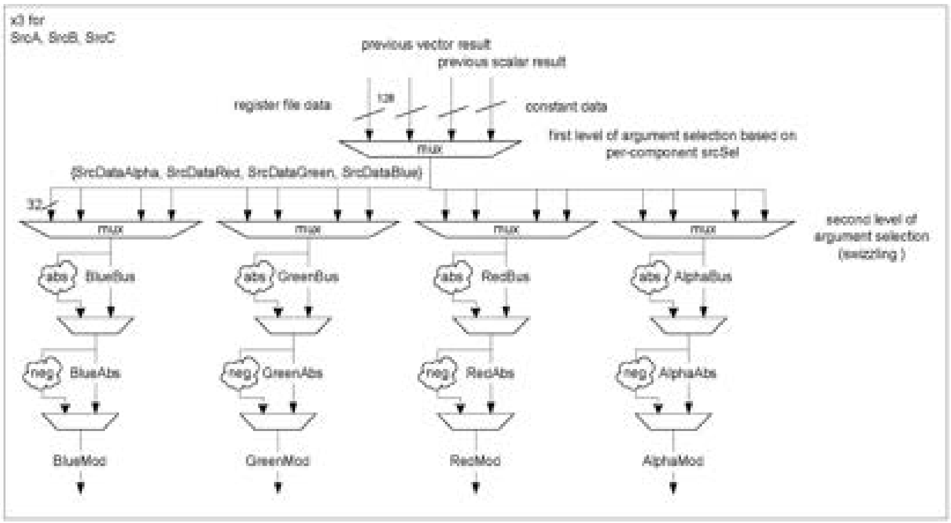


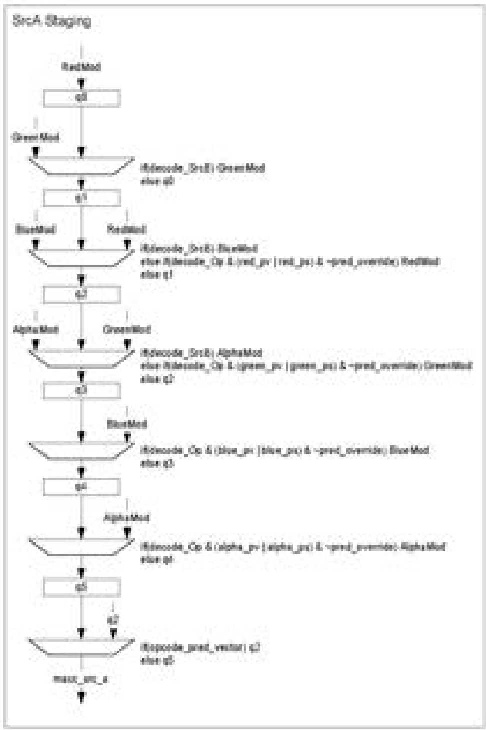
The control of all the multiplexers present at the input and output of the GPRs, input of the TAM and output of the parameter caches is done by the sequencer. In most cases this control represents the phase cycle relative to the ALU instruction start. In the above diagram, the GPR/MAC units of the same row are synchronous in phase. From one row to the other the execution sequence is phased out by one cycle.

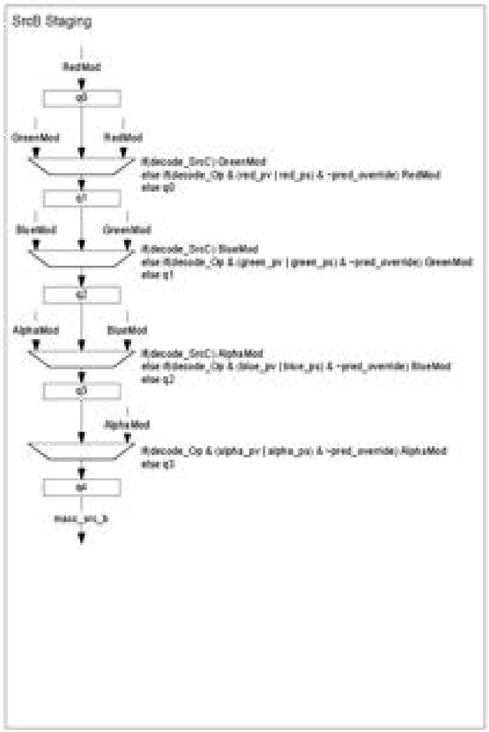
8.3.2 Argument Selection and Routing

The following diagram describes the routing and selection logic for the ALU instruction operands. Select fields or combination of select fields present in the ALU instruction word controls the first two levels of argument selection. The multiplexing control logic into the SRC (A, B or C) registers is a 4-state FSM that serializes the red, green, blue and alpha input arguments into the MAC unit. As mentioned before, the instruction ALU word specifies a pointer into the GPR register file for each of the sources. Also, the instruction specifies a set of select bits for the final MAC input operands A, B and C. The possible choices that Src A, B and C arguments can be selected from are: Register File Data, Constant Data, Previous Vector Result and Previous Scalar Result. The first level of muxing selects between Previous Vector Result, Register File Data and Constant Data, all 128-bit values. The second level of muxing implements the swizzling logic at 32-bit channel granularity.

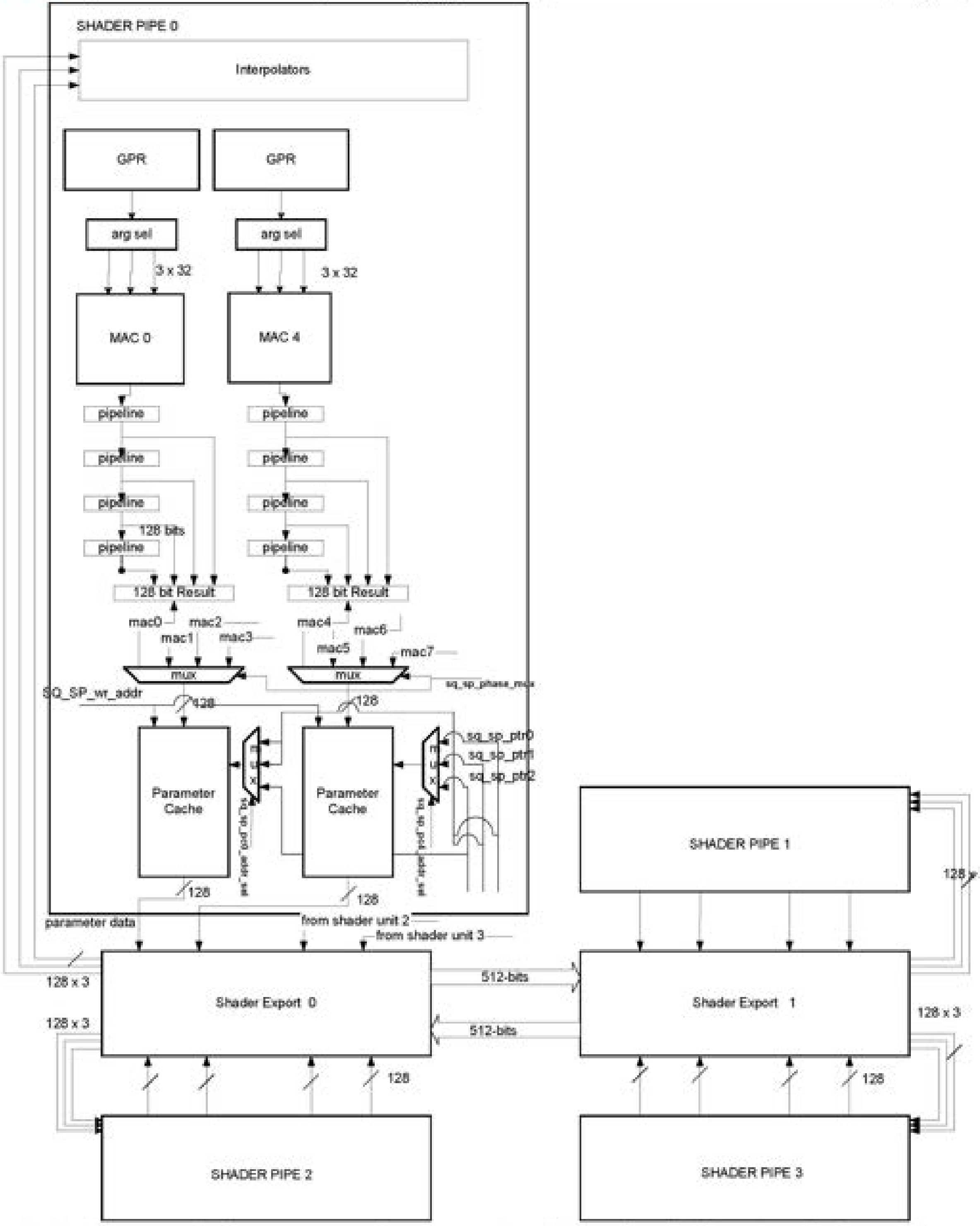
Argument Selection Logic




SrcA Staging
RedMod
q0
GreenMod
if(decode_SrcB) GreenMod
else q0
q1
BlueMod
RedMod
if(decode_SrcB) BlueMod
else if(decode_Op & (red_pv | red_ps) & ~pred_override) RedMod
else q1
q2
AlphaMod
GreenMod
if(decode_SrcB) AlphaMod
else if(decode_Op & (green_pv | green_ps) & ~pred_override) GreenMod
else q2
q3
BlueMod
if(decode_Op & (blue_pv | blue_ps) & ~pred_override) BlueMod
else q3
q4
AlphaMod
if(decode_Op & (alpha_pv | alpha_ps) & ~pred_override) AlphaMod
else q4
q5
q2
if(opcode_pred_vector) q2
else q5
macc_src_a


SrcB Staging
RedMod
q0
GreenMod
RedMod
if(decode_SrcC) GreenMod
else if(decode_Op & (red_pv | red_ps) & ~pred_override) RedMod
else q0
q1
BlueMod
GreenMod
if(decode_SrcC) BlueMod
else if(decode_Op & (green_pv | green_ps) & ~pred_override) GreenMod
else q1
q2
AlphaMod
BlueMod
if(decode_SrcC) AlphaMod
else if(decode_Op & (blue_pv | blue_ps) & ~pred_override) BlueMod
else q2
q3
AlphaMod
if(decode_Op & (alpha_pv | alpha_ps) & ~pred_override) AlphaMod
else q3
q4
macc_src_b

8.3.3 *Parameter Data Path*



The output of the vertex shader program, transformed parameter data is written into Parameter Cache memories. There is one parameter cache (128x128) for each vector unit, resulting in 4 parameter cache memories for one shader pipe. The write data path of each parameter cache is time-multiplexed between the 4 MAC units of a given shader vector unit. The 7-bit write address into parameter cache memory comes from Sequencer unit. This address is broadcasted to all the shader pipes. For more details on the parameter cache write path, please refer to Section 5.1.7 of this document.

The read address into parameter cache memories is a result of a muxing of three possible 7-bit address pointers broadcasted by the Sequencer to all shader pipes. These three pointers are part of "Parameter Cache Read Control Bus" described in Section 5.1.6 of this document.

There are 512-bit worth of data transferred from Shader Pipe to SX blocks for every read of the parameter cache.

Once read from the parameter caches, the parameter data is then routed by the SX units into the interpolation units at the top of the shader pipe.

8.4 Scalar Unit

A large portion of R400 Scalar Unit specification is based on the R300 Math Unit specifications as described in "R300 Vertex Assembler & Processor Architecture Specification" document Version 0.93. Section 3.3 of this document describes all the math operations and their respective opcodes supported by the Scalar Unit. There are four Scalar Units present in each shader pipe. The Scalar unit can perform one action each cycle with the result appearing after a latency of eight cycles. In this respect, the Scalar engine is fully pipelined.

Scalar Engine Functions and Precision

Function	Range	Precision[1]
x^n	$0.0 \le x \le 1.0$ $-128 \le n \le 128$	7 ??
$\frac{1}{x}$	$-\infty \le x \le \infty$	23
$\frac{1}{\sqrt{x}}$	$-\infty \le x \le \infty$	23
e^x	$-128.0 \le x \le 128.0$	16 ??
2^x	$0.0 \le x \le 1.0$	23
$\log_2(x)$	$1.0 \le x \le 2.0$	23

[1] Precision loses 1 bit each time range is doubled

The Scalar Engine also calculates $\frac{1}{x}$, $\frac{1}{\sqrt{x}}$, $\log_2(x)$, and 2^x using sixteen 32-entry lookup tables with four unsigned multipliers (2 ea. 16x16, and 2 ea. 12x12) and a 26-bit unsigned adder. These functions also use a 24-bit floating-point multiplier that applies the power term for the x^n and e^x functions (maybe we do not need this !!??).

8.4.1 *Scalar Engine Pipeline*

All the opcodes (except MULTIPLY) that do not use the power function pipeline are processed in the high precision pipeline. The mantissa for all functions is determined using the following function:

$$P = f(x_0) + A(x - x_0) + B(x - x_0)^2 + C(x - x_0)^3$$

The terms $f(x_0)$, A, B, and C were determined at 32 evenly spaced sample points over the range shown in Table 9-1 for each of the functions $\frac{1}{x}$, $\frac{1}{\sqrt{x}}$, $\log_2(x)$, and 2^x. These terms are stored in sixteen tables, four for each function, and addressed by bits 22 – 18 of the input value x (bits 23 - 19 for the $\frac{1}{\sqrt{x}}$ function). The $(x - x_0)$ terms are deltas which allow third-order interpolation between the sample points. The resolution of these lookup table values for each of the four functions is shown below. The largest resolution is assumed in the hardware. Lookup table values are left justified and smaller terms are packed with zeros. Deltas are also left justified, but smaller deltas have the truncated bits OR'ed with their LSB before and after being squared & cubed. The product terms are truncated to the indicated precision and then right justified before being added to the $f(x_0)$ *term* .

High Precision Function Coefficient Resolution

Function	$f(x_0)$	A	B	C
$\frac{1}{x}$	24	20	15	10
$\frac{1}{\sqrt{x}}$	24	20	16	12
$\log_2(x)$	25	20	15	9
2^x	25	20	14	7
Hardware	25	20	16	12

8.4.1.1.1.1.1 *High Precision Pipeline Exp₂ Preprocessing*

The two opcodes EXP_BASE2_DX and EXP_BASE2_FULL_DX require that the input value be preprocessed before it enters the main pipeline. The input exponent is first checked to determine whether the value can be represented within the IEEE single precision format (see Power Function Pipeline Log$_2$ to Real Conversion for details). If the log$_2$ exponent falls within the valid range, then the exp^2 preprocessor converts the input value to a S7.23 format value:

Input Mantissa (24 bits): 1mmmmmmmmmmmmmmmmmmmmmmm

Input Exponent (biased): eeeeeeee

Bias: 01111111

Subtract to get *Unbiased (signed) Exponent* : SUUUUUUUU

Absolute Value of *Unbiased Exponent* to get *shift value*: vvvvv

Shift Input Mantissa by *shift value* to get *adjusted input* (30 bits):

If *Unbiased Exponent* Positive, shift left: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

If *Unbiased Exponent* Negative, shift right: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Twos complement if input is negative to get S7.23 value: sbbbbbbbccccccccccccccccccccccc

8.4.1.1.1.1.2 *High Precision Pipeline Mantissa Calculation*

Calculation of the high precision mantissa is the same regardless of function type. Exceptions involve the selection of input values and are discussed in the process flow that follows. The multiplication of the first order term $(A * (x - x0))$ uses the power function pipeline multiplier. All other multiplications use dedicated multipliers.

<u>Get Starting value</u>:

EXP opcode:

Use preprocessed S7.23 value: 0sbbbbbbbccccccccccccccccccccccc

All other opcodes:

Use iAG_ME_IN_A, an IEEE floating point value: seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm

<u>Get Index</u>:

RECIP opcode:

Use LSB of exponent and 4 MSB's of input mantissa: ....iiiii...

All other opcodes:

Get lookup table index from 5 MSB's of input mantissa: ....iiiii..

<u>Get *delta*</u>:

RECIP & RECIPSQRT opcodes:

Use 19 LSB's of input mantissa for *delta*: ddddddddddddddddddd

All other opcodes:

Use 18 LSB's of input mantissa for *delta*: dddddddddddddddddd0

<u>Get *Slope1*</u>:

Term *A* from Lookup Table: AAAAAAAAAAAAAAAAAAAA

delta: ddddddddddddddddddd

Multiply to get *mult0*: qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq

RECIP, RECIPSQRT opcodes:

Use 20 MSB's of *mult0* for *Slope1*: qqqqqqqqqqqqqqqqqqqq0

All other opcodes:

Use 21 MSB's of *mult0* for *Slope1*: qqqqqqqqqqqqqqqqqqqqq

<u>Get *ddelta*</u>:

RECIPSQRT & RECIP opcodes:

OR together 4 LSB's of *delta*: 1

Concatenate bits 18 – 4 of *delta* to get *ddelta*: ddddddddddddddd1

RECIP, LOG opcodes:

OR together 5 LSB's of *delta*: 1

Concatenate bits 18 – 5 of *delta* to get *ddelta*: dddddddddddddd10

EXP opcode:

OR together 6 LSB's of *delta*: 1

Concatenate bits 18 – 6 of *delta* to get *ddelta*: ddddddddddddd100

<u>Get *delta_square*</u>:

Square *ddelta* to get *mult1*: rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr

RECIPSQRT opcode:

OR together bits 16 - 15 of *mult1*: 1

Concatenate bits 31 – 17 of *mult1* to get *delta_square*:	rrrrrrrrrrrrrrr1
RECIP & LOG opcodes:	
OR together bits 17 - 16 of *mult1*:	1
Concatenate bits 31 – 18 of *mult1* to get *delta_square*:	rrrrrrrrrrrrrr10
EXP opcode:	
OR together bits 18 - 17 of *mult1*:	1
Concatenate bits 31 – 19 of *mult1* to get *delta_square*:	rrrrrrrrrrrrr100

Get *Slope2*:

Term *B* from Lookup Table:	BBBBBBBBBBBBBBBB
delta_square:	rrrrrrrrrrrrrrrr
Multiply to get *mult2*:	ssssssssssssssssssssssssssssssss
RECIPSQRT opcode:	
Use 16 MSB's of *mult2* for *Slope2*:	ssssssssssssssss0
RECIP opcode:	
Use 15 MSB's of *mult2* for *Slope2*:	0sssssssssssssss0
EXP opcode:	
Use 14 MSB's of *mult2* for *Slope2*:	000ssssssssssssss
LOG opcode:	
Use 15 MSB's of *mult2* for *Slope2*:	00sssssssssssssss

Get *dddelta*:

RECIPSQRTopcode:	
OR together 8 LSB's of *delta*:	1
Concatenate bits 18 – 8 of *delta* to get *dddelta*:	ddddddddddd1
RECIP opcode:	
OR together 10 LSB's of *delta*:	1
Concatenate bits 18 – 10 of *delta* to get *dddelta*:	ddddddddd100
EXP opcode:	
OR together 13 LSB's of *delta*:	1
Concatenate bits 18 – 13 of *delta* to get *dddelta*:	dddddd100000
LOG opcode:	
OR together 11 LSB's of *delta*:	1
Concatenate bits 18 – 11 of *delta* to get *dddelta*:	dddddddd1000

Get *ddelta_square*:

RECIPSQRT opcode:	
OR together bits 20 - 19 of *mult1*:	1

Concatenate bits 31 – 21 of *mult1* to get *ddelta_square*: rrrrrrrrrrrl

RECIP opcode:

OR together bits 22 - 21 of *mult1*: 1

Concatenate bits 31 – 23 of *mult1* to get *ddelta_square*: rrrrrrrrr100

EXPopcode:

OR together bits 25 - 24 of *mult1*: 1

Concatenate bits 31 – 26 of *mult1* to get *ddelta_square*: rrrrrr100000

LOG opcode:

OR together bits 23 - 22 of *mult1*: 1

Concatenate bits 31 – 24 of *mult1* to get *ddelta_square*: rrrrrrrr1000

<u>Get *delta_cubed*</u>:

ddelta : dddddddddddd

ddelta_square : rrrrrrrrrrrr

Multiply to get *mult3*: tttttttttttttttttttttttt

RECIPSQRT opcode:

OR together bits 12 - 11 of *mult3*: 1

Concatenate bits 23 – 13 of *mult3* to get *delta_cubed*: ttttttttttt1

RECIP opcode:

OR together bits 14 - 13 of *mult3*: 1

Concatenate bits 23 – 15 of *mult3* to get *delta_cubed*: ttttttttt100

EXP opcode:

OR together bits 17 - 16 of *mult3*: 1

Concatenate bits 23 – 18 of *mult3* to get *delta_cubed*: tttttt100000

LOG opcode:

OR together bits 15 - 14 of *mult3*: 1

Concatenate bits 23 – 16 of *mult3* to get *delta_cubed*: tttttttt1000

<u>Get *slope3*</u>:

Term *C* from Lookup Table: cccccccccccc

delta_cubed: tttttttttttt

Multiply to get *mult4*: uuuuuuuuuuuuuuuuuuuuuuuu

RECIPSQRT opcode:

Use 12 MSB's of *mult4* for *Slope3*: uuuuuuuuuuuu0

RECIP opcode:

Use 10 MSB's of *mult4* for *Slope3*: 00uuuuuuuuuu0

EXP opcode:

Use 7 MSB's of *mult4* for *Slope3*:	000000uuuuuuu
LOG opcode:	
Use 9 MSB's of *mult4* for *Slope3*:	0000uuuuuuuuu

Term $f(x_0)$ from Lookup Table:	fffffffffffffffffffffffff
Slope1:	qqqqqqqqqqqqqqqqqq
Slope2:	ssssssssssssss
Slope3:	uuuuuuuuuu
Add to get *Full Mantissa*:	MMMMMMMMMMMMMMMMMMMMMMMMM
Right shift 2 bits to get *Final Mantissa*:	MMMMMMMMMMMMMMMMMMMMMMM

8.4.1.1.1.1.3 *High Precision Pipeline* Log_2 *Post Processing*

The two opcodes LOG_BASE2_DX and LOG_BASE2_FULL_DX post process the *full mantissa* to produce the final mantissa and exponent. This step is the same as the process performed in the power function pipeline except that the precision is higher.

Full Mantissa:	MMMMMMMMMMMMMMMMMMMMMMMMM
Unbiased Input Exponent :	SUUUUUUU

Combine *Unbiased Exponent* with *Final Mantissa*:	SUUUUUUUMMMMMMMMMMMMMMMMMMMMMMMMM
Sign multiply to get Log_2 *Mantissa*:	0ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
Normalize (using left *shift L*) to get *Norm Mantissa*:	NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN

Biased Exponent for *shift L* of 0:	10000110
Shift L:	LLL
Subtract to get Biased Log_2 *Exponent*:	EEEEEEEE

Norm Mantissa (right shift by 2, drop MSB):	NNNNNNNNNNNNNNNNNNNNNNN
Biased Log_2 *Exponent:*	EEEEEEEE
Unbiased Exponent Sign:	S
Combine to get final Log_2 Conversion Result:	SEEEEEEEENNNNNNNNNNNNNNNNNNNNNNN

8.4.1.1.1.1.4 *High Precision Pipeline Exponents*

Calculation of the exponents for the LOG_BASE2_DX, LOG_BASE2_FULL_DX, EXP_BASE2_DX, and EXP_BASE2_DX opcodes is covered in the corresponding pre and post processing sections. Exponents for the RECIP_DX, RECIP_FF, RECIP_SQRT_DX, and RECIP_SQRT_FF are calculated in the following way:

__Base Exponent__:

Input Exponent (biased):	0eeeeeeee

RECIPSQRT opcode:

Use Input Exponent (biased): 0bbbbbbbb

RECIP opcode:

Invert Input Exponent (biased): 1bbbbbbbb

Exponent Adjustment:

RECIPSQRT opcode:

Input is power of 4 (bits 23 – 0 are zero): 101111111

Input is not power of 4: 101111101

RECIP opcode:

Input is power of 2 (bits 22 – 0 are zero): 111111111

Input is not power of 2: 111111110

Final Exponent:

Base Exponent bbbbbbbbb

Exponent Adjustment: 1x11111xx

Add to get *Intermediate Exponent* : iiiiiiiiz

RECIPSQRT opcodes:

Use bits 8 – 1 of intermediate exponent: iiiiiiii

RECIP opcode:

Use bits 7 – 0 of intermediate exponent: iiiiiiiz

8.4.1.1.1.1.5 *High Precision Special Outputs*

The Math Engine computes or pipelines several values required by the DirectX 8.0 specification. These values include partial terms for the EXP and LOG opcodes, and the diffuse lighting attribute for the LIT opcode. The EXP opcode requires the input power to be split into integer and fractional parts, with the integer part then raised to a binary power. The integer and fractional parts are readily available in the form of the S7.23 fixed-point value found in the EXP_2 preprocessing step. The integer value is already in binary power form and can be muxed onto the oVE_WDATA output (with a zero mantissa). The fractional part of the power term needs to be normalized and converted to floating point format. Since that is a function that is performed in the LOG_2 post processing circuit, the lower 23 bits of the S7.23 value is left shifted 10 bits and muxed into that circuit. The LOG opcode requires that the input term be split into exponent and mantissa parts, with the exponent converted into a floating-point value. The mantissa is easily combined with an exponent of zero (or 0x7f biased) and muxed onto the oVE_WDATA output. Converson of the exponent requires an additional fixed-to-float circuit to generate the result since the LOG_2 post processing circuit is already used to convert the primary output for that opcode. The diffuse lighting attribute for the LIT opcode is simply pipelined from the iAG_ME_IN_B input and muxed onto the oVE_WDATA output.

8.4.1.1.1.1.6 *Determination of High Precision Coefficients*

The coefficients used to calculate the high precision mantissa were determined using third-order Lagrange polynomials. Since the mantissa is calculated separately from the exponent, the build range for each function was chosen so that the range of the function output values did not change dramatically (see below).

Build Ranges for Lagrangian Coeffiecients

Function	Build Range
$\frac{1}{x}$	x = 1.0 to 2.0

$\frac{1}{\sqrt{x}}$	x = 2.0 to 8.0
2^x	x = 1.0 to 2.0
$\log_2(x)$	x = 2.0 to 4.0

The build ranges were then divided into 32 equal segments. The coefficients for each segment were determined using the two end points of the segment, $f(x_0)$ and $f(x_1)$, as well as two points on the segment itself, $f(x_{01})$ and $f(x_{02})$, such that the distance along the x-axis between any two adjacent points, h, was 1/96 the total length of the build range (see figure below).

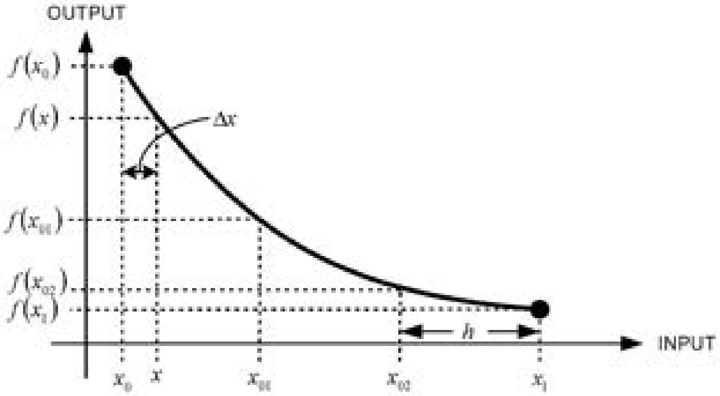


Figure 8-1: Build Range Segment

The third-order Lagrange polynomial used to approximate each segment of each function at any point x along the segment is as follows:

$$\hat{f}(x) = f(x_0)\left[\frac{(x-x_{01})(x-x_{02})(x-x_1)}{(x_0-x_{01})(x_0-x_{02})(x_0-x_1)}\right]+$$

$$f(x_{01})\left[\frac{(x-x_0)(x-x_{02})(x-x_1)}{(x_{01}-x_0)(x_{01}-x_{02})(x_{01}-x_1)}\right]+$$

$$f(x_{02})\left[\frac{(x-x_0)(x-x_{01})(x-x_1)}{(x_{02}-x_0)(x_{02}-x_{01})(x_{02}-x_1)}\right]+$$

$$f(x_1)\left[\frac{(x-x_0)(x-x_{01})(x-x_{02})}{(x_1-x_0)(x_1-x_{01})(x_1-x_{02})}\right]$$

Since $x_{01} = x_0 + h$, $x_{02} = x_0 + 2h$, and $x_1 = x_0 + 3h$, the polynomial can be rewritten in terms of x, x_0, h, $f(x_0)$, $f(x_1)$, $f(x_{01})$, and $f(x_{02})$:

$$\hat{f}(x) = -\frac{1}{6h^3}f(x_0)[(x-(x_0+h))(x-(x_0+2h))(x-(x_0+3h))]+$$

$$\frac{1}{2h^3}f(x_{01})[(x-(x_0))(x-(x_0+2h))(x-(x_0+3h))]+$$

$$-\frac{1}{2h^3}f(x_{02})[(x-(x_0))(x-(x_0+h))(x-(x_0+3h))]+$$

$$\frac{1}{6h^3}f(x_1)[(x-(x_0))(x-(x_0+h))(x-(x_0+2h))]$$

The polynomial is then factored for successive powers of the quantity $(x - x_0)$, i.e., 1, $(x - x_0)$, $(x - x_0)^2$ and $(x - x_0)^3$. Using the notation $\Delta x = x - x_0$, the polynomial can be reduced to the following form:

$$\hat{f}(x) = \sum_{i=0}^{3} a_i (\Delta x)^i$$

where each of the terms a_i is:

$$a_0 = f(x_0)$$

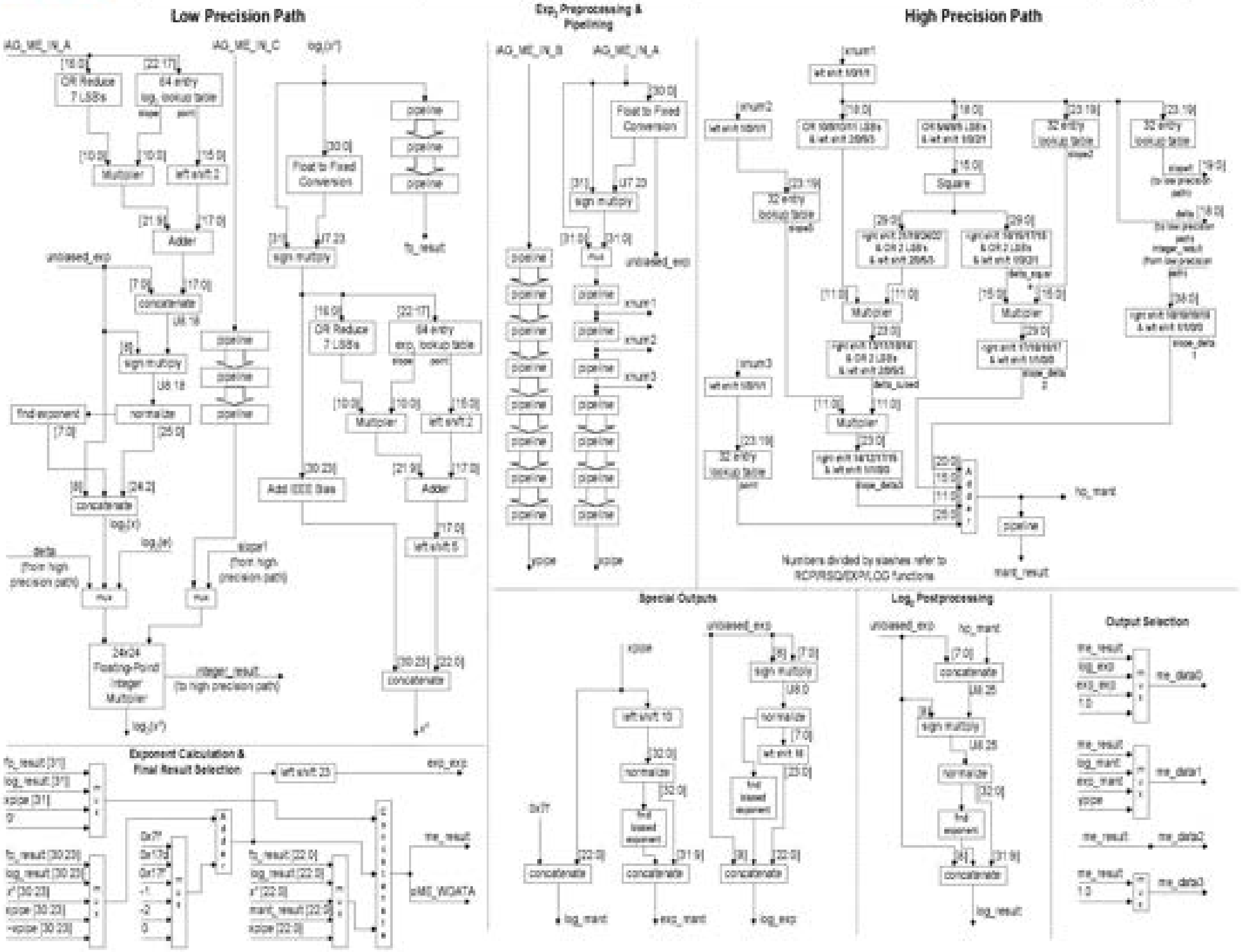
$$a_1 = \frac{1}{3h}\left[-5.5 f(x_0) + 9 f(x_{01}) - 4.5 f(x_{02}) + f(x_1)\right]$$

$$a_2 = \frac{1}{9h^2}\left[9 f(x_0) - 22.5 f(x_{01}) + 18 f(x_{02}) - 4.5 f(x_1)\right]$$

$$a_3 = \frac{1}{27h^3}\left[-4.5 f(x_0) + 13.5 f(x_{01}) - 13.5 f(x_{02}) + 4.5 f(x_1)\right]$$

Using the equations above, coefficients for each segment of each function were calculated and stored with the precision necessary to achieve 24-bit resolution.

The diagram below represents the Scalar Engine part of the pipeline that implements the LUT (Lookup Table) type scalar functions.



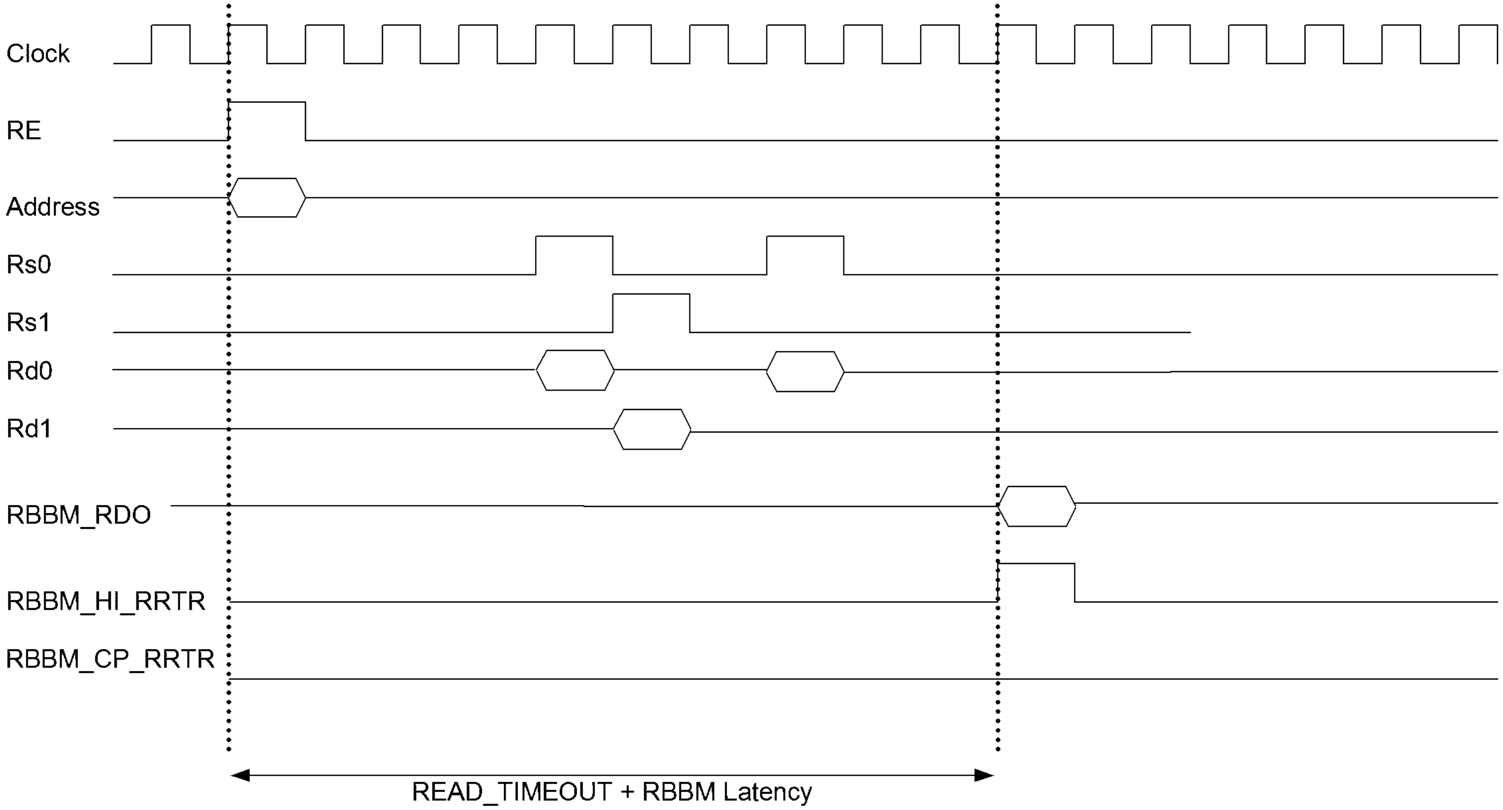
9. Open Issues

1. There's still to be decided the final location for difference engine used in calculating the delta between the vertex parameter values. The two possible place are:
   a. Next to the interpolators inside the parameter engines at the top of each shader pipe, but replicating this way the same logic across 4 shader pipelines.
   b. SX (Shader Exports). If this is the case, then Cylindrical Wrap logic should be propably be moved to the SX blocks.

2. The location of the real time parameter caches as well as the routing of this data by SX units into the parameter interpolators.
3. Data expansion logic in the path from the Fetch Engines into shader units/GPRs. Currently, the texture return data into GPRs is defined as a 512-bit bus for each shader pipe (Section 4.3.3).

Xenos RBBM Shared Read Transaction

Updated: 11/19/2003
John A. Carey



v. 1.80 Tom Frisinger/Jocelyn Houle				R400 Texture Fetch Instruction
Name	**Dword**	**Bits**	**Size**	**Description**
OPCODE	0	4:0	5	Texture fetch instruction opcodes. 1=FetchTextureMap 2=Reserved for Fetch3DNoiseMap (hardware assumes 4Kx4Kx4K noise texture) 3=Reserved for FetchShadowMap 4=Reserved for FetchMultiSample (has slightly different texture constant layout) 16=GetBorderColorFraction(retrieves border color contribution, X=border color fraction) 17=GetCompTexLOD (retrieves LOD computed in texture pipe, X=computed LOD for all pixels of a quad (garbage in a vertex shader)) 18=GetGradients (retrieves slopes of SRC XY relative to screen horizontal and vertical, X=dx/dh, Y=dx/dv, Z=dy/dh, W=dy/dv) 19=GetWeights (retrieves weights used in a bilinear fetch, X=horizontal lerp factor, Y=vertical lerp factor) 24=SetTexLOD (sets LOD register, X=LOD to use for the current pixel) 25=SetGradientsH (sets the horizontal gradients, X=dx/dh, Y=dy/dh, Z=dz/dh) 26=SetGradientsV (sets the vertical gradients, X=dx/dv, Y=dy/dv, Z=dz/dv) 27=Reserved for SetFilter4Weights (loads filter table w/ weights used for arbitrary filters)
SRC_GPR	0	10:5	6	Specifies the source GPR for the coordinates.
SRC_GPR_AM	0	11	1	Specifies the SRC_GPR addressing mode. 0=Logical register addressing 1=Current Loop Index relative register addressing
DST_GPR	0	17:12	6	Specifies the destination GPR for the coordinates.
DST_GPR_AM	0	18	1	Specifies the DST_GPR addressing mode. 0=Logical register addressing 1=Current Loop Index relative register addressing
FETCH_VALID_ONLY	0	19	1	Specifies if the sequencer can specify an optimized fetch or not. 0=No (must fetch all the data, all pixels are going to be valid) 1=Yes (can optimize performance, some pixels can be invalid)
CONST_INDEX	0	24:20	5	Selects which 192 bit constant holds the texture base address and other information.
TX_COORD_DENORM	0	25	1	Specifies if the texture coordinates sent are already normalized or not (i.e. pre-multiplied by texture dimension). Unnormalized mode is only valid when CLAMP mode is clamp. If it is not clamp, the coordinate will be treated as normalized. 0=Normalized, range [0..1] 1=Unnormalized, range [0..dimension]
SRC_SEL_X	0	27:26	2	Selects between WZYX of SRC_GPR which component goes for X in the texture fetch. 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W
SRC_SEL_Y	0	29:28	2	Selects between WZYX of SRC_GPR which component goes for Y in the texture fetch. 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W
SRC_SEL_Z	0	31:30	2	Selects between WZYX of SRC_GPR which component goes for Z in the texture fetch. 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W
DST_SEL_X	1	2:0	3	Select what component of SRC to write to the X component of DST_GPR. This sel happens in addition to sel in constant. 0=SRC_X 1=SRC_Y 2=SRC_Z 3=SRC_W 4=0.0f 5=1.0f ... 7=Mask (keep value in DST_GPR)
DST_SEL_Y	1	5:3	3	Select what component of SRC to write to the Y component of DST_GPR. This sel happens in addition to sel in constant. 0=SRC_X 1=SRC_Y 2=SRC_Z 3=SRC_W 4=0.0f 5=1.0f ... 7=Mask (keep value in DST_GPR)
DST_SEL_Z	1	8:6	3	Select what component of SRC to write to the Z component of DST_GPR. This sel happens in addition to sel in constant. 0=SRC_X 1=SRC_Y 2=SRC_Z 3=SRC_W 4=0.0f 5=1.0f ... 7=Mask (keep value in DST_GPR)
DST_SEL_W	1	11:9	3	Select what component of SRC to write to the W component of DST_GPR. This sel happens in addition to sel in constant. 0=SRC_X 1=SRC_Y 2=SRC_Z 3=SRC_W 4=0.0f 5=1.0f ... 7=Mask (keep value in DST_GPR)
MAG_FILTER	1	13:12	2	Filter used when texture is magnified 0=Point 1=Linear 2=Arbitrary (not supported on R400)

v. 1.80 Tom Frisinger/Jocelyn Houle				R400 Texture Fetch Instruction
Name	**Dword**	**Bits**	**Size**	**Description**
MIN_FILTER	1	15:14	2	Filter used when texture is minified 0=Point 1=Linear 2=Reserved for Arbitrary (not supported on R400) 3=Off (use value specified in constant)
MIP_FILTER	1	17:16	2	Filter used between mipmap levels 0=Point 1=Linear 2=BaseMap (None, disregard mipmapping, always use base map, ignores min_mip_level) 3=Off (use value specified in constant)
ANISO_FILTER	1	20:18	3	Anisotropy Enable 0=Disabled 1=Enabled, Max 1 to 1 2=Enabled, Max 2 to 1 3=Enabled, Max 4 to 1 4=Enabled, Max 8 to 1 5=Enabled, Max 16 to 1 … 7=Off (use value specified in constant)
ARBITRARY_FILTER	1	23:21	3	Specifies type of arbitrary filter to use (not supported on R400) 0=Filter2x4_Sym 1=Filter2x4_Asym 2=Filter4x2_Sym 3=Filter4x2_Asym 4=Filter4x4_Sym 5=Filter4x4_Asym … 7=Off (use value specified in constant)
VOL_MAG_FILTER	1	25:24	2	Filter used between layers of a volume when doing magnification. 0=Point 1=Linear ... 3=Off (use value specified in constant)
VOL_MIN_FILTER	1	27:26	2	Filter used between layers of a volume when doing minification. 0=Point 1=Linear ... 3=Off (use value specified in constant)
USE_COMP_LOD	1	28	1	Specifies if the LOD computed by the texture pipe should be used in LOD computation. 0=No 1=Yes (common case)
USE_REG_LOD	1	30:29	2	Specifies if the LOD register should be added to LOD computation (use SetTexLOD to set LOD register). 0=No (common case) 1=Yes (per pixel, full speed) 2=Reserved 3=Reserved
PRED_SELECT	1	31	1	Indicates to the sequencer if the instruction is predicated or not. 0=Not predicated 1=Predicated (see PRED_CONDITION)
USE_REG_GRADIENTS	2	0	1	Specifies if the gradients stored previously using SetGradients{H|V} must be used or not. Affects the computed LOD value, as well as the line of anisotropy. 0=No (common case) 1=Yes
SAMPLE_LOCATION	2	1	1	Specifies sampling position for gradient/LOD correction. 0=Centroid of fragment 1=Center of fragment
LOD_BIAS	2	8:2	7	LOD bias to apply after computing the mipmap and min/mag determination, but before clamping. Signed fixed point w/ 4 bits of fraction (3.4), and ranges from [–4..4). This value is always summed with all other LOD's.
UNUSED	2	15:9	7	Unused bits. 0=MUST_BE_ZERO
OFFSET_X	2	20:16	5	Value added to X component of texel address right before sampling (texel space). Signed fixed point w/ 1 bit of fraction (4.1), and ranges from [–8..8).
OFFSET_Y	2	25:21	5	Value added to Y component of texel address right before sampling (texel space). Signed fixed point w/ 1 bit of fraction (4.1), and ranges from [–8..8).
OFFSET_Z	2	30:26	5	Value added to Z component of texel address right before sampling (texel space). Controls sample selection for FetchMultiSample (Point Filter). Signed fixed point w/ 1 bit of fraction (4.1), and ranges from [–8..8).
PRED_CONDITION	2	31	1	Indicates to the sequencer what value to use for the predication.
Total	**3**	**96**	**96**	

Notes:

1.) The mipmap LOD is computed using 4 components per direction:

LOD = ucl*LODcomp + url*LODreg + LODinstr + LODconst
LODcomp=LOD computed in the texture pipe
LODreg=LOD specified in advance using the SetTexLOD
LODinst=LOD bias specified in the instruction
LODconst=LOD bias specified in the constants
ucl=USE_COMP_LOD
url=USE_REG_LOD

v. 1.80 Tom Frisinger/Jocelyn Houle				R400 Texture Fetch Constant Fields	FetchMultiSample
Name	**Dword**	**Bits**	**Size**	**Description**	**Name**
TYPE	0	1:0	2	Constant type/state. Used by PM4 parser/capture utilities. HW must ignore. 0=Invalid Constant 1=Invalid Vertex Constant (N/A) 2=Valid Texture Constant 3=Valid Vertex Constant (N/A)	TYPE
FORMAT_COMP_X	0	3:2	2	Format of source component (X). See Numbers page for details. 0=Unsigned 1=Signed 2=Unsigned Biased 3=Unsigned Gamma'd (NUM_FORMAT_ALL should be RF)	FORMAT_COMP_X
FORMAT_COMP_Y	0	5:4	2	Format of source component (Y). See Numbers page for details. 0=Unsigned 1=Signed 2=Unsigned Biased 3=Unsigned Gamma'd (NUM_FORMAT_ALL should be RF)	FORMAT_COMP_Y
FORMAT_COMP_Z	0	7:6	2	Format of source component (Z). See Numbers page for details. 0=Unsigned 1=Signed 2=Unsigned Biased 3=Unsigned Gamma'd (NUM_FORMAT_ALL should be RF)	FORMAT_COMP_Z
FORMAT_COMP_W	0	9:8	2	Format of source component (W). See Numbers page for details. 0=Unsigned 1=Signed 2=Unsigned Biased 3=Unsigned Gamma'd (NUM_FORMAT_ALL should be RF)	FORMAT_COMP_W
CLAMP_X	0	12:10	3	Clamp mode for first texture coordinate (X) 0=Wrap/Repeat 1=Mirror 2=Clamp to last texel, [0..1] normalized, [0..dimension] unnormalized 3=MirrorOnce to last texel, [-1..1] 4=Clamp half way to border color, [0..1] normalized, [0..dimension] unnormalized 5=MirrorOnce half way to border color, [-1..1] 6=Clamp to border color, [0..1] normalized, [0..dimension] unnormalized 7=MirrorOnce to border color, [-1..1]	CLAMP_X
CLAMP_Y	0	15:13	3	Clamp mode for first texture coordinate (Y) 0=Wrap/Repeat 1=Mirror 2=Clamp to last texel, [0..1] normalized, [0..dimension] unnormalized 3=MirrorOnce to last texel, [-1..1] 4=Clamp half way to border color, [0..1] normalized, [0..dimension] unnormalized 5=MirrorOnce half way to border color, [-1..1] 6=Clamp to border color, [0..1] normalized, [0..dimension] unnormalized 7=MirrorOnce to border color, [-1..1]	CLAMP_Y
CLAMP_Z	0	18:16	3	Clamp mode for first texture coordinate (Z) 0=Wrap/Repeat 1=Mirror 2=Clamp to last texel, [0..1] normalized, [0..dimension] unnormalized 3=MirrorOnce to last texel, [-1..1] 4=Clamp half way to border color, [0..1] normalized, [0..dimension] unnormalized 5=MirrorOnce half way to border color, [-1..1] 6=Clamp to border color, [0..1] normalized, [0..dimension] unnormalized 7=MirrorOnce to border color, [-1..1]	MSAA_FILTER_FUNCTION 0=Point Filter 1=Box Filter (normal)
SIGNED_RF_MODE_ALL	0	19	1	Mapping to use when converting source (WZYX) signed RF (and unsigned RF biased) to float. See Numbers page for details. 0=ZERO_CLAMP_MINUS_ONE 1=NO_ZERO (see note below about filtering this format) **UNSUPPORTED UNDER XENOS	SIGNED_RF_MODE_ALL
DIM	0	21:20	2	Texture dimensionality (transitionnal, see DWORD 5) 0=1D texture 1=2D texture 2=3D texture 3=CubeMap texture (assumes 2D textures for size determination)	DIM (MSAA_ARRAY) 1=MUST_BE_ONE
PITCH	0	30:22	9	Pitch in units of 32 texels.	PITCH (MSAA)
TILED	0	31	1	Tiling Enable 0=Disabled 1=Enabled	TILED (MSAA_TILING) 1=MUST_BE_ONE
DATA_FORMAT	1	5:0	6	Texture format. See DATA_FORMAT table. All formats permitted.	RESULT_FORMAT (RB -> TC) -See RB
ENDIAN_SWAP	1	7:6	2	Endian control 0=ENDIAN_NONE No endian swapping 1=ENDIAN_8IN16 8 bit swap within 16 bit word - 0xAABBCCDD -> 0xBBAADDCC 2=ENDIAN_8IN32 8 bit swap within 32 bit word - 0xAABBCCDD -> 0xDDCCBBAA 3=ENDIAN_16IN32 16 bit swap within 32 bit word - 0xAABBCCDD -> 0XCCDDAABB	ENDIAN_SWAP (MSAA_ENDIAN)
REQUEST_SIZE	1	9:8	2	Optimizes read request size. 0=256 bit requests (data in FB typically) 1=512 bit requests (data in AGP/3GIO typically) 2=1024 bit requests (TBD) 3=Reserved	RESULT_NUMBER (RB -> TC) -See RB
DIM_HI	1	10	1	Bit indicating if 2D maps are stack maps or not. 0=Normal 1=Stack map (DIM must be set to 1=2D)	
NEAREST_CLAMP_POLICY	1	11	1	Controls when nearest filtering will sample border color/texels. 0=D3D mode - Nearest filtering samples border color/texels depending on clamping mode 1=OGL mode - Nearest filtering will never sample border color/texels when using 'Clamp half way to border color' or 'MirrorOnce half way to border color' clamping modes.	NEAREST_CLAMP_POLICY
BASE_ADDRESS	1	31:12	20	Starting address for the base map, (4KB aligned, minimum). See Features page for alignment restriction details.	MSAA_BASE -See RB
SIZE	2	31:0	32	Texture size containing all the dimensions. (if OPCODE is Fetch3DNoiseMap, HW assumes 4Kx4Kx4K) All sizes are expressed as size-1 (0 cannot be expressed). See Features page for size restriction details. 1D: 24 bits for width [23:0] 2D: 13 bits for width [12:0], 13 bits for height [25:13] 3D: 11 bits for width [10:0], 11 bits for height [21:11], 10 bits for depth [31:22] Cube/Stack: 13 bits for width [12:0], 13 bits for height [25:13], 6 bits for depth [31:26] (Cube needs to set depth of 6)	SIZE (MSAA)
NUM_FORMAT_ALL	3	0	1	Indicates if source (WZYX) data is repeating fraction (RF) or integer (INT). See Numbers page for details. (N is the size of the component in bits derived from DATA_FORMAT and gamma) 0=RF: a repeating fraction number (0.N) that represents a closed interval, []. 1=INT: an integer number (N.0) that represents a half-open interval, [].	NUM_FORMAT_ALL
DST_SEL_X	3	3:1	3	Select what component of SRC to write to the X component of the returning data. 0=SRC_X 1=SRC_Y 2=SRC_Z 3=SRC_W 4=0.0f 5=1.0f	DST_SEL_X

v. 1.80 Tom Frisinger/Jocelyn Houle				R400 Texture Fetch Constant Fields	FetchMultiSample
DST_SEL_Y	3	6:4	3	Select what component of SRC to write to the Y component of the returning data. 0=SRC_X 1=SRC_Y 2=SRC_Z 3=SRC_W 4=0.0f 5=1.0f	DST_SEL_Y
DST_SEL_Z	3	9:7	3	Select what component of SRC to write to the Z component of the returning data. 0=SRC_X 1=SRC_Y 2=SRC_Z 3=SRC_W 4=0.0f 5=1.0f	DST_SEL_Z
DST_SEL_W	3	12:10	3	Select what component of SRC to write to the W component of the returning data. 0=SRC_X 1=SRC_Y 2=SRC_Z 3=SRC_W 4=0.0f 5=1.0f	DST_SEL_W
EXP_ADJUST_ALL	3	18:13	6	Final bias for exponent applied at end of conversion to 32-bit floating point. Signed integer with range [-32..32) This is always applied to all channels (WZYX). Set to 0 normally.	EXP_ADJUST_ALL
MAG_FILTER	3	20:19	2	Filter used when texture is magnified 0=Point 1=Linear 2=Arbitrary (not supported on R400)	MAG_FILTER
MIN_FILTER	3	22:21	2	Filter used when texture is minified 0=Point 1=Linear 2=Arbitrary (not supported on R400)	MIN_FILTER
MIP_FILTER	3	24:23	2	Filter used between mipmap levels 0=Point 1=Linear 2=BaseMap (None, disregard mipmapping, always use base map, ignores MIN_MIP_LEVEL)	MIP_FILTER 2=MUST_BE_TWO
ANISO_FILTER	3	27:25	3	Anisotropy Enable 0=Disabled 1=Enabled, Max 1 to 1 2=Enabled, Max 2 to 1 3=Enabled, Max 4 to 1 4=Enabled, Max 8 to 1 5=Enabled, Max 16 to 1	ANISO_FILTER
ARBITRARY_FILTER	3	30:28	3	Specifies type of arbitrary filter to use (not supported on R400) 0=Filter2x4_Sym 1=Filter2x4_Asym 2=Filter4x2_Sym 3=Filter4x2_Asym 4=Filter4x4_Sym 5=Filter4x4_Asym	MSAA_NUM_SAMPLES -See PA
BORDER_SIZE	3	31	1	Give the border size (0 or 1). This is used for border texels.	BORDER_SIZE 0=MUST_BE_ZERO
VOL_MAG_FILTER	4	0	1	Filter used between layers of a volume when doing magnification. 0=Point 1=Linear	MSAA_CLEAR_LO -See RB
VOL_MIN_FILTER	4	1	1	Filter used between layers of a volume when doing minification. 0=Point 1=Linear	
MIN_MIP_LEVEL	4	5:2	4	LOD index of the largest mipmap we can use (0 is largest).	
MAX_MIP_LEVEL	4	9:6	4	LOD index of the smallest mipmap we can use (greater than MIN_MIP_LEVEL). HW will use minimum of this value and total number of mipmap levels in mipmap chain.	
MAG_ANISO_WALK	4	10	1	Do anisotropy walk when anisotropy filter is enabled and magnifying. 0=Disabled (1 sample) 1=Enabled	
MIN_ANISO_WALK	4	11	1	Do anisotropy walk when anisotropy filter is enabled and minifying. 0=Disabled (1 sample) 1=Enabled	
LOD_BIAS	4	21:12	10	LOD bias to apply after computing the mipmap and determining min/mag, but before clamping. Signed fixed point w/ 5 bits of fraction (5.5), and ranges from [-16..16).	
GRAD_EXP_ADJUST_H	4	26:22	5	Exponent adjust applied to the horizontal gradients' exponent. Signed integer, ranges from [-16..16).	
GRAD_EXP_ADJUST_V	4	31:27	5	Exponent adjust applied to the vertical gradients' exponent. Signed integer, ranges from [-16..16).	
BORDER_COLOR	5	1:0	2	Border Color. See Border Color page for details. 0=ABGR Black 1=ABGR White 2=ACbYCr Black 3=ACbCrY Black	BORDER_COLOR
FORCE_BC_W_TO_MAX	5	2	1	Force W component of border color value to maximum value. This overrides the W value in BORDER_COLOR. See Border Color page for details. 0=Disabled 1=Enabled	FORCE_BC_W_TO_MAX
TRI_JUICE	5	4:3	2	Tri-linear performance tweaker. Corresponds to the point below which mip fraction should be clamped to 0, therefore sampling a single map. Tri-linear filtering remains piecewise linear and continuous. 0=0 (normal) 1=1/6 2=1/4 3=3/8	MSAA_FORMAT -See RB
ANISO_BIAS	5	8:5	4	Anisotropic filtering performance tweaker. The bias is a u1.3 number, range [0, 2). It is subtracted from the computed aniso ratio, yielding a lower number of samples taken. Improves performance while keeping filtering quality better than plain trilinear filtering.	
DIM	5	10:9	2	Texture dimensionality. 0=1D texture 1=2D texture 2=3D texture 3=CubeMap texture (assumes 2D textures for size determination, special gradient path for proper LOD determination)	MSAA_NUMBER -See RB
MIP_PACKING	5	11	1	Indicates whether mipmap tail packing is used or not. 0=Disabled (every mip level starts at a 4KB boundary) 1=Enabled (as soon as either width or height is 16 or less, mip levels are packed together)	
MIP_ADDRESS	5	31:12	20	Address of start of mipmap chain, (4KB aligned, minimum). Must be valid value when mipmapping is enabled. See Features page for alignment restriction details.	MSAA_TILE_BASE -See RB
Total	**6**	**192**	**192**		

Notes:
1.) Filtering of signed repeating fractions with SIGNED_RF_MODE_ALL == NO_ZERO may not produce the correct results because of the implicit bias in this format. This format is really only meant to be used with filtering off. Programmers should only enable filtering if they understand the mathematical implications of doing so.
2.) ARBITRARY_FILTER is undefined in the constant for FetchMultiSample and should be set via instruction.
3.) MAG_ANISO_WALK and MIN_ANISO_WALK are forced to 1 (enabled) when anisotropy is enabled for FetchMultiSample.

v. 1.80 Tom Frisinger/Jocelyn Houle				**R400 Vertex Fetch Instruction**
Name	**Dword**	**Bits**	**Size**	**Description**
OPCODE	0	4:0	5	Vertex fetch instruction opcodes. 0=FetchVertex
SRC_GPR	0	10:5	6	Specifies the source GPR for index
SRC_GPR_AM	0	11	1	Specifies the SRC_GPR addressing mode. 0=Logical register addressing 1=Current Loop Index relative register addressing
DST_GPR	0	17:12	6	Specifies the destination GPR for the result
DST_GPR_AM	0	18	1	Specifies the DST_GPR addressing mode. 0=Logical register addressing 1=Current Loop Index relative register addressing
FETCH_VALID_ONLY	0	19	1	Specifies if the sequencer can specify an optimized fetch or not. 1=MUST_BE_ONE
CONST_INDEX	0	24:20	5	Selects which 192 bit constant holds the vertex base/limit pair.
CONST_INDEX_SEL	0	26:25	2	Select which part of 192 bit constant holds base/limit pair. 0=Low (bits 0-63) 1=Middle (bits 64-127) 2=High (bits 128-191)
COUNT_LO	0	29:27	3	Number of DWORDs to "prefetch" for Mega/Mama fetch. [COUNT_HI][COUNT_LO] encode the total count (minus 1). Currently, this value cannot be greater than 0x7 on R500. Ignored for vfetches through TC. HW will clamp minimum count to size implied in DATA_FORMAT. 0=Prefetch 1 DWORD 1=Prefetch 2 DWORD ...
SRC_SEL	0	31:30	2	Select which 32bit component in SRC_GPR has index. 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W
DST_SEL_X	1	2:0	3	Select what component of SRC to write to the X component of DST_GPR. 0=SRC_X 1=SRC_Y 2=SRC_Z 3=SRC_W 4=0.0f 5=1.0f ... 7=Mask (keep value in DST_GPR)
DST_SEL_Y	1	5:3	3	Select what component of SRC to write to the Y component of DST_GPR. 0=SRC_X 1=SRC_Y 2=SRC_Z 3=SRC_W 4=0.0f 5=1.0f ... 7=Mask (keep value in DST_GPR)
DST_SEL_Z	1	8:6	3	Select what component of SRC to write to the Z component of DST_GPR. 0=SRC_X 1=SRC_Y 2=SRC_Z 3=SRC_W 4=0.0f 5=1.0f ... 7=Mask (keep value in DST_GPR)
DST_SEL_W	1	11:9	3	Select what component of SRC to write to the W component of DST_GPR. 0=SRC_X 1=SRC_Y 2=SRC_Z 3=SRC_W 4=0.0f 5=1.0f ... 7=Mask (keep value in DST_GPR)
FORMAT_COMP_ALL	1	12	1	Sign of source components (WZYX). See Numbers page for details. 0=Unsigned 1=Signed
NUM_FORMAT_ALL	1	13	1	Indicates if source (WZYX) data is repeating fraction (RF) or integer (INT). See Numbers page for details. (N is the size of the component in bits derived from DATA_FORMAT) 0=RF: a repeating fraction number (0.N) that represents a closed interval, []. 1=INT: an integer number (N.0) that represents a half-open interval, [).
SIGNED_RF_MODE_ALL	1	14	1	Mapping to use when converting source (WZYX) signed repeating fractions to float. See Numbers page for details. 0=ZERO_CLAMP_MINUS_ONE (D3D) 1=NO_ZERO (OGL) **UNSUPPORTED UNDER XENOS
INDEX_ROUND	1	15	1	Control for converting index from float to integer. 0=Truncate to negative infinity 1=Round
DATA_FORMAT	1	21:16	6	Vertex format. See DATA_FORMAT table. Only vertex formats permitted.
COUNT_HI	1	23:22	2	Reserved for expansion. See COUNT_LO 0=MUST_BE_ZERO
EXP_ADJUST_ALL	1	29:24	6	Final bias for exponent applied at end of conversion to 32-bit floating point. Signed integer with range [-32..32) This is alway applied to all channels (WZYX). Set to 0 normally.
FETCH_TYPE	1	30	1	Type of Fetch. Ignored for vfetches through TC. 0=Mega/Prefetch/Mama fetch (COUNT_* used) 1=Mini/Non-Prefetch/Baby fetch (COUNT_* ignored)
PRED_SELECT	1	31	1	Indicates to the sequencer if the instruction is predicated or not. 0=Not predicated

v. 1.80 Tom Frisinger/Jocelyn Houle				R400 Vertex Fetch Instruction
OFFSET_X	2	30:8	23	Which DWORD we want to fetch. Signed integer.
PRED_CONDITION	2	31	1	Indicates to the sequencer what value to use for the predication.
Total	**3**	**96**	**96**	

v. 1.80 Tom Frisinger/Jocelyn Houle				R400 Vertex Fetch Constant Fields
Name	**Dword**	**Bits**	**Size**	**Description**
TYPE	0	1:0	2	Constant type/state. Used by PM4 parser/capture utilities. HW must ignore. 0=Invalid Constant 1=Invalid Vertex Constant 2=Valid Texture Constant (N/A) 3=Valid Vertex Constant
BASE_ADDRESS	0	31:2	30	Base address (must be DWORD aligned)
ENDIAN_SWAP	1	1:0	2	Endian control 0=ENDIAN_NONE No endian swapping 1=ENDIAN_8IN16 8 bit swap within 16 bit word - 0xAABBCCDD -> 0xBBAADDCC 2=ENDIAN_8IN32 8 bit swap within 32 bit word - 0xAABBCCDD -> 0xDDCCBBAA 3=ENDIAN_16IN32 16 bit swap within 32 bit word - 0xAABBCCDD -> 0XCCDDAABB
SIZE	1	25:2	24	Size of array in DWORDs.
CLAMP_X	1	26	1	Clamp mode 0=Clamp to last 1=Clamp to border color
BORDER_COLOR	1	27	1	Border Color. See Border Color page for details. 0=ABGR Black 1=Reserved
REQUEST_SIZE	1	29:28	2	Optimizes read request size. 0=256 bit requests (data in FB typically) 1=512 bit requests (data in AGP/3GIO typically) 2=1024 bit requests (TBD) 3=Reserved
CLAMP_DISABLE	1	30	1	Disable address clamping. 0=Clamping enabled 1=Clamping disabled
UNUSED	1	31	1	Unused bits. 0=MUST_BE_ZERO
Total	**2**	**64**	**64**	

Notes:
1.) DWORD fetch address = BASE_ADDRESS + SRC_GPR * STRIDE + OFFSET_X.

R400 DATA_FORMAT

Value	DATA_FORMAT	Vertex Format?	Texture Format?	Degammable?	Filterable? (15)	Cycles Multiplier
0	FMT_1_REVERSE (X)*	No	Yes	No	Yes	x1
1	FMT_1 (X)*	No	Yes	No	Yes	x1
2	FMT_8 (X)*	No	Yes	Yes	Yes	x1
3	FMT_1_5_5_5 (WZYX)*	No	Yes	Yes	Yes	x1
4	FMT_5_6_5 (ZYX)*	No	Yes	Yes	Yes	x1
5	FMT_6_5_5 (ZYX)	No	Yes	Yes	Yes	x1
6	FMT_8_8_8_8 (WZYX)*	Yes	Yes	Yes	Yes	x1
7	FMT_2_10_10_10 (WZYX)	Yes	Yes	No	No	x1
8	FMT_8_A (X) (duplicate format, see 2)*					
9	FMT_8_B (X) (duplicate format, see 2)*					
10	FMT_8_8 (YX)	No	Yes	Yes	Yes	x1
11	FMT_Cr_Y1_Cb_Y0_REP (Z=Cb, Y=Y, X=Cr) (Packed 422)*	No	Yes	No	Yes	x1
12	FMT_Y1_Cr_Y0_Cb_REP (Z=Cb, Y=Y, X=Cr) (Packed 422)*	No	Yes	No	Yes	x1
13	RESERVED					
14	FMT_8_8_8_8_A (WZYX) (duplicate format, see 6)*					
15	FMT_4_4_4_4 (WZYX)*	No	Yes	Yes	Yes	x1
16	FMT_10_11_11 (ZYX)	Yes	Yes	No	No	x1
17	FMT_11_11_10 (ZYX)	Yes	Yes	No	No	x1
18	FMT_DXT1 (WZYX)	No	Yes	Yes	Yes	x1
19	FMT_DXT2_3 (WZYX)	No	Yes	Yes	Yes	x1
20	FMT_DXT4_5 (WZYX)	No	Yes	Yes	Yes	x1
21	RESERVED					
22	FMT_24_8 (X) (depth/stencil format)	No	Yes	No	Yes (14)	x1
23	FMT_24_8_FLOAT (X) (depth/stencil format)	No	Yes	No	No (14)	x1
24	FMT_16 (X)	No	Yes	No	Yes	x1
25	FMT_16_16 (YX)	Yes	Yes	No	Yes	x1
26	FMT_16_16_16_16 (WZYX)	Yes	Yes	No	Yes	x2
27	FMT_16_EXPAND (X)	No	Yes	No	Yes	x1
28	FMT_16_16_EXPAND (YX)	No	Yes	No	Yes	x2
29	FMT_16_16_16_16_EXPAND (WZYX)	No	Yes	No	Yes	x4
30	FMT_16_FLOAT (X)	No	Yes	No	No	x1
31	FMT_16_16_FLOAT (YX)	Yes	Yes	No	No	x1
32	FMT_16_16_16_16_FLOAT (WZYX)	Yes	Yes	No	No	x2
33	FMT_32 (X)	Yes	Yes	No	Yes	x1
34	FMT_32_32 (YX)	Yes	Yes	No	Yes	x2
35	FMT_32_32_32_32 (WZYX)	Yes	Yes	No	Yes	x4
36	FMT_32_FLOAT (X)	Yes	Yes	No	No	x1
37	FMT_32_32_FLOAT (YX)	Yes	Yes	No	No	x2
38	FMT_32_32_32_32_FLOAT (WZYX)	Yes	Yes	No	No	x4
39	FMT_32_AS_8 (X)	No	Yes	Yes	Yes	x1
40	FMT_32_AS_8_8 (X)	No	Yes	Yes	Yes	x1
41	FMT_16_MPEG (X)	No	Yes	No	Yes	x1
42	FMT_16_16_MPEG (YX)	No	Yes	No	Yes	x1
43	FMT_8_INTERLACED (X)	No	Yes	Yes	Yes	x1
44	FMT_32_AS_8_INTERLACED (X)	No	Yes	Yes	Yes	x1
45	FMT_32_AS_8_8_INTERLACED (X)	No	Yes	Yes	Yes	x1
46	FMT_16_INTERLACED (X)	No	Yes	No	Yes	x1
47	FMT_16_MPEG_INTERLACED (X)	No	Yes	No	Yes	x1
48	FMT_16_16_MPEG_INTERLACED (YX)	No	Yes	No	Yes	x1
49	FMT_DXN (YX)	No	Yes	Yes	Yes	x1
50	FMT_8_8_8_8_AS_16_16_16_16 (WZYX)	No	Yes	Yes	Yes	x2
51	FMT_DXT1_AS_16_16_16_16 (WZYX)	No	Yes	Yes	Yes	x2
52	FMT_DXT2_3_AS_16_16_16_16 (WZYX)	No	Yes	Yes	Yes	x2
53	FMT_DXT4_5_AS_16_16_16_16 (WZYX)	No	Yes	Yes	Yes	x2
54	FMT_2_10_10_10_AS_16_16_16_16 (WZYX)	No	Yes	Yes	Yes	x2
55	FMT_10_11_11_AS_16_16_16_16 (ZYX)	No	Yes	No	Yes	x2
56	FMT_11_11_10_AS_16_16_16_16 (ZYX)	No	Yes	No	Yes	x2
57	FMT_32_32_32_FLOAT (ZYX)	Yes	No	No	No	x3
58	FMT_DXT3A (X)	No	Yes	Yes	Yes	x1
59	FMT_DXT5A (X)	No	Yes	Yes	Yes	x1
60	FMT_CTX1 (YX)	No	Yes	Yes	Yes	x1
61	FMT_DXT3A_AS_1_1_1_1 (WZYX)	No	Yes	No	Yes	x1
…						
63	RESERVED (Not for use)	N/A	N/A	N/A	N/A	N/A

Notes:

1.) *'s in DATA_FORMAT indicates formats in specific locations for 2D compatibility and should not be moved.

2.) FMT_1_REVERSE is same format as FMT_1 but with bit order within a byte swapped.

3.) Texture data is expanded going into the L2 cache.

4.) Vertex data is not expanded going into the L2 cache. Expansion/conversion is done on way to GPRs.

5.) Use DST_SEL_n in instruction or constant to modify routes from those shown above.

6.) Compressed Texture Formats

Uncompressed to L2 cache using standard fixed decompression algorithm. From the L2 cache POV the components are stored as unsigned RF [0..1].

TP will still do number format conversion on these formats. Software will typically/may want to program the following conversions:

FMT_DXT1, FMT_DXT2_3, FMT_DXT4_5 = unsigned RF [0..1]

FMT_DTN = unsigned RF biased [-1..1] (ZCMO or NZ)

FMT_DXT3A = unsigned RF [0..1] - (FYI, this format aka u4)

FMT_DXT5A = unsigned RF [0..1] (perhaps also useful as unsigned RF biased [-1..1] (ZCMO or NZ)) - (FYI, this format aka ATI1N, BXT1)

FMT_CTX1 = unsigned RF [0..1] - (FYI, this format aka BXT3)

7.) Degamma only valid for unsigned RF numbers (including FMT_DXT*), since the degamma logic assume the data is unsigned RF and remaps as such.

8.) Enabling degamma in any channel does not change the format in which the data is stored in the L2 cache. This enables software to make trade-offs between high quality

9.) #39-48 are expected to be video/multimedia specific formats.

10.) Interlaced formats: Z=0.0f for even, Z=1.0f for odd.

11.) FMT_24_8* formats: only fetches the 24b, in either fixed or floating point format.

FMT_24_8 should be set to unsigned RF to match what the RB represents.

FMT_24_8_FLOAT will get automatically converted to the [0, 1] range.

In order to read the stencil bits, a separate constant must be used, using FMT_8_8_8_8 and reading the X channel only.

12.) FMT_Cr_Y1_Cb_Y0_* and FMT_Y1_Cr_Y0_Cb_* have 8 bit components.

13.) DATA_FORMATs with fixed numbers formats. That is, they are hardcoded as indicated below and thus ignore

*32_FLOAT = 32 bit IEEE float, sE8M23, used directly without conversion

*16_FLOAT = sE5M10, expanded to 32_FLOAT on way to GPR

FMT_24_8 = 24 bit unsigned RF [0..1], 8 bits dropped

FMT_24_8_FLOAT = unsigned E4M20, expanded to 32_FLOAT on way to GPR and 8 bits dropped

*16_EXPAND = FLOAT_16 clamped and converted to signed fixed point 16.16 [-32K..32K). This permits filtering for 16 bit floating point components.

16_MPEG = (16 bit signed integer, clamp to [-256..255] before filtering)

14.) FMT_24_8_FLOAT will ignore filtering for 24 bit floating point depth. Filtering will be honored for unsigned integer stencil.

15.) Unfiltered formats only support Point as filter types (min/mag/mip), with anisotropy disabled. Results are otherwise undefined, but HW is not permitted to crash.

16.) FMT_DXT3A_AS_1_1_1_1 is akin to DXT3A in the sense it is 4x4 elements of 4b each. The difference is every bit represents a different channel.

R400 Numbers

```
// Fixed to Float Conversions

float Tf;  // floating point output
unsigned int N;  // size of component in bits
```

unsigned integer

```
  // Range: [0..(1 << N))
  unsigned int Ti;
  Tf = (float)Ti;
  // ex. N=8 -> 0=0.0,  255=255.0
```

signed integer

```
  // Range: [-(1 << (N-1))..(1 << (N-1)))
  signed int Ti;
  Tf = (float)Ti;
  // ex. N=8 -> -128=-128.0,  127=127.0
  // it's expected the HW implementation will invert the high
  // order bit and treat result as an unsigned integer biased
```

unsigned integer biased

```
  // Range: [-(1 << (N-1))..(1 << (N-1)))
  unsigned int Ti;
  Tf = (float)Ti - (float)(1 << (N-1));
  // ex. N=8 -> 0=-128.0, 128=0.0, 255=127.0
```

unsigned integer gamma'd

```
  // degamma ignored, same as unsigned integer
  // Range: [0..(1 << N))
  unsigned int Ti;
  Tf = (float)Ti;
  // ex. N=8 -> 0=0.0,  255=255.0
```

unsigned RF

```
  // Range: [0..1]
  unsigned int Ti;
  Tf = (float)Ti / (float)((1 << N) - 1);
  // ex. N=8 -> 0=0.0,  255=1.0
```

signed RF (ZCMO - ZERO_CLAMP_MINUS_ONE)

```
  // Range: [-1..1]
  signed int Ti;
  Tf = (Ti == -(1 << (N-1))) ? -1.0f : (float)Ti / (float)((1 << (N-1)) - 1);
  // ex. N=8 -> -128=-1.0, -127=-1.0, 0=0.0, 127=1.0
  // it's expected the HW implementation will invert the high
  // order bit and treat result as an unsigned RF biased (ZCMO)
```

signed RF (NZ - NO_ZERO)

```
  // Range: [-1..1]
  signed int Ti;
  Tf =  (float)(2*Ti + 1) / (float)((1 << N) - 1);
  // ex. N=8 -> -128=-1.0,  0=1/255.0, 127=1.0
  // it's expected the HW implementation will invert the high
  // order bit and treat result as an unsigned RF biased (NZ)
```

unsigned RF biased (ZCMO - ZERO_CLAMP_MINUS_ONE

```
  // Range: [-1..1]
  unsigned int Ti;
  Tf = (Ti == 0) ? -1.0f : ((float)Ti - (float)(1 << (N-1))) / (float)((1 << (N-1)) - 1);
  // ex. N=8 -> 0=-1.0, 1=-1.0, 128=0.0, 255=1.0
```

unsigned RF biased (NZ - NO_ZERO)

```
  // Range: [-1..1]
  unsigned int Ti;
  Tf = ((float)Ti / (float)((1 << N) - 1)) * 2.0f - 1.0f;
  // ex. N=8 -> 0=-1.0, 128=1/255.0, 255=1.0
```

unsigned RF gamma'd

```
  // Range: [0..1]
  unsigned int Ti;
  Tf = UndoGamma((float)Ti / (float)((1 << N) - 1));
  // ex. N=8 -> 0=0.0,  255=1.0
```

```
// It is expected that the HW implementation will appromiate this function with a LUT
// Any implementation should return 0 for 0 and 1 for 1
float UndoGamma( float c )
{
    if( c <= 0.03928 )
    {
        return (c / 12.92);
    }
    else
    {
        return pow( (c + 0.055)/1.055, 2.4 );
    }
}
```

example: N=4

decimal	hex	binary	unsigned INT	signed INT	unsigned INT biased	unsigned INT gamma'd	unsigned RF	signed RF (ZCMO)	signed RF (NZ)	unsigned RF biased (ZCMO)	unsigned RF biased (NZ)	unsigned RF gamma'd
0	0	0000	0	0	-8	0	0	0	1/15	-1	-1	UndoGamma(0)
1	1	0001	1	1	-7	1	1/15	1/7	3/15	-1	-13/15	UndoGamma(1/15)
2	2	0010	2	2	-6	2	2/15	2/7	5/15	-6/7	-11/15	UndoGamma(2/15)
3	3	0011	3	3	-5	3	3/15	3/7	7/15	-5/7	-9/15	UndoGamma(3/15)
4	4	0100	4	4	-4	4	4/15	4/7	9/15	-4/7	-7/15	UndoGamma(4/15)
5	5	0101	5	5	-3	5	5/15	5/7	11/15	-3/7	-5/15	UndoGamma(5/15)
6	6	0110	6	6	-2	6	6/15	6/7	13/15	-2/7	-3/15	UndoGamma(6/15)
7	7	0111	7	7	-1	7	7/15	1	1	-1/7	-1/15	UndoGamma(7/15)
8	8	1000	8	-8	0	8	8/15	-1	-1	0	1/15	UndoGamma(8/15)
9	9	1001	9	-7	1	9	9/15	-1	-13/15	1/7	3/15	UndoGamma(9/15)
10	A	1010	10	-6	2	10	10/15	-6/7	-11/15	2/7	5/15	UndoGamma(10/15)
11	B	1011	11	-5	3	11	11/15	-5/7	-9/15	3/7	7/15	UndoGamma(11/15)
12	C	1100	12	-4	4	12	12/15	-4/7	-7/15	4/7	9/15	UndoGamma(12/15)
13	D	1101	13	-3	5	13	13/15	-3/7	-5/15	5/7	11/15	UndoGamma(13/15)
14	E	1110	14	-2	6	14	14/15	-2/7	-3/15	6/7	13/15	UndoGamma(14/15)
15	F	1111	15	-1	7	15	1	-1/7	-1/15	1	1	UndoGamma(1)
range			[0..16)	[-8..8)	[-8..8)	[0..16)	[0..1]	[-1..1]	[-1..1]	[-1..1]	[-1..1]	[0..1]

R400 Border Color

	ABGR Black	ABGR White	ACbYCr Black	ACbCrY Black	FORCE_BC_W_TO_MAX
unsigned / unsigned gamma'd	WZYX = (0)	WZYX = (1)	W=(0), Z=1(0), Y=0001(0), X=1(0)	W=(0), Z=1(0), Y=1(0), X=0001(0)	W=(1)
signed / unsigned biased / 16_EXPAND	WZYX = (0)	WZYX = 0(1)	WZYX = (0)	WZYX = (0)	W=0(1)
16_MPEG	WZYX = (0)	WZYX = (0)11111111	WZYX = (0)	WZYX = (0)	W=(0)11111111
_FLOAT	WZYX = 0.0f	WZYX = 1.0f	WZYX = 0.0f	WZYX = 0.0f	W=1.0f

Notes:

1.) The above values indicate the values used by the filter for border texels. Debias, gamma correction and MPEG clamping have already been applied.

2.) ()'s indicate a repeating value.

3.) There is no filtering of floating point values.

4.) Because certain number attributes like signed and unsigned are selectable per component, the border color used for each componet will be selected appropriately.

R400 Features, Functionality, Caveats, Restrictions and Notes

General Notes:

1.) all specified as big endian (WZYX, W in high channel). WZYX is interchangable with ABGR. Anything not explictly defined, returned, or otherwise should be assumed to be undefined/garbage from a software perspective. In reality, the hardware may have well defined behavior for power or simplictiy reasons (see HW specs), but software should avoid relying on it.

2.) all signed fixed point numbers are considered to be 2's complement

R400 Basics

1.) Macro tile on r400 = 32x32 pixels
2.) L2 texture cache is 32 Kbytes on R400

Maximum Texture Sizes				
	w/o border, normalized	w/ border, normalized	w/o border, unnormalized	w/ border, unnormalized
1D	8K	8K-2	16M	16M-2
2D	8Kx8K	(8K-2)x(8K-2)	8Kx8K	(8K-2)x(8K-2)
3D (<=64 bit texels)	2Kx2Kx1K	2Kx2Kx1K	2Kx2Kx1K	2Kx2Kx1K
3D (128 bit texels)	2Kx2Kx1K	2Kx2Kx(1K-2)	2Kx2Kx1K	2Kx2Kx(1K-2)
Cube	8Kx8K	(8K-2)x(8K-2)	8Kx8K	(8K-2)x(8K-2)
Stack	8Kx8Kx64	(8K-2)x(8K-2)x64	8Kx8Kx64	(8K-2)x(8K-2)x64
Noise (not supported on r400)	4Kx4Kx4K	N/A	4Kx4Kx4K	N/A

Addressing Support		
	Normalized	Unnormalized
Wrap	Yes	Treat coord as normalized
Mirror	Yes	Treat coord as normalized
Clamp to last texel	Yes	Yes
MirrorOnce to last texel	Yes	Treat coord as normalized
Clamp half way to border color	Yes	Yes
MirrorOnce half way to border color	Yes	Treat coord as normalized
Clamp to border color	Yes	Yes
MirrorOnce to border color	Yes	Treat coord as normalized
Mipmapping	Yes	No

NP2 Texture Support		
	Basemap	Mipmaps
Padding	Next tile (32 texels)	Next power of 2
Wrap	Yes	Yes
Mirror	Yes	Yes
Clamp to last texel	Yes	Yes
MirrorOnce to last texel	Yes	Yes
Clamp half way to border color	Yes	Yes
MirrorOnce half way to border color	Yes	Yes
Clamp to border color	Yes	Yes
MirrorOnce to border color	Yes	Yes

Notes:

1.) Textures with borders do not have the border size included in the width/height/depth. This is determined by the BORDER_SIZE field. Eg. A 1024x1024 texture with a border would have width=1024 and height=1024 with BORDER_SIZE=1. However, the padding and alignment constraints are that of a NP2 texture of 1026x1026 dimensions.

2.) NP2 mipmaps are floored in size. A texture with a 51x51 base map would have a mipmap chain of sizes 25x25, 12x12, 6x6, 3x3, 1x1.

Alignment Constraints (minimum)			
	1D texture	2D/3D texture	Vertex
Local/Video tiled	4K bytes	4K bytes	4 bytes
System/AGP tiled	4K bytes	4K bytes	4 bytes
Local/Video linear	4K bytes	4K bytes	4 bytes
System/AGP linear	4K bytes	4K bytes	4 bytes

Notes:

1.) 2D/3D texturing from linear surfaces may incur a large performance penalty.

2.) 1D tiled and linear formats are the same.

3.) Data of different formats cannot occupy the same 4k block of memory.

Anisotropic Filtering Support			
	1D texture	2D texture	3D texture
Point	Yes	Yes	No
Linear	Yes	Yes	No
Filter2x4_Sym	Yes	Yes	No
Filter2x4_Asym	Yes	Yes	No
Filter4x2_Sym	Yes	Yes	No
Filter4x2_Asym	Yes	Yes	No
Filter4x4_Sym	Yes	Yes	No
Filter4x4_Asym	Yes	Yes	No

Notes:

1.) If anisotropy in enabled on 3D textures, it only happens on the X and Y dimension (we don't sample more than 2 layers in Z)

R400 Open Questions

1.) Do we need to support the SGN_NORM_ZERO mode for vertex data?
Status: **Closed**. In the 1/2/2 and 1/9/2 meetings with MS they agreed with us and signed normalized vertex data will map the the same way as signed textures.
Answer: No. Need to make sure this ends up in refrast.

2.) Do we need a bit saying if texture/vertex is in local or system memory, or can it be devined from the base address?
Status: **Closed**.
Answer: The TC/MH will be able to determine this. The as specified tiling bit will be all that is needed.

3.) Do we need to support FMT_4?
Status: **Closed**. Will only support if absolutely necessary for ClearType
Answer: No. ClearType will not be using this format.

4.) Do we need to support FMT_4_4?
Status: **Closed**.
Answer: No. Multimedia has signed off on this. It's believed that FMT_8 with EXP_ADJUST can be used to emulate it when necessary.

5.) Do we need to support FMT_3_3_2?
Status: **Closed**.
Answer: No. No group seems to need this anymore.

6.) How are we going to support D3D source ColorKey?
Status: **Closed**
Answer: To be handled by software with shader assisted filtering if necessary.

7.) Will shadow and depth fetches with border color work correctly? Do they have too?
Status: **Closed**.
Answer: Yes mostly and Yes. No problem with depth textures. HW "zero" and "one" will work as expected. Shadows are a little different since the border color will be considered a post shadow compare value instead of a z value. With a bit of driver fiddling, almost every case can be made to work for zero and one. It's expected that the edge cases can be ignored, but if absolutely necessary the non HW border color shader could can be used. Further documentation on this is coming.

8.) Do we support NP2 textures with borders?
Status: **Closed**.
Answer: Yes, see table on Features page.

9.) Do we include the border in texture size?
Status: **Closed**.
Answer: No, see table on features page.

10.) We can only support up to 8k texture sizes. This means that we can't support an 8k texture with a border. Is this a problem?
Status: **Closed**.
Answer: No, this is a reasonable limitation.

11.) We allow the basemap and mipmap chains to be in different memories (local/system). However, we only have one select for tiled and endian which places some restrictions. Is this acceptable? Note that question 2.) may impact this.
Status: **Closed**.
Answer: They must be the same.

12.) The packed pixels extensions allows for textures of format 5_5_5_1 and 10_10_10_2. Do we want to support these?
Status: **Closed**.
Answer: This is really only an issue for OGL. The OGL driver will do a bit rotation to get into 1_5_5_5 or 2_10_10_10 format and use DST_SEL to route RGB appropriately.

13.) Do we want to remove the FetchCubeMap opcode and add CubeMap to the DIM field?
Sataus: **Closed**.
Answer: Yes. This is more consistent with the APIs and is necessary to support generic border color on a cubemap since we can't do a fetchcube and fetchwhite at the same time.

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	13 March, 2001	[date \@ "d MMMM,	Version 0.26	1 of 123

AUTHORS: John Carey

ISSUE TO	COPY NO.

Specification of the Command Processor Architecture Crayola

THIS DOCUMENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATI TECHNOLOGIES INC. THROUGH UNAUTHORIZED USE OR DISCLOSURE.

APPROVALS

Name/Dept	Signature/Date

Remarks:

"Copyright 2001, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished work created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this unpublished work. The copyright notice is not an admission that publication has occurred. This work contains confidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or transmitted in any form or by any means without the prior written permission of ATI Technologies Inc."

1. Table of Contents

2. Table of Figures

3. Revision History

05 July 01	Version 0.00	Initial version from version 1.14 of the R300 CP Specification. Updated Open-Issues section. – J. Carey
27 February 02	Version 0.02	Note: Version 0.01 is not formally released. Removal of TID from MH Interface, Added internal "helper" packets, Added 2D implementation notes as appendix. Removed CP_STATE_SUBBLK_CNTL register. Sub-block sizes are in the Set_State packet. Details of Multi-Pass operation added. Pre-Fetch Parser Algorithm.
28 March 02	Version 0.03	Renamed 2D Scratch Memory to 2D Defaults Memory. Updated interface to Memory Hub based on Ken Correll's e-mails. Updated registers and added BIOS scratch registers. Renamed TIMEOUT to INTERVAL in time-out counters. Updated interface signals. Added dynamic clocking disable signal to the CP_DEBUG register and added description in the Power Management Support section of the document. Brush, Palette, and Immediate data base addresses are quad-octword aligned. Updated information on the 2D Booleans.
October, 29 02	Version 0.04	Interface Updates. Added CP_MH_reqmem signal. Display "end of frame" counters are changed to "start of frame" counters. Update power management signals to/from MH. Updated registers. Update register names. Updated Scratch Memory Contents. Added 8 more bios scratch registers. DMA engine waits for MIU to finish writes before raising the interrupt. Removed CP_CG_2D_Mode signal – No longer needed per power management meeting on 05-27-2002. Updated initialization sequence. Added Performance Counter.
Nov. 11 02	Version 0.05	Updated CP_STAT and Added CP_RT_STAT register.
Nov. 14 02	Version 0.06	Update 2D B0 Boolean Decoding for "No Brush" and Update CP_DMA_STAT format.
Nov. 20 02	Version 0.07	Set 2D B7 Boolean if Src_Type=8bpp TLU and Dst_Type==32bpp
Nov. 21 02	Version 0.08	Reverted Version 0.07.
Dec. 3 02	Version 0.09	Update CP_IB2D_BASE, CP_RB_BASE, CP_IB1_BASE, CP_IB2_BASE and CP_RT_BASE to be 31:5.
Jan. 8, 03	Version 0.10	Updated Queue Thresholds and Available Counts per Memory Size Decrease.
Jan. 29, 03	Version 0.11	Added note to CP_DEBUG that its unused bits should not be optimized out.
Jan. 30, 03	Version 0.12	Update width of microcode write and read registers and add note on microcode loading.
Feb. 3, 03	Version 0.13	Add Clock Gating Diagram. Add note that debug bus data I/O is asynchronous.
March 12, 03	Version 0.14	Add Pre-Fetching Disable Registers. See the Pre-Fetch Disable.doc document.
March 20, 03	Version 0.15	Updated width of the fields in the CP_NON_PREFETCH_CNTRS register.
March 21, 03	Version 0.16	Removed PREFETCH_DISABLE_OVERRIDE from CP_Debug register.
March 24, 03	Version 0.17	Added Scratch Compare Interrupt.
April 04, 03	Version 0.18	Added Boolean and Loop Matching Signals to State Debug Register.
April 14, 03	Version 0.19	Miscellaneous Updates.
April 16, 03	Version 0.20	Add Debug Registers for Tag Return Status from Memory Hub.
April 17, 03	Version 0.21	Add interrupt for Type-0 or Type-1 packets found in an Indirect Buffer.
May 16, 2003	Version 0.22	Clarification of Pre-Write-Timer and Pre-Write-Limit usage.
May 29, 2003	Version 0.23	Added oper=comp.
June 4, 2003	Version 0.24	Added Debug bit to CP_DEBUG register to simplify flow control in the ME pipeline.
July 1, 2003	Version 0.25	Added Section to Host Programming Considerations.
July 1, 2003	Version 0.26	Min / Max Function Clarification.

4. Open Issues / Items to Complete (Updated: 05-10-2002)

1) What should the "Protected Mode" address range be for PM4 Error Checking? *A: This is programmable with the Me_Init packet.*

2) Need to add programmable count for the number of DWORD writes that can be issued to the Memory Hub before sending a write confirm. This allows for multiple characters to be sent between write confirmations for Small_Text and HostData_Blt packets. *A: This is programmable with the Me_Init packet.*

3) Should there be both a "single" and "dual" ring mode for managing the Instruction Memory. *A: 04-10-2002: S. Morein, A. Gruber, M. Fowler, P. Rogers discussed this and all agreed that it is o.k. to support only one mode. The mode will be "dual ring".*

4) Should the CP's Micro Engine support byte enables for transactions to the RBBM? Current chips do not, so unless told otherwise the BE from the ME will not be supported. Note that the DMA engines do use the BE to the RBBM, so this is why the CP has the BE on its interface. *A: The CP's micro engine will not support byte enables.*

5) Should "drain-o" event FIFO replace the Pipeline Store DWORD function – Need to discuss with Phil. *A: 04-03-3002 – Discussion with Phil in Orlando. He is o.k. with the "drain-o" event replacing the pipelined store DWORD function. But the events should also write to registers.*

6) Addition of RT stream set/reset flop in the CP's micro engine. Real-time stream sets the bit when the ME starts processing the RT stream. Event(clear_rt_busy) in the real-time stream is used to clear the bit. The bit is output to the RBBM as CP_RBBM_rt_idle.

7) Define 2D support requirements for the CRAYOLA CP (I.e. translation of legacy 2D PM4 command packets to shader code equivalent). *A. Documented in 2D Appendix and PM4 Specification*.

8) Define Power Management Support Requirements in CP (i.e. Block Shutdown & Voltage/Clock Determination). Pete P. will call a meeting for this topic. *A: Meeting held 01-18-2002. CP control of power management will be within command streams.*

9) Is it a problem that the writes to the MH take 9 clocks – Why not separate the write data from the write address? *A: The CP writes DWORDs to communicate semaphores back to the SW for synchronization. There is not a performance requirement for this case, so it is o.k. for this to take at least 9 clocks.*

10) Does the MH handle the ordering return data from fast requests to the Video Memory that may come after slower requests to the AGP? *A: No. Tag information helps CP to re-order the data.*

11) Define Support of Microsoft High-Priority Streams if different from Real-Time Streams. *A: Meeting with MicroSoft indicated that this functionality is covered in the definition of the Real-Time Streams.*

12) Define Command (PM4) Error Checking Support Requirements. *A: Added. See specification.*

13) Finalized Ring Buffer RPTR update mechanism for CRAYOLA. *A: The CP will issue a Pipelined Store DWORD (PSD) write every time it passes a granularity threshold. When the CP gets a pulse back from the RC, it will then write the RPTR.*

14) The CP may be connected to the Memory Hub via a daisy chain interface. Michael Doggett with discuss this possibility with S. Morein. Until heard otherwise, it is assumed that the CP has a dedicated interface to the Memory Hub. *A: 01-14-2002. CP is a dedicated interface to the Memory Hub, but implements the DCC protocol. If the CP is added to a DCC chain, then the DCC wrapper can be used around the CP.*

15) Define de-allocation protocol for State Management. *A: For Instruction (Vertex and Pixel) Memory de-allocation, the CP receives two 3-bit buses from the Sequencer. The contents of these buses are the context that the Sequencer is processing. The CP detects a change on these buses and de-allocates the code associated with N-1 context. For Renderstate, the CP receives a 3-bit bus from the RB and a "valid" bit. When RB pulses the valid bit, the CP will de-allocate the context that is indicated on the 3-bit bus.*

16) Define requirement to support Per-Draw Command Constants. *A: 10-29-2001 Per S. Morein: These are no interest to the Driver folks so it is no longer a requirement for the CP.*

17) Define Implementation of OpenGL Immediate Mode Support in CP. *A: CP skips immediate data for 2D packets. The 3D draw packets have been removed for CRAYOLA.*

18) Phil Rogers is beginning to think about a very radical change to the way that we give commands to the chip. This involves no Ring or Indirect Buffers. Instead, the commands are PIO'd to the CP to dedicated command buffers. The writes to these buffers will spill over to buffers that are in video memory. The CP would have logic to assemble the command data to the Micro-Engine (i.e. From command FIFOs or from Video Memory). *A: This is Optimized PIO mode which was decided not to be a requirement in the CRAYOLA.*

19) Define DMA function requirements for CRAYOLA and resolve whether the CP still contains the DMA function. *A: POR from the CRAYOLA Summit (09-06-2001) is that the GUI/VID DMA functionality is NOT in the design. Real Time Streams can support this functionality and generate an interrupt via a software interrupt mechanism.*

20) Define CP Output Bandwidth Requirements. (Khan output was increased to 128/bits per clock.) *A: 09-19-2001 CP output data path is 128-bits to the PA and legacy 2D unit. The output data path to the other units is 32-bits.*

21) Define MC->CP interface width. *A: 09-19-2001 The Input path from the Memory Hub should be 128-bits to provide for the index bandwidth requirements.*

22) Determine WAIT_UNTIL/ISYNC logic requirements and determine if they are in the CP or RBBM. *A: 08-23-01 S. Morein states that they should be in the CP. The read operations will however travel down the same path as the initiators so the read of the status will reflect any/all initiators that proceeded.*

23) Look-Ahead process needs to be added to fetcher logic to find next Indirect Buffer to fetch. When the first IB's last read request is issued, the next IB to be fetched will start. Data will accumulate in the respective IB memory behind the first. *A: 09-21-2001 Documented in this specification.*

24) Multi-pass Overflow Loop logic (S. Morein). *A: 09-20-2001 Documented in this specification.*

25) Does the CP Micro-Engine Require Floating Point or Data Conversion Hardware? *A: No per S. Morein (08-23-2001: 2D code is all in fixed point and if any conversions are required, the data can be pre-converted. Besides, the CP does not handle any of the data for the hardware below.*

26) Addition of a 32-bit write code for the CMD field to the MH. *A: The writing of a DWORD is assumed to not be a performance issue. Therefore it is o.k. for this to take 9 clocks to issue from the CP. It is planned that the CP will try to group multiple consecutive writes it has pending if these writes fall into the same double-octword.*

5. Overview

The Command Processor is a programmable processor that is meant to provide some on-chip intelligence for a Graphics Controller device. The CP architecture has been approached as a special-purpose computing engine, targeted at fetching and interpreting a PROMO4 command stream.

The Command Processor takes on several tasks in a typical Graphics Controller:

1) Acts as a receiver of command streams from the video and graphics device driver(s) running on the host CPU. These command streams are read from system memory using bus-mastering on the PCI or AGP bus

2) Parses and interprets command packets, as defined in the PM4 Specification, and writes the parsed data to internal modules of the Graphics Controller.

3) State Management for the Graphics Controller – Managing both Real-Time and Non-Real-Time State Contexts.

4) Real-Time Stream Event Engine and Command Parsing.

5.1 Assumptions

1. There is no requirement to render without the use of the CP (i.e. CP Bypass Rendering & State Management in Driver). There is nothing in the design of the CP or RBBM that precludes this however.
2. Incremental and Non-Incremental State Updates from the Driver can be inter-mixed.
3. It is assumed that Type-0 incremental updates will be written to context #0 of the GFX decode space. Furthermore, they will not start outside and end in context #0 of the GFX decode space nor will they start inside and end outside context #0 the GFX decode space.
4. The design of the command stream pre-fetching logic is implemented as a fixed state machine in the Crayola. It is therefore not possible to add Type-3 packets that require pre-fetched data without modifying the RTL of the CP.

5.2 Items Added to CP Design for Crayola

1. State Management Control
2. “Viz Query A” Support
3. Real-Time Stream Support
4. Pre-Fetching of Data Streams
5. 2D Packet Translation to Shader Implementation
6. PM4 Error Checking
7. Reg->Reg DMA Transactions
8. Multi-Pass Command Stream Support
9. Timestamp Synchronization Events
10. Implicit Sync Control for 2D/3D Switching – Function moved from RBBM
11. More Generalized Micro Engine Design

5.3 Items Removed From CP Design for Crayola

1. GUI / VID DMA Engines per Crayola Summit 09-06-2001. Replaced with single DMA Engine design.
2. “Vector Mode” -- 128-bit Immediate Data Output from CP. In all the performance cases, the CP does not supply the immediate data to the VGT. Therefore the output from the CP through the RBBM is only 32-bits. The interface from the Memory Hub is also reduced to 32-bits.
3. Pipelined Timestamp that was in the R300 has been replaced with the Timestamp Synchronization Events (VsDone, PsDone, and Cache Flush).
4. Push Mode (PIO) has been removed, leaving only Bus Mastering for the CP to fetch command streams.

5.4 Bandwidth & Performance Analysis

5.4.1 CP Requests to Memory BW Analysis (Update: 10-22-2001)

Assume that the dominant Read request Client is for the Indirect Buffer Data.

Assume that the dominant Write Client is to write Constant Data.

Assume 1.6 GBytes/sec available bandwidth on this bus.

Assuming we need to make 400M peak Triangles/Sec with index data.

Assume 96 bits of indices/triangle (32-bit indices)

So, each 256-bit read request is for ~2.67 Triangles.

Which is ~150M Read Requests/Sec.

This leaves ~250M clocks/sec for Constant Memory writes.

It takes the CP 18 clocks/constant to write the data (Two 256-bit write transfers with address phases).

This provides a constant memory write BW of ~13.8M Constants/Sec (~888 MBytes/Sec MAX).

Data Point (From Dave Gosselin):

“The most stressing character draw from the Island demo is roughly 70 state changes a frame at about 30 frames per second so about 2100 state changes. This for just the characters, Alex would need to tell you about the terrain since I'm not as familiar with that side. These draws update roughly 96 constant vectors which are 4 32bit floats”.

Island Demo Characters: 2100 State/Sec * 96 Constants/State * 16 Bytes/Constant = 3.076 MBytes/Sec

From Alex Vlachos:

Either most or none of the constant store will change between draw calls. The 2 main things (besides shaders) that break up our draw calls are matrix changes for characters and texture changes. However, I can see us wanting to update smaller portions as we become more comfortable writing shaders and design more flexible shader libraries at the app level.

But for the majority of the time, most or none would be my answer....especially for legacy apps.

5.4.2 Packet Performance (Updated: 04-10-2002)

Type-0 packets processed through the Micro Engine (ME) can take at most 2 clocks per DWORD per S. Morein on 04-10-2002.

State packets should get give a DWORD per clock.

Packet Locations for Performance Cases:

- Set_State – Packets will be in the frame buffer (Local Memory) for the performance case. (P. Rogers)
- Load_Constant_Context – These may change frequently and be bigger, so they may be in AGP for the performance case. (P. Rogers)
- 2D Packets – These will either be in the Ring Buffer (AGP) or in an Indirect Buffer (Local Memory). (P. Rogers)
- 3D_DRAW_INDX – This packet is for performance. This packet is important for cases with lots of lists of a small number of indices in each list. This is a common situation in benchmarks. (P. Rogers)

5.4.3 Analysis for Outstanding Requests To Cover AGP Latency (Updated: 03-13-2003)

Clock Ratio = 444Mhz / 66MHz = 6.72 => AGP provides a Double Octword every 6.72 core clocks to Memory Hub

AGP Latency = 35 System Clocks * 6.72 = 235 Core Clocks

So, CP needs 235 DWORDs storage to hide full AGP latency. However, this is not needed as a result of experiments on R300. The following text is a summary from an e-mail discussion:

In an attempt to summarize:

1. Table from Phil's e-mail:

Mem	Depth	Use
AGP	200	PM4 RingBuffer (subject to Jeffrey's testing)
AGP	200	Indirect Buffer 1
LM	150	Indirect Buffer 2
LM	150	State
AGP	300	Real Time CBs
FB	70	Real Time State

----

2. The R300 experiment that was run for 2D showed that there was less than a 1% difference in the scores between a ring buffer setting of 0x0A (80 DWORDs) and 0x5C (736 DWORDs).

3. Also, Andy commented that the AGP latency that should be used is 30-35 clocks instead of 50. This gives a DWORD storage requirement of 235 instead of 336.

So would you agree with this update:

Mem	*Depth*	*Use*
AGP	*128*	*PM4 RingBuffer {Gave 512 - IB1 - IB2, which is still more than 80 DWORDs from R300 experiment}*
AGP	*232*	*Indirect Buffer 1 {Adjusted up to reflect 35 clock agp latency & Rounded to multiple of 8 DWORDs}*
LM	*152*	*Indirect Buffer 2 {Rounded to a multiple of 8 DWORDs}*

= 512

Mem	*Depth*	*Use*
LM	*152*	*State {Rounded to a multiple of 8 DWORDs}*
AGP	*288*	*Real Time CBs {Gave 512 - State - RT State, which is greater than 235 storage for 35 clock latency}*
FB	*72*	*Real Time State {Rounded to a multiple of 8 DWORDs}*

= 512

6. Host Programming Model Description

This section describes the manner in which the host CPU communicates with the graphics controller chip.

6.1 Pull Model

The *Pull Model* utilizes bus-mastering on the part of the graphics controller, as it actively goes out and reads from an area of system memory in which the host CPU has previously placed command information. This information is in one of two forms:

1) A sequence of register writes to setup the state of a processing engine on the graphics controller, and then starting the engine running. Typically, engines are started as a side-effect of writing to a special "trigger" or "initiator" register.
2) A sequence of *Command Packets*, which are a "compressed" way of conveying the command information to the graphics controller, relying on an intelligent processor in the graphics controller to convert the command packets into register writes to other processing engines in the graphics controller.

An important part of the pull model is how the host and the graphics controller manage access to the shared buffer in system memory. This is discussed in the following section.

6.2 Ring Buffer Management

When the Graphics Controller is set to operate in the bus-mastering mode (pull model), the host application, say a driver, has to allocate a block of system memory as a buffer for the *command packets* it issues to the Graphics Controller. The command packets, or simply packets, instruct the Graphics Controller to carry out operations such as drawing objects on the screen. This memory block is treated as if it is a ring that allows the packets to be placed into and taken away from the memory in a circular manner, thus the name *Ring Buffer*.

The Ring Buffer is a shared memory space between two cooperating processors. It is used to implement one-way communication from the Host processor (the Writer) to the Graphics Controller (the Reader). Each processor must maintain the state that it believes that the Ring Buffer is in. The state is composed of:

- ❑ Buffer Base: The address of the beginning of the buffer.
- ❑ Buffer Size: The size of the buffer.
- ❑ Write Pointer: The address that the Host is writing to.
- ❑ Read Pointer: The address that the Graphics Controller is reading from.

In order for the Ring Buffer to work properly, both processors must maintain a consistent view of this state. The Buffer Base and Buffer Size are generally initialized when the system is first brought-up, and rarely changed after that point. It is a simple task to initialize both the Reader's and the Writer's copies of this state. The Read and Write Pointers, on the other hand, change quite frequently as the Ring Buffer is in operation. In order to achieve consistency, when the Writer (the host) updates the Write Pointer, he must send that value to the Reader's (the Graphics Controller's) copy of the Write Pointer. And similarly, when the Reader updates the Read Pointer, he must send that value to the Writer's copy of the Read Pointer.

Packets are placed into the memory block, or buffer, from the beginning towards the end, i.e., from lower addresses toward higher addresses. Once the data placement hits the end, it starts from the beginning again. Meanwhile, the packets are consumed from the head of the queue in a manner similar to how they were placed.

Figure 6-1 illustrates how the ring buffer operates when combined with the bus-mastering operation.

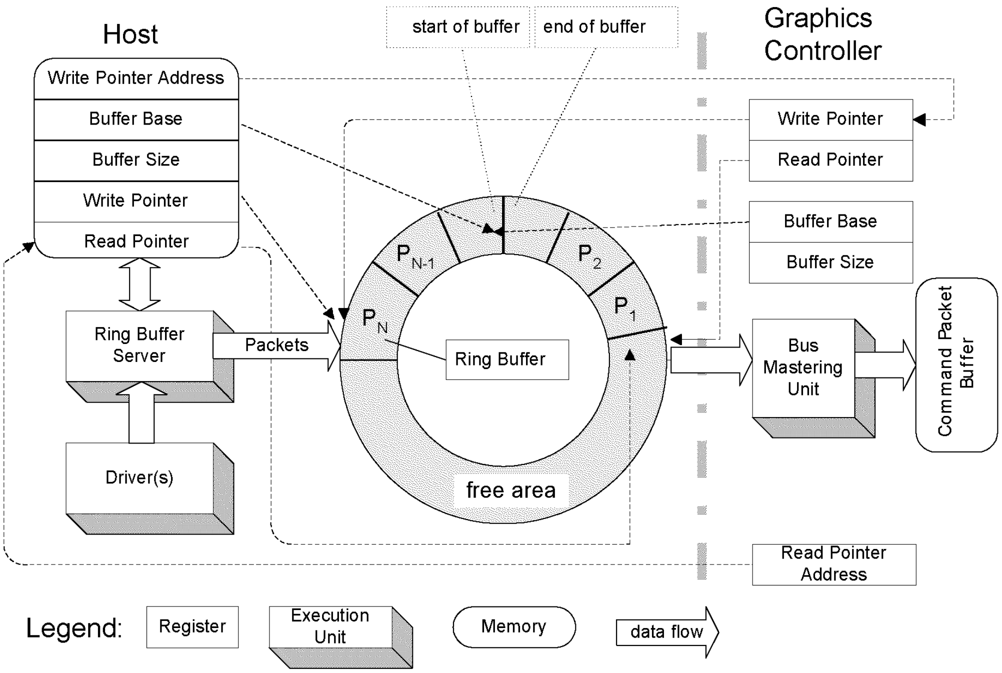


Figure 6-1: Ring Buffer and its Control Structure

In the figure, packets are placed into the buffer in a counter-clockwise order, forming a *packet queue*. The first packet in the queue is denoted by P_1, and the last by P_n. The start of the queue, P_1, is pointed to by the Read Pointer(s). The memory portion that is not occupied by packets is called the *free area*, and it is pointed to by the Write Pointer(s).

Initially, both the read and write pointers may point to the same location of the ring buffer, e.g. the start of the memory block. The two pointers pointing to the same location of the ring buffer generally implies one of two situations. One is that the buffer is empty, and the other is that the buffer is full. We want to define this situation as an empty buffer. To resolve the ambiguity of both pointers being equal, we must prevent the case of a full buffer from ever happening. It is the Host's responsibility to ensure that there is at least one free location in the buffer.

On the host side, the driver places command packets into the free area of the ring buffer, and informs the Graphics Controller of any changes to the Write Pointer by writing directly to the Write Pointer register inside the Graphics Controller. The host tracks free-space in the buffer by comparing its Read and Write Pointers, and suspends writing if the buffer becomes (almost) full.

On the Graphics Controller side, packets are taken away one-by-one from the head of the packet queue, pointed to by its Read Pointer, through the Host Bus Interface, and placed into the Command Packet Buffer. As the Graphics Controller updates its copy of the Read Pointer, it uses a bus-mastering write to update the Host's copy of the Read Pointer, residing in a shared memory location. The Graphics Controller has a register that holds the memory address of where the Host's Read Pointer resides, and uses that for the address of the bus-mastering write. The Graphics Controller tracks free-space in the buffer by comparing its Read and Write Pointers, and suspends reading if the buffer becomes empty (i.e., Read Pointer == Write Pointer).

To reduce traffic on the system memory bus, the Graphics Controller should not update the Host's copy of the Read Pointer every time it changes on the Graphics Controller side. To facilitate this, we have adopted a concept of a *block* of DWORDs in the packet queue. The Graphics Controller will update the host's copy of the Read Pointer every time it has consumed a "block's-worth" of data from the ring buffer. The other time when the Graphics Controller will update the Read Pointer is when it thinks that the packet queue is empty. The size of the block is programmable, to allow the programmer to trade-off the amount of time the system bus spends doing real data transfer vs the amount of time it spends on the communication overhead of updating read/write pointers. Larger block sizes tend to reduce communication overhead, at the "expense" of reducing the number of blocks in the queue, which reduces the amount of "slip" (or de-coupling) between the Host and the Graphics

Controller.

To reduce traffic on the system memory bus, the driver may want to minimize the frequency of accesses to its copies of the Read and Write Pointers. To minimize reads of the Read Pointer, it can check them once, calculate an amount of free space, and then decrement a local copy of the amount of free space as it adds packets to the queue. When it sees that the free-space is small (queue nearly full), it can start this procedure over again. (Its copy of the Read Pointer may have changed since the last time he read it.) The host also has the option of updating the Graphics Controller's Write Pointer on a less-frequent basis than with every write he does to the packet queue, possibly on a block-basis similar to the Graphics Controller's mechanism. However, if the buffer is running close to empty, any delay in updating the Graphics Controller's Write Pointer may add latency to the Graphics Controller's response to this command packet. Also, the host must be careful to update the Graphics Controller's copy of the Write Pointer if it wants the Graphics Controller to read from the queue until it is empty.

When the queue has become (almost) full, the host will have to poll the Read Pointer until space becomes available. In certain systems (Pentium II for example), this polling will stay within the processor cache, thus avoiding traffic on the system bus, and the snoop logic of the host CPU will take care of maintaining consistency between the main memory and the processor cache when the Graphics Controller performs its bus-mastering write of the Read Pointer. It is important to note that the Read Pointer must reside in PCI space in order for this snoop technique to work. AGP writes are not snooped.

6.3 Indirect Buffer Management

The Command Processor has the capability to read commands from other locations in memory, outside of the Ring Buffer. These locations are known as Indirect Buffer1 and Indirect Buffer2. This is accomplished as follows: there is a packet in the Primary command stream (being read from the ring buffer) which sets up the Indirect Buffer1 Address and Size registers of the Command Processor. The writing of the Indirect Buffer1 Size register triggers the Command Processor to begin fetching the new stream from the provided address. The last packet to be parsed from the Primary stream is the one that sets the Indirect Buffer1 Address and Size registers. The CP then begins fetching data from Indirect Buffer1. The data stream in Indirect Buffer1 may set up the Indirect Buffer2 Address and Size registers of the Command Processor. As before, writing of the Indirect Buffer1 Size register triggers the Command Processor to begin fetching the new stream from the provided address. The last packet to be parsed from the Indirect Buffer1 stream is the one that sets the Indirect Buffer2 Address and Size registers. The CP fetches the correct amount of data from Indirect Buffer2 until The Buffer2 Size is exhausted; it then returns to its interpretation of packets from Indirect Buffer1. The CP fetches the correct amount of data from Indirect Buffer1 until the Buffer1 Size is exhausted; it then returns to its interpretation of packets from the Primary Stream (being read from the ring buffer).

Indirect State Data buffers are only supported for non-real-time command streams.

6.4 Data Coherency Management

6.4.1 Ring Buffer Fetching Coherency

The Rage128 product revealed a weakness in some motherboard chipsets in that there is no mechanism to guarantee that data written by the CPU to memory is actually in a readable state before the Graphics Controller receives an update to its copy of the Write Pointer. In an effort to alleviate this problem, we've introduced a mechanism into the Graphics Controller that will delay the actual write to the Write Pointer for some programmable amount of time, in order to give the chipset time to flush its internal write buffers to memory.

There are two register fields that control this mechanism: PRE_WRITE_TIMER and PRE_WRITE_LIMIT. A timer-counter is associated with the PRE_WRITE_TIMER and a limit-counter is associated with the PRE_WRITE_LIMIT. There is also a staging register placed "in front of" the actual Write Pointer register of the CP.

All host CP_RB_WPTR writes go into the staging register and are held there until one of two events occurs:

(1) The timer-counter of PRE_WRITE_TIMER has expired; or

(2) The host has written the staging register PRE_WRITE_LIMIT times – decrementing the limit-counter to zero

When one of these conditions is satisfied the staging register contents is written to the ring's write pointer – initiating fetch requests.

The timer-counter is seeded with PRE_WRITE_TIMER every time the host writes to the Write Pointer register address (CP_RB_WPTR). The limit-counter is seeded with the PRE_WRITE_LIMIT every time the ring's write pointer is updated or whenever the PRE_WRITE_LIMIT is written by the host.

This implementation does not **guarantee** a certain time-delay between the host write to the Write Pointer, and the Graphics Controller read of the system memory; because the host could flood the Graphics Controller with multiple writes (more than the PRE_WRITE_LIMIT) in a short amount of time, thus overriding the time-delay imposed by the PRE_WRITE_TIMER.

Additional Notes:

1. Programming the PRE_WRITE_TIMER and PRE_WRITE_LIMIT to zero disables the delay function.
2. It is valid to set the PRE_WRITE_LIMIT to zero and the PRE_WRITE_TIMER >0. This will result in a fixed delay without a write limit.
3. Setting the PRE_WRITE_LIMIT > 0 and the PRE_WRITE_TIMER to zero is invalid as this setting may never update the ring's write pointer. The logic detects this case and will result in no delay being applied to the ring's write pointer update.

6.4.2 *Micro Engine Write Data Coherency (Updated: 07-20-2002)*

The CP writes data to external memory, which is later read by the graphics pipeline. The CP controls when the consumers of this information can issue read requests in order to make sure that the data is written to memory before being consumed. The CP controls the consumers of the data by using the write confirmation signal it receives from the Memory Hub.

The following data applies:

1. 2D Brush Data – The Micro Engine issues write confirmation on the last DWORD write and issues a SC_WAIT_WC event initiator to the Scan Converter (SC). The ME does not wait. Instead the event initiator written to the SC is used to throttle the SC's processing of the next packet until the write confirmation occurs – the assertion of the CP_SC_wc_inc signal.
2. 2D Palette Data – Same as the 2D Brush.
3. 2D Immediate Data – Same as the 2D Brush.
4. Constants – The Micro Engine issues the write confirmation on the last DWORD write. The ME does not wait. The Pre-Fetch Parser will stall a Load_Constant_Context packet if its write confirm counter is not zero. The PFP's counter is decremented by the ME when it receives confirmation from the Memory Hub.

6.4.3 VGT DMA Draw Initiator Deadlock Avoidance (Update: 02-05-2002)

The CP's Pre-Fetch Parser (PFP) issues index DMA requests to the VGT by writing the VGT_DMA_BASE and VGT_DMA_SIZE registers for the DRAW_INDX packets. Because we desire to hide the fetch latencies of the index DMA operations, these DMA requests are issued long before the corresponding DRAW_INITIATOR is sent to the VGT. Both the DMA request and the DRAW_INITIATOR enter the VGT through the same interface. So, care must be taken so as not to write so many DMA requests that the interface into the VGT is backed-up. This would block the ability for the CP to write the DRAW_INITIATOR. If this happens then the chip is deadlocked.

To prevent this situation, the RBBM will increment a counter every time it writes the VGT_DMA_SIZE register. The counter is decremented when the RBBM writes to the VGT_DRAW_INITIATOR register with the SOURCE_SELECT field set to "VGT DMA Data" is sent to the Global Register Bus.

The RBBM will stop issuing index DMA requests to the VGT when the count value is greater than a programmable threshold.

6.5 Command Stream Synchronization

6.5.1 WAIT_UNTIL

In the RBBM, there is an event engine that can be used to synchronize the sending of transactions to the Register Backbone based on status signals from its clients. The CP also has a mechanism that can directly provide the Host with knowledge of command status. This mechanism is the eight "SCRATCH" registers and their associated functionality.

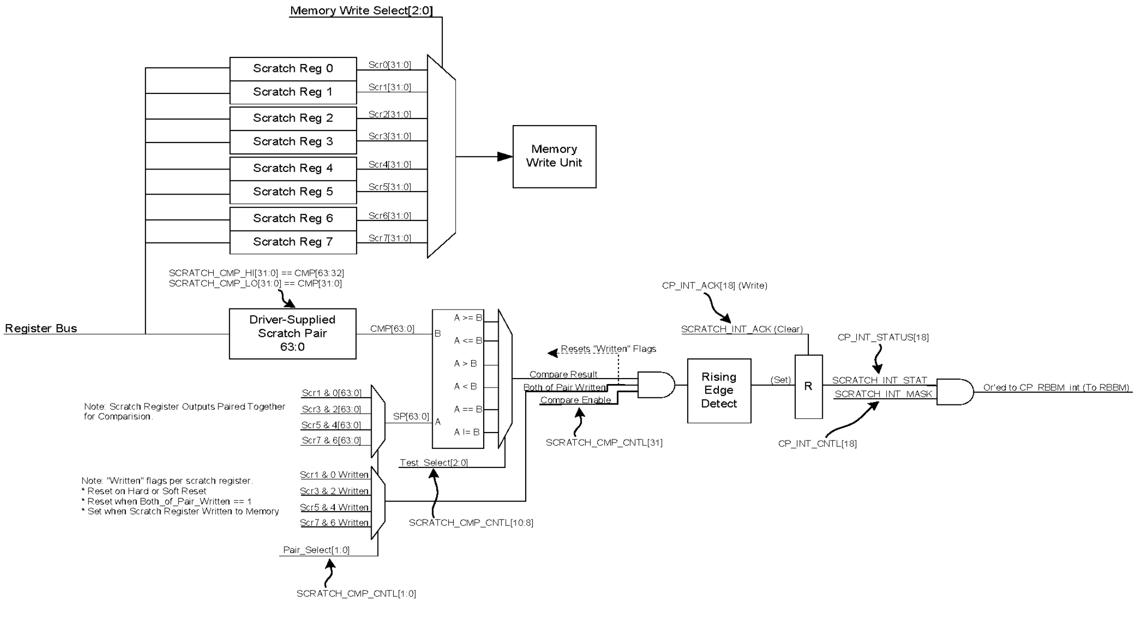
6.5.2 SCRATCH REGISTERS

Associated with the eight "SCRATCH" registers in the CP is a scratch address register and a write mask. When a scratch register is written, the CP will subsequently write its value to a location equal to what is programmed in the SCRATCH_ADDR register plus the number (0 to 7) of the scratch register. The writing of the scratch register's value by the CP is qualified by the register's write mask (SCRATCH_UMSK).

So, at the end of processing an Indirect Buffer, for example, a Type-0 packet can be inserted that writes a data pattern to SCRATCH_REG1. The driver software can poll the external location SCRATCH_ADDR+1 and when it changes to the value that was inserted in the Type-0 packet, the Driver will "know" that the CP has completed parsing the indirect buffer up to that point. Note that this status only indicates that the CP has fetched the data to that point. The data still may be being used by the rest of the pipeline.

An interrupt is also associated with the scratch registers, which is asserted when the scratch register pair selected is written to memory and is greater than or equal to the pair of values written by the Driver. The following diagram illustrates the generation of this interrupt.

Scratch Register Compare Interrupt



Note: Scratch Register Outputs Paired Together for Comparision.

Note: "Written" flags per scratch register.
* Reset on Hard or Soft Reset
* Reset when Both_of_Pair_Written == 1
* Set when Scratch Register Written to Memory

Updated: 3/26/2003
by John A. Carey

6.5.3 Vertex Shader Done Event (Updated: 04-12-2002)

The Vertex Shader Done (VSD) event allows the Driver to have a pipelined timestamp indicating the usage completion of a vertex shader by the Sequencer.

The Driver sets-up the VSD event by sending the CP the EVENT_WRITE PM4 packet. When the CP processes this packet it initializes the write-back with the address and data supplied in the packet and sends an event initiator down the pipeline to tell the Sequencer to report when it is done with the vertex shader.

When the Sequencer is done, it sends a “vertex shader done” event back to the CP which causes the write-back of the data to the external memory address supplied in the EVENT_WRITE PM4 packet.

6.5.4 Pixel Shader Done (Updated: 04-12-2002)

The Pixel Shader Done (PSD) event allows the Driver to have a pipelined timestamp indicating the usage completion of a pixel shader by the Sequencer.

The Driver sets-up the PSD event by sending the CP the EVENT_WRITE PM4 packet. When the CP processes this packet it initializes the write-back with the address and data supplied in the packet and sends an event initiator down the pipeline to tell the Sequencer to report when it is done with the pixel shader.

When the Sequencer is done, it sends a “pixel shader done” event back to the CP which causes the write-back of the data to the external memory address supplied in the EVENT_WRITE PM4 packet.

6.5.5 Cache Flush Done (Updated: 04-12-2002)

The Cache Flush Done (CFD) event allows the Driver to have a pipelined timestamp indicating that the Renderer Common (RC) has completed flushing its cache.

The Driver sets-up the CFD event by sending the CP the EVENT_WRITE PM4 packet. When the CP processes this packet it initializes the write-back with the address and data supplied in the packet and sends an event initiator down the pipeline to tell the RC to report when it has completed flushing its cache.

When the RC is done, it asserts the RC_CP_cache_flush signal to the CP, which causes the write-back of the data to the external memory address supplied in the EVENT_WRITE PM4 packet.

6.5.6 Direct Semaphore Writes (Updated: 04-12-2002)

For Crayola, a direct semaphore write command packet is included – Mem_Write. This packet contains an external memory address and data. When the CP processes the packet it writes data to the specified address. See the Crayola PM4 Specification for details of this command packet.

6.5.7 Interrupt Generation (Updated: 05-10-2002)

The CP has the ability to generate interrupts from the command streams. Three registers – CP_INT_CNTL, CP_INT_STATUS, and CP_INT_ACK – control operation of the interrupts within the CP. All the CP interrupts are logically OR'd to generate a single interrupt to the RBBM.

The CP_INTERRUPT PM4 packet provides the ability to generate an interrupt. The Micro Engine (ME) sets the appropriate interrupt status flag based on the stream (Primary, IB1, IB2, RT*) stream that contained the CP_INTERRUPT packet.

The CP's DMA engine generates an interrupt, which is reported in the CP_INT_* registers.

6.5.8 Non-Queued WAIT Until

The CP PM4 specification has the "WAIT_GUI_IDLE" command packet. This writes to the NQWAIT_UNTIL register in the RBBM that stalls subsequent writes from the CP until the GUI is idle. Note that the stall point is before the Command FIFO and Event Engine in the RBBM. The stall affects both "queued" and "non-queued" data.

6.5.9 DMA Synchronization with Command Stream

The CP has a DMA engine that can be synchronized with the non-real-time command stream. A CP_SYNC bit can be set when a DMA is initiated which performs this synchronization.

6.5.10 2D ←→ 3D Synchronization (Updated: 05-08-2002)

The CP implements auto-synchronization control between 2D and 3D switching. A 2D packet can be held-up until the 3D is idle or a 3D packet can be held-off until the 2D is complete.

Per Andy Gruber: I'd like to see this controlled by 2 Bits - 1 for 2D > 3D and 1 for 3D < 2D. It would be nice if these bits could be changed 'dynamically' by a type 0 packet such that a 2D driver than needed to do a scaled blt (3D operation) could do:

Normal 2D PM4 packet

Packet to turn off synchronization

3D Packet for Scale Blt

Normal 2D PM4 packets

Packet to restore synchronization

This would allow the driver to intermix 2D and 3D packets without suffering the synchronization penalty (but only under the control of a sophisticated driver).

The synchronization occurs in the Micro Engine. The RBBM also has an implicit synchronization control register for WAIT_IDLEGUI and CPSCRATCH_IDLEGUI as in previous chips.

6.6 State Management

6.6.1 *Overview (Updated: 04-15-2002)*

There is one context in the CRAYOLA which is applicable to both Renderstate and Constants. The CP assigns this context in a rotating manner (i.e. 0,1,2,3,4,5,6,7,0,1,…). The instruction code (Vertex and Pixel Shaders) is associated with a context, but the Instruction Memory (IM) pointers are managed by the CP. The IM is large enough to store shader programs for multiple contexts.

The context of the state is reported to the rest of the graphics pipe via the DRAW_INITIATOR. The DRAW_INITIATOR is sent for every primitive is in the context registers. By the CP writing the DRAW_INITIATOR to the assigned context, it tells the rest of the chip the assigned context.

Real-Time is always assigned state context zero. Real-Time streams are disabled as a default. The PM4 packet ME_INIT can be used to enable/disable real-time streams. A read-only status bit RTS_ENABLE in the CP_STATE_CNTL register can be read to determine if the CP has real-time streams enabled. When enabled, RTS uses context 0, which is then unavailable for non-Real-Time stream processing. In this case, the CP will assign the following contexts for non-real-time: 1,2,3,4,5,6,7,1,…

6.6.2 *Type-3 PM4 Packet State Management (Updated: 04-14-2003)*

For state management via Type-3 PM4 packets, the CP has the following packets available:

1. Set_State – Sub-Block data, Vertex Shader, and Pixel Shader Updates.
2. Set_Constant – Constant Update (ALU, Texture, Loop, Boolean, and Reg)
3. Load_Constant_Context – CP fetches previously stored constants into chip.
4. Invalidate_State – Invalidates Constant Pointers, Shader Code Pointers, and one or more Sub-Block Pointers for a given sub-block.

6.6.3 *Type-3 Fetched State (Updated: 01-30-2002)*

The CP fetches the following state data:

1. Shader instruction code (Vertex and Pixel) – The CP maintains the relative pointers to the Instruction Memory.
2. Constants – Embedded in the Command Stream (Set_Constant) or in a separate buffer (Load_Constant_Context).
3. Renderstate that is contained in sub-blocks.

The CP instructs the VGT to fetch the indices associated with the DRAW_INDX packet and supports the legacy 3D_DRAW_INDX_2 packet.

6.6.4 *State Buffer Synchronization (Updated: 01-30-2002)*

The CP can notify the Driver of the consumption of the sub-block buffers by the "store DWORD" mechanism. This is not an automatic process however. The Driver needs to place a MEM_WRITE PM4 packet after the SET_STATE packet to generate the semaphore DWORD write for this synchronization.

6.6.5 State Context Switching (Updated: 07-18-2002)

The CP manages the context switching for the graphics pipeline. It assigns the context and stalls updates to the graphics pipe if a context is not available. The CP supports both "incremental" and "pointer-based" updates of the state data. "Incremental" is just a fancy term for the use of Type-0 packets. "Pointer-Based" is another fancy term to indicate that state data is updated by one of the following packets: Set_State, Set_Constant, Load_Constant_Context, and Im_Load. Note that Type-1 packets cannot address the GFX decode spaces, so they cannot be used to update context-based Renderstate.

When the CP detects the need for a context switch, it does the following:

1. Writes the *VGT_EVENT_INITIATOR* register with the "Context_Done" ID set. This is used at the end of the pipe for reporting that the context is completed (i.e Causes RC to generate the RC_CP_Context_Done signal).
2. Assigns the next available context and waits until it is available. For instruction updates, the CP also waits for enough Instruction Memory to be available to fit the context's instruction code update (Vertex and Pixel Shaders).
3. The CP tells the graphics pipe to copy the prior GFX state set to the assigned GFX state set. The CP does this by writing to the *GFX_COPY_STATE* register at the assigned context. The lower 3-bits of the GFX_COPY_STATE value is the context that the state should be copied from.
4. The CP then writes the new data to the assigned state set by processing the "Incremental" and/or "Pointer-Based" packets.
5. After the first "state update" packet is processed, the STATE_CONTEXT_DIRTY flag is set so that consecutive "state update" packets do not also generate new contexts. Note that the STATE_CONTEXT_DIRTY flag is visible in the CP_STATE_CNTL register.

The need for a context switch is determined by the occurrence of the first "incremental" packet to a GFX decode space (Driver writes to context #0 by the defined protocol) after a "Draw" packet or the first "Pointer Based" packet, which does not have a pointer match for prior state, following a "Draw" packet.

Note that when the CP powers-up, the context is defaulted to '0' and the STATE_CONTEXT_DIRTY flag is set. Real-Time processing is also disabled. If real-time is subsequently enabled, the CP will issue a GFX_COPY_STATE write from 0 to 1 for the non-real-time processing before submitting any non-real-time writes.

Then, when the first "Draw" packet is processed, the CP will clear the STATE_CONTEXT_DIRTY flag so that a new context will be assigned on the next "state update" packet.

The CP recognizes the following packets as "state update" packets:

1. Type-0 to the GFX decode space. CP will write the Type-0 packet's data to assigned GFX context registers.
2. Set_State with a Pointer Mismatch.
3. Set_Constant.
4. Load_Constant_Context with a Pointer Mismatch.
5. Im_Load with a Pointer Mismatch.
6. Im_Load_Immediate.
7. Set_Scissors.
8. Load_Palette.

It is assumed that Type-0 incremental updates to the GFX decode spaces will not span outside context #0 of the GFX decode space. They will not start outside and end in the GFX decode space nor will they start inside and end outside the GFX decode space.

The CP recognizes the following packets as "Draw" packets:

1. Draw_Indx – Multiple can be sent per context.
2. 3d_Draw_Indx_2 – Multiple can be sent per context.
3. All 2D Packets Except:
    a. Ply_NextScan
    b. Set_Scissors
    c. Load_Palette
    d. 2D_Endian_Mode
4. Mpeg_Index
5. Viz Query

All other packets do not assign a new context when processed.

6.6.6 Constant Read/Write Coherency (Updated: 10-09-2002)

The CP issues read requests for constant data (LOAD_CONSTANT_CONTEXT) at the Pre-Fetch Parser (PFP) and issues external memory writes of constant data at the Micro Engine (ME). These read and write operations however can be consecutive in the command stream.

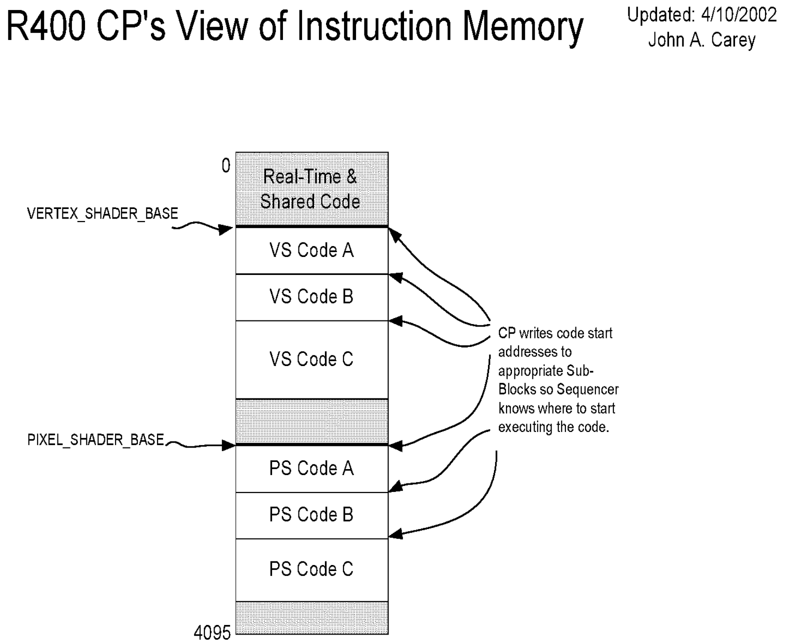
The CP must ensure that a read operation at the PFP gets all the constants that are written from the ME. To do this the PFP implements a "constant write counter" that increments when it processes a Set_Constant packet. When the ME processes the Set_Constant packet, it writes the last DWORD for the constant with a "write confirmation". When the write has completed, as indicated from the de-assertion of the MH_CP_writeclean signal, the MIU sends a pulse to the ME, which then generates a pulse to the PFP to decrement the "constant write counter". The PFP stalls the processing of the LOAD_CONSTANT_CONTEXT packet until the "constant write counter" is zero.

6.6.7 *Instruction Memory Management (Updated: 05-10-2002)*

The CP manages the loading of the shader code Instruction Memory (IM).

From the CP's perspective, the Instruction Memory (IM) is a single memory, which is divided into sections – one for Vertex shader code, one for Pixel shader code, and a third for Real-Time & Shared shader code. Programmable thresholds are maintained for dividing this memory. The Micro Engine uses the Vertex and Pixel shader base addresses in determining where to write the shader instructions for the Set_State, Im_Load, and Im_Load_Immediate packets. The ME_INIT packet is used to program these partition values. See the PM4 Specification for details.

Note that free space in the Real-Time section of the IM can also be used for subroutines (i.e. Shared Code) if needed. Figure 6-2 shows the CP's view of the instruction memory.



R400 CP's View of Instruction Memory

Updated: 4/10/2002
John A. Carey

CP writes code start addresses to appropriate Sub-Blocks so Sequencer knows where to start executing the code.

Figure 6-2: CP's View of Instruction Memory.

6.6.8 *Hardware Resources for State Management (Updated: 04-14-2003)*

The State Management (SM) functionality is mostly implemented in the Micro Engine (ME) via specific command packets. See the State Management Packets section of the CRAYOLA PM4 Specification for details.

Hardware Resources for the Micro Engine:

1. Eight Sub-Block Registers store the Sub-Block State Pointers and sizes. These are located at the Pre-Fetch Parser (PFP) Logic. Each Sub-Block Pointer registers stores the last address pointer used in the context determination scheme.
2. Vertex Shader Base and Size storage – Used by the Pre-Fetch Parser to determine whether to fetch the instruction code in the Set_State packet.
3. Pixel Shader Base and Size storage -- Used by the Pre-Fetch Parser to determine whether to fetch the instruction code in the Set_State packet.
4. ALU Constant Base Address – Used by Pre-Fetch Parser to determine whether to fetch the ALU constant data associated with a Load_Constant_Context packet.
5. Texture Fetch Constant Base Address – Same as ALU, but for Texture Constants.
6. Loop Constant Base Address – Same as ALU, but for Loop Constants.
7. Boolean Constant Base Address – Same as ALU, but for Boolean Constants.
8. Register Base Address – Same as ALU, but for register updates using the Set_Constant and LCC packets.
9. Offset Values for Each of the Sub-Blocks within the GFX decode space – Driver programs these with the ME_INIT packet during initialization.
10. 2D Indirect Buffer Base and Size – The Driver programs these values during initialization. The CP automatically fetches indirect buffer for 2D state on switch from 3D to 2D.
11. Instruction Memory Partioning Pointers – ME uses pointers in managing the instruction memory ring(s).

6.7 "Viz Query A" Support (Updated: 11-05-2001)

"Viz Query A" (henceforth just "Viz Query") support allows a number of "DRAW_INDX" packets to be omitted from the rendering process (by CP discard) based on the result of a prior (set of) draw packet's total invisibility.

The CP receives "DISARD" flags from the Scan Converter (SC), which indicate the visibility of the indices in the draw packet.

The driver issues "DRAW_INDX" packets with the VIZQ_USE flag set, which specifies that a visibility test is to be performed. The index identifying which query to use is also provided. The CP then discards the entire packet if the DISCARD flag is set.

The CP has an interface from the Scan Converter to update the DISCARD flags. The DISCARD flags are stored internal to the CP in registers. The Driver can use Type-0 packets to clear/invalidate the flag results.

See the VIZ_QUERY packet definition in the CRAYOLA PM4 Specification for more details on the protocols associated with Viz Query.

6.8 Predicated Packet Support (Updated: 08-26-2003)

Predicated packet support allows Type-3 packets to be discarded from the command stream that fail the bin compare test. All Type-3 packets with bit 0 set in their header will be tested against the Bin registers (if bit 0 is cleared the packet will proceed without the test). If the test passes, the packet will proceed through the CP for processing, otherwise it will be discarded if the test fails.

The test is setup by the drivers via 2 PM4 packets. There are BIN_MASK_HI & BIN_MASK_LO registers that represents the current driver defined bin category and are set by the SET_BIN_MASK packet. There are also a BIN_SELECT_HI & BIN_SELECT_LO registers that represents the bin category of the subsequent command stream data and are updated by the SET_BIN_SELECT packet. Once these registers are set, the CP can then compares the Bin registers for subsequent predicated Type-3 packets.

See the PM4 Command Specification for further details.

6.9 Multi-Pass Shader Support (Updated: 02-06-2002)

The CP supports the Multi-pass shader support requirement by being able to execute a single Indirect Buffer multiple times. The Type-3 INDIRECT_BUFFER packet has a MULTIPASS flag in its definition. When set, the CP will stop pre-fetching consecutive indirect buffers until it receives confirmation from the Scan Converter that it should either loop or continue. If a loop is required, the Pre-Fetch Parser (PFP) in the CP will re-fetch the same Indirect Buffer. If the CP is told to continue, the PFP will just go to the next command packet in the command stream.

There is no restriction on the number of times an Indirect Buffer can be re-fetched.

6.10 Error Checking / Command Buffer Validation (Updated: 05-29-2002)

The goal for CRAYOLA is to have the CP perform some error checking/validation of the fetched command streams in order to recognize erroneous data. This erroneous data may be from Driver bugs, corruption due to bugs on the system bus or memory, or even hostile software attacking the command buffers.

All of the error checking is implemented in the Micro Engine (ME). Since the error checking degrades performance, it is intended only for debug. To enable these features, the microcode needs to be recompiled with UCODE_DEBUG defined.

<u>Error Checking / Validation Options:</u>

1. Checksum logic – Computes checksum on the in-coming command stream data and compares to known checksum. Performing checksums on the command streams may cause the performance to degrade. <u>This option is therefore not implemented in the CRAYOLA Command Processor</u>.

2. Reserved/Unused Bit Checking – The Micro Engine looks at "reserved" or "unused" bits in the command packet headers to determine if the header is valid. Note that the Driver has to ensure that the "reserved" bits are always set to '0'. This level of protection is enabled by the ME_INIT packet. The RESERVED_BIT_ERROR interrupt is asserted if an error occurs.

    a. Type-0 Packet Header – No "Reserved" bits exist to check.

    b. Type-2 Packet Header – Reserved bits are 29:0 – Not Checked. Drivers Use the Reserved Bit Fields.

    c. Type-3 Packet Header – Reserved bits are 7:0

3. Checking for Valid Opcodes in the Command Packet Headers -- CP has a valid Opcode table which is part of the microcode image. The test is performed in the Micro Engine (ME). This level of protection is always enabled in the CRAYOLA CP. The OPCODE_ERROR interrupt is asserted if an error occurs.

4. Putting Special fixed packets at the beginning of valid Command Buffers (Indirect and Real-Time) -- The CP checks for the IB_PREAMBLE packet before continuing on with the processing of the Indirect Buffers. This level of protection is enabled by setting the IB_START_CHECK_ENABLE bit in the ME_INIT packet. The IB_ERROR interrupt is asserted if an error occurs.

5. A bit could be set that limits the CP's writing to certain locations of the register map (i.e. Limiting access of the command packets to the first portion of the register map that re-programs the PLLs etc.). This level of protection is enabled by setting the PROTECTED_MODE_ENABLE bit in the ME_INIT packet. The PROTECTED_MODE_ERROR interrupt is asserted if an error occurs.

<u>Error Recovery Options</u>:

1. Stop and Notify Driver – Upon recognition of the error, the CP notifies the driver either via an interrupt and / or writing a semaphore value. The notification via an interrupt is what is implemented in the CRAYOLA CP.
2. Attempt to Recover – If an error is detected in an Indirect or Real-Time Stream Buffer, the CP should skip over the entire buffer and return control back to the Primary Command stream. An error detected in the Primary Command Stream should just halt the CP. Either way; the CP should probably notify the Driver that an error has occurred. <u>The CRAYOLA CP does not implement any recovery algorithm</u>.

When the CP detects an error it raises an interrupt and stops. The CP's soft reset needs to be used to clear the error condition.

6.11 System Bug Fixes with the CP (Updated: 01-30-2002)

The CP has been used in prior projects to work-around hangs and other bugs that arise post-silicon. The following list highlights how the CP has been used in the past for this purpose:

1. Addition of Extra Data writes to Registers in Command Stream – Used to reset or force constants to registers. Constants are embedded in the microcode.
2. Re-Ordering of Command Stream Data writes.
3. Saving Data Writes for Future Use in Other Command Packets – Data typically saved to CP's micro-code RAM then used by other Type-3 command packets.
4. Swapping/Data Alignment within Specific DWORDs of Command Packets – This is somewhat restrictive in the current CP implementation.
5. Late Addition of Type-3 Command Packets.

It is planned that the CRAYOLA CP will provide these same capabilities. It is also possible to provide the following:

1. More-Generalized Data Swapping Capabilities
2. More-General ALU Operations (Add, Subtract, Multiply)
3. Addition of Logic Operations to CP Micro Engine For Data Masking/Setting/Negation
4. Generalize Data Shifting Operations (i.e. Single-Clock Data Shifts/Rotates: Left/Right)

6.12 Minimal Power-Up Initialization for the Command Processor (Updated: 10-03-2002)

The following sequence is the minimal power-up initialization sequence for the CP:

1) Program the CP_RB_CNTL register with the appropriate Ring Buffer settings.
2) Program the CP_RB_BASE register with the base address of the Ring Buffer.
3) Write the CP_RB_RPTR_ADDR register with the Ring Buffer reporting address if the CP is to write the ring buffer read pointer back to memory (i.e. CP_RB_CNTL.RB_NO_UPDATE = 0).
4) If interrupts are to be used, then set the CP_INT_CNTL to enable the appropriate interrupts and write 0xFFFFFFFF to the CP_INT_ACK to clear all interrupts pending.
5) If the scratch registers will be used for synchronization:
    - a) Program SCRATCH_ADDR with the external base address.
    - b) Program the SCRATCH_SWAP with the correct data swap function.
    - c) Set the appropriate bits in the SCRATCH_UMSK to enable write-back of data from the CP.
6) Load the CP's Non-Real-Time microcode:
    - a) Clear microcode RAM start address by writing zero to the CP_ME_RAM_WADDR register.
    - b) Write the microcode to the CP_ME_RAM_DATA port register.
7) If Real-Time will be enabled, Load the CP's Real-Time microcode.
    - a) Clear microcode RAM start address by writing zero to the CP_RT_ME_RAM_WADDR register.
    - b) Write the real-time microcode to the CP_RT_ME_RAM_DATA port register.
8) Initialize the 2D Default State:
    - a) Write the 2D Indirect Buffer Base Address [31:2] register.
    - b) Write the 2D Indirect Buffer Size [19:0] register.
    - c) Write the 2D default registers for the GUI Master Control handler (DEFAULT_PITCH_OFFSET, DEFAULT2_PITCH_OFFSET, etc.). See the "2D Control Registers" section of this document.
    - d) Set the Destination Rotate 2D Boolean (B11) and any of the B31-to-B19 Booleans in the CP_2D_BOOLEANS register if needed. See the "2D Control Registers" section of this document.
9) Clear the ME_HALT bit in the CP_ME_CNTL register to start the Non-Real-Time Micro Engine.
10) If Real-Time processing is desired, clear the ME_HALT_RT bit in the CP_ME_CNTL register.
11) Issue the ME_INIT PM4 packet in the Ring Buffer to program variables that the Micro Engine uses. See the PM4 specification for details.
    - a) The ME_INIT PM4 packet includes the 2D/3D implicit synchronization controls for 2D processing.
12) If Real-Time Processing is desired, issue an ME_INIT PM4 packet in a real-time stream. This will program variables that the Real-Time Micro Engine uses. See the PM4 specification for details.

If this is a recovery from a power-management shutdown and it is desired that the Ring Buffer read pointer be set to non-zero, the following steps need to also be performed before any ring buffer submits.

1) Set the RB_RPTR_WR_ENA bit in the CP_RB_CNTL register to enable writing of the RPTR.
2) Write the CP_RB_RPTR_WR register with the RPTR value.
3) Write CP_RB_WPTR, to make it match the RPTR, causing the ring buffer to appear to be empty.
4) Clear the RB_RPTR_WR_ENA bit if no further writes of the RPTR are desired.

6.13 Soft Resetting the Command Processor (Updated: 11-06-2002)

A certain sequence of actions is required of the in order to perform a “clean” soft reset of the CP:

1) Disarm all Real-Time Streams by clearing the “ARM” bit in each of the CP_RT*_COMMAND registers.
2) Write to the proper RBBM_SOFT_RESET register to assert and then de-assert the Soft Reset signal to the CP.
3) Clear the ME_HALT bit in the CP_ME_CNTL register to start the Non-Real-Time Micro Engine.
4) Clear the ME_HALT_RT bit in the CP_ME_CNTL register to start the Real-Time Micro Engine.
5) Re-arm any Real-Time Streams by setting the “ARM” bit in their CP_RT*_COMMAND register.

Note: The duration of the soft reset signal must be long enough to ensure that all data from the Memory Hub has been returned to the CP. Also it would be wise to soft reset the Memory Hub as well so as to make sure the handshaking signals on both units are in sync.

6.14 Controlling Register Transactions Queued vs. Non-Queued (Updated: 05-08-2002)

The CP can control whether register writes are queued or non-queued through the RBBM. This applies to all register transactions except for PFP writes and Real-Time register transactions. The PFP writes and Real-time register transactions have their own dedicated paths through the RBBM.

Here is a summary of the control features:

1. PM4 Packets: The queued vs. non-queued setting is in the ME_INIT PM4 packet. This packet is sent to the CP after the microcode is loaded and the ME_HALT bit is cleared.
2. DMA Engine: QUEUED flag (bit 21) is available in the DMA Descriptor.
3. Real-Time Event Engine: Bit 15 in the RT*_COMMAND register is the "QUEUED_REG_PATH" control. This bit controls whether the register read operations (for polling) are queued or non-queued.

7. Register Descriptions

The Command Processor has a server interface from the RBBM (Register Backbone Manager) of the chip. Through this interface, any client to the RBBM can program control registers inside the CP.

7.1 Ring Buffer Control Register (Updated: 09-25-2002)

CP_RB_CNTL
Ring Buffer Control [CSF]

Field Name	Bit(s)	Description
RB_RPTR_WR_ENA	31	Enables the transfer of the Writable Read Pointer to the active Read Pointer when the Write Pointer is written. Default = 0 (Disabled)
RB_NO_UPDATE	27	Ring Buffer No Write to Read Pointer 0=Write to Host's copy of Read Pointer in system memory. 1=Do not write to Host's copy of Read pointer. The purpose of this control bit is to have a fall-back position if the bus-mastered write to system memory doesn't work, in which case the driver will have to read the Graphics Controller's copy of the Read Pointer directly, with some performance penalty. Default = 0
Reserved	19:18	*Maximum Fetch Size use to occupy this bit location. For CRAYOLA the fetch size is set at 256-bits per request.*
BUF_SWAP	17:16	Endian Swap Control for Ring, Indirect, Real-Time, and State Data Buffer Fetches. Only affects the chip behavior if the buffer resides in system memory. 0 = No swap 1 = 16-bit swap: 0xAABBCCDD becomes 0xBBAADDCC 2 = 32-bit swap: 0xAABBCCDD becomes 0xDDCCBBAA 3 = Half-DWORD swap: 0xAABBCCDD becomes 0xCCDDAABB Default = 0
RB_BLKSZ	13:8	Ring Buffer Block Size (Granularity Threshold). This defines the number of quadwords that the Command Processor will read between updates to the Host's copy of the Read Pointer. This size is expressed in $\log_2$ of the actual size (in 64-bit quadwords). For example, for a block of 1024 quadwords, you would program this field to 10(decimal). A value of 0 specifies a 2 DWORD block size. A value of 1 specifies a 4 DWORD block size. A value of 2 specifies a 8 DWORD block size. …etc… A value of 19 specifies a (2^{20} – 1) DWORD block size. *Values greater than 19 specify a (2^{20} – 1) DWORD block size.* Default = 0
RB_BUFSZ	5:0	Ring Buffer Size. This size is expressed in $\log_2$ quadword size of the Ring Buffer. Valid values are 0 to 19. A value of 0 is clamped to 2 and specifies an 8 DWORD ring buffer. A value of 1 is clamped to 2 and specifies an 8 DWORD ring buffer. A value of 2 specifies an 8 DWORD ring buffer. …etc… A value of 19 specifies a 2^{20} DWORD ring buffer. *Values greater than 19 specify a 2^{20} DWORD ring buffer.* Default = 0

7.2 Ring Buffer Base Register

CP_RB_BASE
Ring Buffer Base [CSF]

Field Name	Bit(s)	Description
RB_BASE	31:5	Ring Buffer Base – Double Octword Aligned. Address of the beginning of the ring buffer. Default = 0xABCDEFE0

7.3 Ring Buffer RPTR Report Address Register

CP_RB_RPTR_ADDR
Ring Buffer Read Pointer Address [CSF]

Field Name	Bit(s)	Description
RB_RPTR_ADDR	31:2	Ring Buffer Read Pointer Address. Address of the Host's copy of the Read Pointer. Default = 0.
RB_RPTR_SWAP	1:0	Swap control of the reported read pointer address. See CP_RB_CNTL.BUF_SWAP for the encoding. Default = 0.

7.4 Ring Buffer RPTR (For Polling) Register

CP_RB_RPTR (RO)
Ring Buffer Read Pointer [CSF]

Field Name	Bit(s)	Description
RB_RPTR	19:0	Ring Buffer Read Pointer. This is an index (in DWORDs) of the current element being read from the ring buffer. Default = 0. ***Note that this status resets to zero from a soft/hard reset of the CP.***

7.5 Writable Ring Buffer RPTR Register

CP_RB_RPTR_WR
Writable Ring Buffer Read Pointer [CSF]

Field Name	Bit(s)	Description
RB_RPTR_WR	19:0	Writable Ring Buffer Read Pointer. Writable for updating the RB_RPTR after an ACPI. Default = 0.

7.6 Ring Buffer Write Pointer Register

CP_RB_WPTR (Initiator)
Ring Buffer Write Pointer [CSF]

Field Name	Bit(s)	Description
RB_WPTR	19:0	Ring Buffer Write Pointer. This is an index (in DWORDs) of the last known element to be written to the ring buffer (by the Host). Default = 0. Writing to this register initiates the CP to fetch the ring buffer data. ***Note that this status resets to zero from a soft/hard reset of the CP.***

7.7 Ring Buffer Coherency Control

CP_RB_WPTR_DELAY
Ring Buffer Write Pointer Delay [CSF]

Field Name	Bit(s)	Description
PRE_WRITE_TIMER	27:0	Pre-Write Timer. The number of clocks that a write to the CP_RB_WPTR register will be delayed until actually taking effect. Default = 0
PRE_WRITE_LIMIT	31:28	Pre-Write Limit. The number of times that the CP_RB_WPTR register can be written (while the PRE_WRITE_TIMER has not expired) before the CP_RB_WPTR register is forced to be updated with the most recently written value. Default = 0

7.8 Indirect Buffer #1 Control Registers

CP_IB1_BASE (R/W)
Indirect Buffer1 Base [CSF]

Field Name	Bit(s)	Description
IB1_BASE	31:5	Indirect Buffer #1 Base. Address of the beginning of an indirect buffer initiated from the Ring Buffer.

CP_IB1_BUFSZ (R/W)
Indirect Buffer1 Size [CSF]

Field Name	Bit(s)	Description
IB1_BUFSZ	19:0	Indirect Buffer #1 Size. This size is expressed in DWORDs. This field is an initiator to begin fetching commands from the Indirect Buffer. Default = 0

7.9 Indirect Buffer #2 Control Registers

CP_IB2_BASE (R/W)
Indirect Buffer2 Base [CSF]

Field Name	Bit(s)	Description
IB2_BASE	31:5	Indirect Buffer #2 Base. Address of the beginning of an indirect buffer initiated from Indirect Buffer #1.

CP_IB2_BUFSZ (R/W)
Indirect Buffer2 Size [CSF]

Field Name	Bit(s)	Description
IB2_BUFSZ	19:0	Indirect Buffer #2 Size. This size is expressed in DWORDs. This field is an initiator to begin fetching commands from the Indirect Buffer. Default = 0

7.10 Real-Time Stream Fetcher Control Registers

CP_RT_BASE (RO)
Real-Time Buffer Base [RTEE]

Field Name	Bit(s)	Description
RT_BASE	31:5	Real-Time Command Buffer Base. Address of the beginning of the real-time state buffer.

CP_RT_BUFSZ (RO)
Real-Time Buffer Size [RTEE]

Field Name	Bit(s)	Description
RT_BUFSZ	19:0	Real-Time Command Buffer Size. This size is expressed in DWORDs.

CP_RT_ST_BASE (RO)
Real-Time State Buffer Base [CSF]

Field Name	Bit(s)	Description
RT_ST_BASE	31:2	Real-Time State Data Buffer Base. Address of the beginning of the real-time state buffer.

CP_RT_ST_BUFSZ (RO)
Real-Time State Buffer Size [CSF]

Field Name	Bit(s)	Description
RT_ST_BUFSZ	19:0	Real-Time State Data Buffer Size. This size is expressed in DWORDs.

7.11 State Queue Control Registers (Updated: 09-13-2002)

CP_ST_BASE (RO)
State Buffer Base [CSF]

Field Name	Bit(s)	Description
ST_BASE	31:2	State Data Buffer Base. Address of the beginning of the state buffer.

CP_ST_BUFSZ (RO)
State Buffer Size [CSF]

Field Name	Bit(s)	Description
ST_BUFSZ	19:0	State Data Buffer Size. This size is expressed in DWORDs.

7.12 Micro Engine Queue Control Registers (Updated: 01-07-2003)

CP_QUEUE_THRESHOLDS
Command Stream Queue Thresholds [CSF]

Field Name	Bit(s)	Description
STQ_RT_ST_START	30:24	Real-Time State Data Double Octword Start in the State Queue (Before PFP) Default=0x2B
STQ_RT_START	22:16	Real-Time Data Double Octword Start in State Stream Queue (Before PFP) Default=0x16
CSQ_IB2_START	13:8	Indirect Buffer #2 Data Double Octword Start in Command Stream Queue (Before PFP) Default=0x2B
CSQ_IB1_START	5:0	Indirect Buffer #1 Data Double Octword Start in Command Stream Queue (Before PFP) Default=0x16

CP_MEQ_THRESHOLDS
Micro Engine Queue Thresholds [PFP]

Field Name	Bit(s)	Description
ROQ_END	30:24	Double Octword End of Reorder Queues. Default=0x40 Note: Programmable only if bit set in CP_DEBUG.
MEQ_ END	22:16	Double Octword End of PFP-to-ME Command Queue. Default=0x40 Note: Programmable only if bit set in CP_DEBUG.

CP_CSQ_AVAIL (RO)
Command Stream Queue Available Status [CSF]

Field Name	Bit(s)	Description
CSQ_CNT_IB2	28:20	Count of available DWORDs in the CS queue for the Indirect #2 Stream. Read Only.
CSQ_CNT_IB1	18:10	Count of available DWORDs in the CS queue for the Indirect #1 Stream. Read Only.
CSQ_CNT_RING	8:0	Count of available DWORDs in the CS queue for the Primary Stream. Read Only.

CP_STQ_AVAIL (RO)
Command Stream Queue Available Status [CSF]

Field Name	Bit(s)	Description
STQ_CNT_RT_ST	28:20	Count of available DWORDs in the CS queue for the Real-Time State Stream. Read Only
STQ_CNT_RT	18:10	Count of available DWORDs in the CS queue for the Real-Time Stream. Read Only.
STQ_CNT_ST	8:0	Count of available DWORDs in the CS queue for the Non-Real-Time State. Read Only.

CP_MEQ_AVAIL (RO) Command Stream Micro Engine Queue Available Status [CSF]		
Field Name	**Bit(s)**	**Description**
MEQ_CNT	9:0	Count of available DWORDs in the ME queue. Read Only.

7.13 Micro Engine Control and Status (Updated: 12-02-2002)

The Micro Engine Control and Status registers are used to indicate the busy status of the Micro Engine, but they are mainly used for debugging the Micro Engine (ME).

CP_ME_CNTL (R/W)
Micro Engine Debug Control [ME]

Field Name	Bit(s)	Description
PROG_CNT_SIZE	31	Determines if CP_PROG_COUNTER is updated every 1 or 16 core clocks by setting field to 0 or 1 respectively. Default=0.
ME_BUSY_RT	30	Micro Engine is Busy Processing a Real-Time Stream. Read-Only.
ME_BUSY	29	Micro Engine is Busy Processing a Non-Real-Time Stream. Read-Only.
ME_HALT	28	Halt the Non-Real-Time Micro Engine. The non-RT ME halts at next packet boundary. Initialization needs to clear this bit after loading the microcode to start the CP. Default = 1.
ME_HALT_RT	27	Halt the Real-Time Micro Engine. The RT ME halts at next packet boundary. Initialization needs to clear this bit after loading the microcode to start the CP. Default = 1.
PIX_DEALLOC_FIFO_EMPTY	26	Pixel Shader De-Allocation FIFO is Empty. Read-Only.
VTX_DEALLOC_FIFO_EMPTY	25	Vertex Shader De-Allocation FIFO is Empty. Read Only.
Reserved	24:16	Reserved
ME_STATMUX	15:0	Selects internal registers on Micro Engine's internal test bus. Bit 15 = Micro Engine Select (0 => Non-RT, 1 => RT) Bits 14:0 = Select for Signal Data is observed by reading the CP_ME_STATUS register. • Micro Engine GPRs and Current State (RT and nRT) • Scratch Register Contents (RT and nRT) See table in the ME Implementation of the CP Spec.

CP_ME_STATUS (RO)
Micro Engine Debug Data [ME]

Field Name	Bit(s)	Description
ME_DEBUG_DATA	31:0	Debug Data from Micro Engine, which is selected from the ME_STATMUX field in the CP_ME_CNTL register. Read-Only.

7.14 Non-Real-Time Micro Engine Command Store/Cache Control Registers

CP_ME_RAM_WADDR (R/W)
Non-Real-Time Micro Engine Code Store Write Address [ME]

Field Name	Bit(s)	Description
ME_RAM_WADDR	10:0	Micro Engine RAM Write Address Writing this register puts the RAM access circuitry into "Write Mode". The write address auto-increments from writes to the data register.

CP_ME_RAM_RADDR (R/W)
Non-Real-Time Micro Engine RAM Read Address [ME]

Field Name	Bit(s)	Description
ME_RAM_RADDR	10:0	Micro Engine RAM Address Writing this register puts the RAM access circuitry into "Read Mode". The read address auto-increments from reads to CP_ME_RAM_DATA.

CP_ME_RAM_DATA
Non-Real-Time Micro Engine RAM Data Port [ME]

Field Name	Bit(s)	Description
ME_RAM_DATA	31:0	Micro Engine RAM Data Used to load the non-real-time micro code.

CP_ME_RDADDR (RO)
Non-Real-Time Micro Engine Read Address [ME]

Field Name	Bit(s)	Description
ME_RDADDR	31:0	Non-RT Micro Engine Writes to this address to Generate a Read Request. Bits 31:2: DWORD Address Bits 1:0: Swap Code

7.15 Real-Time Micro Engine Command Store/Cache Control Registers

CP_RT_ME_RAM_WADDR (R/W)
Real-Time Micro Engine Code Store Write Address [ME]

Field Name	Bit(s)	Description
RT_ME_RAM_WADDR	8:0	Real-Time Micro Engine RAM Write Address Writing this register puts the RAM access circuitry into “Write Mode”. The write address auto-increments from writes to the data register.

CP_RT_ME_RAM_RADDR (R/W)
Real-Time Micro Engine RAM Read Address [ME]

Field Name	Bit(s)	Description
RT_ME_RAM_RADDR	8:0	Real-Time Micro Engine RAM Address Writing this register puts the RAM access circuitry into “Read Mode”. The read address auto-increments from reads to CP_RT_ME_RAM_DATA.

CP_RT_ME_RAM_DATA
Real-Time Micro Engine RAM Data Port [ME]

Field Name	Bit(s)	Description
RT_ME_RAM_DATA	31:0	Real-Time Micro Engine RAM Data High Used to load the real-time micro code.

CP_RT_ME_RDADDR (RO)
Real-Time Micro Engine Read Address [ME]

Field Name	Bit(s)	Description
RT_ME_RDADDR	31:0	Real-Time Micro Engine Writes to this address to Generate a Read Request. Bits 31:2: DWORD Address Bits 1:0: Swap Code

7.16 PFP Microcode RAM Registers

CP_PFP_UCODE_ADDR (R/W)
Pre-fetch Parser RAM Address [PFP]

Field Name	Bit(s)	Description
UCODE_ADDR	8:0	The Address is used for both reads & writes into the PFP`s microcode RAM. See the 'Host Programming Considerations' for further details.

CP_PFP_UCODE_DATA (R/W)
Pre-fetch Parser RAM Data [PFP]

Field Name	Bit(s)	Description
UCODE_DATA	23:0	The Data is used for both reads & writes into the PFP`s microcode RAM. See the 'Host Programming Considerations' for further details.

CP_PFP_RT_UCODE_ADDR (R/W)
Real-Time Pre-fetch Parser RAM Address [PFP]

Field Name	Bit(s)	Description
RT_UCODE_ADDR	7:0	The Address is used for both reads & writes into the PFP`s real-time microcode RAM. See the 'Host Programming Considerations' for further details.

CP_PFP_RT_UCODE_DATA (R/W)
Real-Time Pre-fetch Parser RAM Data [PFP]

Field Name	Bit(s)	Description
RT_UCODE_DATA	23:0	The Data is used for both reads & writes into the PFP`s reak-time microcode RAM. See the 'Host Programming Considerations' for further details.

7.17 Miscellaneous Registers (Updated: 06-04-2003)

The CP_Debug register is in the design for "chicken" bits – as in "I am a bit 'chicken' that this will actually work, so I include a signal to bypass the function". ***Note that the "Reserved" bits are preserved (i.e. Not optimized) so that they can be used for ECOs if neeed.***

CP_DEBUG
CP Debug Register [RBIU]

Field Name	Bit(s)	Description
MIU_WRITE_PACK_DISABLE	31	Disable Packing of Write Transactions in the MIU. Default = 0 (Pack Enabled)
SIMPLE_ME_FLOW_CONTROL	30	If set, the micro-engine pipeline will stall if the data out register is stalled. No bubbles in the pipeline will be filled. Default = 0.
DMA_DSCR_OCTW_SEL	29	Swaps Descriptor Octwords. Default = 0.
PREFETCH_MATCH_DISABLE	28	Disables Matching of Prior Pointers for PM4 Packets. Default = 0 (Match Enabled).
DYNAMIC_CLK_DISABLE	27	If set the Clock Gating Circuitry in the CP will keep the clocks enabled. Default=0.
PREFECTH_PASS_NOPS	26	If set, the PFP will pass Type-2 and Type-3 NOP packets. Default=0.
MIU_128BIT_WRITE_ENABLE	25	If set, allows 128-bit writes, when applicable, to the MH. Default=0.
PROG_END_PTR_ENABLE	24	If set, writes to the ROQ & MEQ end pointers become programmable. These should be programmed for experimental use only and otherwise left at their default values.
PREDICATE_DISABLE	23	Disables predicated packet logic so that no packets are discarded.
CP_DEBUG_UNUSED	22:0	Reserved – ***Bits preserved for possible ECO usage.***

7.18 Performance Monitoring Registers (Updated: 10-29-2002)

The CP_PERFMON_CNTL is a global register that the CP instances. Other units also shadow this register. The CP is the only client that responds to reads.

CP_PERFMON_CNTL
Performance Monitor Control [RBIU]

Field Name	Bit(s)	Description
Reserved	31:10	Reserved
PERFMON_ENABLE_MODE	9:8	Control for matching perf_counter_flag 00: Always Count 01: Always Count 10: Count if `perf_counter_flag` is true 11: Count if `perf_counter_flag` is not true
Reserved	7:4	Reserved
PERFMON_STATE	3:0	Put the counters into some state: 0: Disable and Reset 1: Enables Counting 2: Freeze- Disables Counting

CP_PERFCOUNTER_SELECT
Performance Counter #1 Select [RBIU]

Field Name	Bit(s)	Description
Reserved	31:6	Reserved
PERFCOUNT_SEL	5:0	Select for Inputs to the CP's Performance Counter. See the Performance Monitoring section of this specification.

CP_PERFCOUNTER_LO (RO)
Performance Counter #1 Lower Bits [RBIU]

Field Name	Bit(s)	Description
PERFCOUNT_LO	31:0	Lower Bits of CP's Performance Counter.

CP_PERFCOUNTER_HI (RO)
Performance Counter Upper Bits [RBIU]

Field Name	Bit(s)	Description
Reserved	31:16	Reserved
PERFCOUNT_HI	15:0	Upper Bits of CP's Performance Counter.

CP_PROG_COUNTER
32-bit Programmable Counter [ME]

Field Name	Bit(s)	Description
COUNTER	31:0	Increments every 1 or 16 core clocks as programmed in CP_ME_CNTL[31]. Used with the PM4 MEM_WRITE & EVENT_WRITE packets and written to memory (See PM4 Spec. for more detail).

7.19 Synchronization & Interrupt Registers (Updated: 12-20-2001)

SCRATCH_REG0 **Scratch Register 0 [RBIU]**		
Field Name	**Bit(s)**	**Description**
SCRATCH_REG0	31:0	Scratch Pad Register.

SCRATCH_REG1 **Scratch Register 1 [RBIU]**		
Field Name	**Bit(s)**	**Description**
SCRATCH_REG1	31:0	Scratch Pad Register.

SCRATCH_REG2 **Scratch Register 2 [RBIU]**		
Field Name	**Bit(s)**	**Description**
SCRATCH_REG2	31:0	Scratch Pad Register.

SCRATCH_REG3 **Scratch Register 3 [RBIU]**		
Field Name	**Bit(s)**	**Description**
SCRATCH_REG3	31:0	Scratch Pad Register.

SCRATCH_REG4 **Scratch Register 4 [RBIU]**		
Field Name	**Bit(s)**	**Description**
SCRATCH_REG4	31:0	Scratch Pad Register.

SCRATCH_REG5 **Scratch Register 5 [RBIU]**		
Field Name	**Bit(s)**	**Description**
SCRATCH_REG5	31:0	Scratch Pad Register.

SCRATCH_REG6 **Scratch Register 6 [RBIU]**		
Field Name	**Bit(s)**	**Description**
SCRATCH_REG6	31:0	Scratch Pad Register.

SCRATCH_REG7
Scratch Register 7 [RBIU]

Field Name	Bit(s)	Description
SCRATCH_REG7	31:0	Scratch Pad Register.

SCRATCH_UMSK
Scratch Register Update Mask [RBIU]

Field Name	Bit(s)	Description
SCRATCH_SWAP	17:16	Endian Swap method to use when the CP writes the scratch pad registers out to memory. Also used for Pipeline Store DWORD writes. The MH/MC does the actual data swapping. 0 = No swap (Default) 1 = 16-bit swap: 0xAABBCCDD → 0xBBAADDCC 2 = 32-bit swap: 0xAABBCCDD → 0xDDCCBBAA 3 = Half-DWORD swap: 0xAABBCCDD → 0xCCDDAABB
SCRATCH_UMSK	7:0	Update Mask for Scratch Pad Registers. One bit for each of the pad registers. 1 = Write the contents of the respective Scratch Pad register to Memory (using the SCRATCH_ADDR register as a base) whenever that Scratch Pad register is written. 0 = No write to memory. Default = 0

SCRATCH_ADDR
Scratch Register Update Address [RBIU]

Field Name	Bit(s)	Description
SCRATCH_ADDR	31:5	Memory Address to which the contents of scratchpad registers should be written. SCRATCH_REG0 is written to address: SCRATCH_ADDR + 0 SCRATCH_REG1 is written to address: SCRATCH_ADDR + 4 SCRATCH_REG2 is written to address: SCRATCH_ADDR + 8 … etc…

CP_ME_VS_EVENT_SRC
Vertex Shader Event Done -- Source [ME]

Field Name	Bit(s)	Description
VS_DONE_SRC	0	Source select to be written for a Vertex Shader Done event. Selects the source as 0-CP_ME_VS_EVENT_DATA or 1-CP_ME_VS_EVENT_DATA_SWM.

CP_ME_VS_EVENT_ADDR
Vertex Shader Event Done -- Address [ME]

Field Name	Bit(s)	Description
VS_DONE_ADDR	31:2	External memory address written for a Vertex Shader Done event. Read returns last value written to memory.
VS_DONE_SWAP	1:0	Swap used when writing the data. Read returns last value written to memory.

CP_ME_VS_EVENT_DATA
Vertex Shader Event Done -- Data [ME]

Field Name	Bit(s)	Description
VS_DONE_DATA	31:0	Data to be written for a Vertex Shader Done event. Read returns last value written to memory.

CP_ME_VS_EVENT_ADDR_SWM
Vertex Shader Event Done – S/W Managed - Address [ME]

Field Name	Bit(s)	Description
VS_DONE_ADDR_SWM	31:2	External memory address written for a Vertex Shader Done event. Read returns last value written to memory.
VS_DONE_SWAP_SWM	1:0	Swap used when writing the data. Read returns last value written to memory.

CP_ME_VS_EVENT_DATA_SWM
Vertex Shader Event Done – S/W Managed - Data [ME]

Field Name	Bit(s)	Description
VS_DONE_DATA_SWM	31:0	Data to be written for a Vertex Shader Done event. Read returns last value written to memory.

CP_ME_PS_EVENT_SRC
Pixel Shader Event Done -- Source [ME]

Field Name	Bit(s)	Description
PS_DONE_SRC	0	Source select to be written for a Pixel Shader Done event. Selects the source as 0-CP_ME_PS_EVENT_DATA or 1-CP_ME_PS_EVENT_DATA_SWM.

CP_ME_PS_EVENT_ADDR
Pixel Shader Event Done -- Address [ME]

Field Name	Bit(s)	Description
PS_DONE_ADDR	31:2	External memory address written for a Pixel Shader Done event. Read returns last value written to memory.
PS_DONE_SWAP	1:0	Swap used when writing the data. Read returns last value written to memory.

CP_ME_PS_EVENT_DATA
Pixel Shader Event Done -- Data [ME]

Field Name	Bit(s)	Description
PS_DONE_DATA	31:0	Data to be written for a Pixel Shader Done event. Read returns last value written to memory.

CP_ME_PS_EVENT_ADDR_SWM
Pixel Shader Event Done – S/W Managed - Address [ME]

Field Name	Bit(s)	Description
PS_DONE_ADDR_SWM	31:2	External memory address written for a Pixel Shader Done event. Read returns last value written to memory.
PS_DONE_SWAP_SWM	1:0	Swap used when writing the data. Read returns last value written to memory.

CP_ME_PS_EVENT_DATA_SWM
Pixel Shader Event Done – S/W Managed - Data [ME]

Field Name	Bit(s)	Description
PS_DONE_DATA_SWM	31:0	Data to be written for a Pixel Shader Done event. Read returns last value written to memory.

CP_ME_CF_EVENT_SRC
Cache Flush Event Done -- Source [ME]

Field Name	Bit(s)	Description
CF_DONE_SRC	0	Source selects Data or Counter value to be written for a Cache Flush Done event. Read returns last value written to memory.

CP_ME_CF_EVENT_ADDR
Cache Flush Event Done -- Address [ME]

Field Name	Bit(s)	Description
CF_DONE_ADDR	31:2	External memory address written for a Cache Flush Done event. Read returns last value written to memory.
CF_DONE_SWAP	1:0	Swap used when writing the data. Read returns last value written to memory.

CP_ME_CF_EVENT_DATA
Cache Flush Event Done -- Data [ME]

Field Name	Bit(s)	Description
CF_DONE_DATA	31:0	Data to be written for a Cache Flush Done event. Read returns last value written to memory.

CP_ME_NRT_ADDR
Non-Real-Time Write Address [ME] ** Written only by Micro Engine

Field Name	Bit(s)	Description
NRT_WRITE_ADDR	31:2	External memory address written for a non-Real-Time Mem_Write.
NRT_WRITE_SWAP	1:0	Swap used when writing the data.

CP_ME_NRT_DATA
Non-Real-Time Write Data [ME] ** Written only by Micro Engine

Field Name	Bit(s)	Description
NRT_WRITE_DATA	31:0	Data to be written for a non-Real-Time Mem_Write.

CP_ME_RT_ADDR
Real-Time Write Address [ME] ** Written only by Micro Engine

Field Name	Bit(s)	Description
RT_WRITE_ADDR	31:2	External memory address written for a Real-Time Mem_Write.
RT_WRITE_SWAP	1:0	Swap used when writing the data.

CP_ME_RT_DATA
Real-Time Write Data [ME] ** Written only by Micro Engine

Field Name	Bit(s)	Description
RT_WRITE_DATA	31:0	Data to be written for a Real-Time Mem_Write.

CP_INT_CNTL (R/W)
CP Interrupt Control [ME, DMA]

Field Name	Bit(s)	Description
RB_INT_MASK	31	Interrupt Mask for CP_INTERRUPT packet in the Ring Buffer The mask is used to enable the generation of the interrupt to the Host. 0 – Disabled (Default) 1 – Enabled: Interrupt will generate interrupt to Host.
IB1_INT_MASK	30	Interrupt Mask for CP_INTERRUPT packet in IB1 Stream (See bit 31 for definition).
IB2_INT_MASK	29	Interrupt Mask for CP_INTERRUPT packet in IB2 Stream (See bit 31 for definition).
DMA_INT_MASK	28	Interrupt Mask for the DMA Engine (See bit 31 for definition).
IB_ERROR_MASK	27	Interrupt Mask for the Indirect Buffer Start Error (See bit 31 for definition).
RESERVED_BIT_ERROR_MASK	26	Interrupt Mask for the Reserved Bit Check Error (See bit 31 for definition).
PROTECTED_MODE_ERROR_MASK	25	Interrupt Mask for the Protected Mode Error (See bit 31 for definition).
OPCODE_ERROR_MASK	24	Interrupt Mask for the Opcode Check Error (See bit 31 for definition).
T0_T1_PACKET_IN_IB_MASK	23	Interrupt Mask for the Type 0/1 Packet Check in the IBs (See bit 31 for definition).
RT_RESERVED_BIT_ERROR_MASK	22	Interrupt Mask for the Real-Time Reserved Bit Check Error (See bit 31 for definition).
RT_PROTECTED_MODE_ERROR_MASK	21	Interrupt Mask for the Real-Time Protected Mode Error (See bit 31 for definition).
RT_OPCODE_ERROR_MASK	20	Interrupt Mask for the Real-Time Opcode Check Error (See bit 31 for definition).
SW_INT	19	Interrupt Mask for the Software Interrupt (See bit 31 for definition).
SCRATCH_INT_MASK	18	Interrupt Mask for the Scratch Register Compare. (See bit 31 for definition).
Reserved	17:16	Reserved
RTS15_INT_MASK	15	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #15 (See bit 31 for definition).
RTS14_INT_MASK	14	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #14 (See bit 31 for definition).
RTS13_INT_MASK	13	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #13 (See bit 31 for definition).
RTS12_INT_MASK	12	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #12 (See bit 31 for definition).
RTS11_INT_MASK	11	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #11 (See bit 31 for definition).
RTS10_INT_MASK	10	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #10 (See bit 31 for definition).
RTS9_INT_MASK	9	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #9 (See bit 31 for definition).
RTS8_INT_MASK	8	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #8 (See bit 31 for definition).
RTS7_INT_MASK	7	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #7 (See bit 31 for definition).
RTS6_INT_MASK	6	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #6 (See bit 31 for definition).
RTS5_INT_MASK	5	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #5 (See bit 31 for definition).
RTS4_INT_MASK	4	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #4 (See bit 31 for definition).
RTS3_INT_MASK	3	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #3 (See bit 31 for definition).
RTS2_INT_MASK	2	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #2 (See bit 31 for definition).
RTS1_INT_MASK	1	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #1 (See bit 31 for definition).
RTS0_INT_MASK	0	Interrupt Mask for CP_INTERRUPT packet in the RT Slice #0 (See bit 31 for definition).

CP_INT_STATUS (R/W)
CP Interrupt STATUS [ME, DMA]

Field Name	Bit(s)	Description
RB_INT_STAT	31	Interrupt status for CP_INTERRUPT packet in Ring Buffer. ***Default = 0***. Read: 0 – No Event (Interrupt did not occur). 1 – Even Occurred. This will be set even if the corresponding mask is ‘0’ in the CP_INT_CNTL register. Write: 0 – No affect. 1 – Set Interrupt Status (For state restoration from power-down). *** Soft Resetting the CP will clear this register ***
IB1_INT_STAT	30	Interrupt status for CP_INTERRUPT packet in IB1 stream (See bit 31 for definition)
IB2_INT_STAT	29	Interrupt status for CP_INTERRUPT packet in IB2 stream (See bit 31 for definition)
DMA_INT_STAT	28	Interrupt status for the DMA Engine (See bit 31 for definition)
IB_ERROR_STAT	27	Interrupt status for the Indirect Buffer Start Error (See bit 31 for definition)
RESERVED_BIT_ERROR_STAT	26	Interrupt status for the Reserved Bit Check Error (See bit 31 for definition)
PROTECTED_MODE_ERROR_STAT	25	Interrupt status for the Protected Mode Error (See bit 31 for definition)
OPCODE_ERROR_STAT	24	Interrupt status for the Opcode Check Error (See bit 31 for definition)
T0_T1_PACKET_IN_IB_STAT	23	Interrupt status for the Type 0/1 Packet Check in the IBs (See bit 31 for definition).
RT_RESERVED_BIT_ERROR_STAT	22	Interrupt status for the Real-Time Reserved Bit Check Error (See bit 31 for definition)
RT_PROTECTED_MODE_ERROR_STAT	21	Interrupt status for the Real-Time Protected Mode Error (See bit 31 for definition)
RT_OPCODE_ERROR_STAT	20	Interrupt status for the Real-Time Opcode Check Error (See bit 31 for definition)
SW_INT	19	Interrupt status for the Software Interrupt (See bit 31 for definition).
SCRATCH_INT_STAT	18	Interrupt status for the Scratch Register Compare Interrupt (See bit 31 for definition).
Reserved	17:16	Reserved.
RTS15_INT_STAT	15	Interrupt status for CP_INTERRUPT packet in RT Slice #15 (See bit 31 for definition).
RTS14_INT_STAT	14	Interrupt status for CP_INTERRUPT packet in RT Slice #14 (See bit 31 for definition).
RTS13_INT_STAT	13	Interrupt status for CP_INTERRUPT packet in RT Slice #13 (See bit 31 for definition).
RTS12_INT_STAT	12	Interrupt status for CP_INTERRUPT packet in RT Slice #12 (See bit 31 for definition).
RTS11_INT_STAT	11	Interrupt status for CP_INTERRUPT packet in RT Slice #11 (See bit 31 for definition).
RTS10_INT_STAT	10	Interrupt status for CP_INTERRUPT packet in RT Slice #10 (See bit 31 for definition).
RTS9_INT_STAT	9	Interrupt status for CP_INTERRUPT packet in RT Slice #9 (See bit 31 for definition).
RTS8_INT_STAT	8	Interrupt status for CP_INTERRUPT packet in RT Slice #8 (See bit 31 for definition).
RTS7_INT_STAT	7	Interrupt status for CP_INTERRUPT packet in RT Slice #7 (See bit 31 for definition).
RTS6_INT_STAT	6	Interrupt status for CP_INTERRUPT packet in RT Slice #6 (See bit 31 for definition).
RTS5_INT_STAT	5	Interrupt status for CP_INTERRUPT packet in RT Slice #5 (See bit 31 for definition).
RTS4_INT_STAT	4	Interrupt status for CP_INTERRUPT packet in RT Slice #4 (See bit 31 for definition).
RTS3_INT_STAT	3	Interrupt status for CP_INTERRUPT packet in RT Slice #3 (See bit 31 for definition).
RTS2_INT_STAT	2	Interrupt status for CP_INTERRUPT packet in RT Slice #2 (See bit 31 for definition).
RTS1_INT_STAT	1	Interrupt status for CP_INTERRUPT packet in RT Slice #1 (See bit 31 for definition).
RTS0_INT_STAT	0	Interrupt status for CP_INTERRUPT packet in RT Slice #0 (See bit 31 for definition).

CP_INT_ACK (WO)
CP Interrupt Acknowledge [ME, DMA]

Field Name	Bit(s)	Description
RB_INT_ACK	31	Interrupt acknowledge for CP_INTERRUPT packet in the Ring Buffer. Write: 0 – No affect. 1 – Clear just this Interrupt Status Bit (Non-Persistent Write).
IB1_INT_ACK	30	Interrupt acknowledge for CP_INTERRUPT packet in IB1 stream (See bit 31).
IB2_INT_ACK	29	Interrupt acknowledge for CP_INTERRUPT packet in IB2 stream (See bit 31).
DMA_INT_ACK	28	Interrupt acknowledge for Read Error (See bit 31 for definition).
IB_ERROR	27	Interrupt acknowledge for the Indirect Buffer Start Error (See bit 31 for definition).
RESERVED_BIT_ERROR_ACK	26	Interrupt acknowledge for the Reserved Bit Check Error (See bit 31 for definition).
PROTECTED_MODE_ERROR_ACK	25	Interrupt acknowledge for the Protected Mode Error (See bit 31 for definition).
OPCODE_ERROR_ACK	24	Interrupt acknowledge for the Opcode Check Error (See bit 31 for definition).
T0_T1_PACKET_IN_IB_ACK	23	Interrupt acknowledge for the Type 0/1 Packet Check in the IBs (See bit 31 for definition).
RT_RESERVED_BIT_ERROR_ACK	22	Interrupt acknowledge for the Real-Time Reserved Bit Check Error (See bit 31 for definition).
RT_PROTECTED_MODE_ERROR_ACK	21	Interrupt acknowledge for the Real-Time Protected Mode Error (See bit 31 for definition).
RT_OPCODE_ERROR_ACK	20	Interrupt acknowledge for the Real-Time Opcode Check Error (See bit 31 for definition).
SW_INT_ACK	19	Interrupt acknowledge for the Software Interrupt (See bit 31 for definition).
SCRATCH_INT_ACK	18	Interrupt acknowledge for the Scratch Register Compare Interrupt (See bit 31 for definition).
Reserved	17:16	Reserved.
RTS15_INT_ACK	15	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #15 (See bit 31 for definition).
RTS14_INT_ACK	14	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #14 (See bit 31 for definition).
RTS13_INT_ACK	13	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #13 (See bit 31 for definition).
RTS12_INT_ACK	12	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #12 (See bit 31 for definition).
RTS11_INT_ACK	11	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #11 (See bit 31 for definition).
RTS10_INT_ACK	10	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #10 (See bit 31 for definition).
RTS9_INT_ACK	9	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #9 (See bit 31 for definition).
RTS8_INT_ACK	8	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #8 (See bit 31 for definition).
RTS7_INT_ACK	7	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #7 (See bit 31 for definition).
RTS6_INT_ACK	6	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #6 (See bit 31 for definition).
RTS5_INT_ACK	5	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #5 (See bit 31 for definition).
RTS4_INT_ACK	4	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #4 (See bit 31 for definition).
RTS3_INT_ACK	3	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #3 (See bit 31 for definition).
RTS2_INT_ACK	2	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #2 (See bit 31 for definition).
RTS1_INT_ACK	1	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #1 (See bit 31 for definition).
RTS0_INT_ACK	0	Interrupt acknowledge for CP_INTERRUPT packet RT Slice #0 (See bit 31 for definition).

SCRATCH_CMP_HI
Scratch Register Compare Value High

Field Name	Bit(s)	Description
SCRATCH_CMP_HIGH	31:0	Most-Significant DWORD for Scratch Register Comparison This is programmed by the Driver before enabling the Scratch Compare Interrupt in the SCRATCH_CMP_CNTL register.

SCRATCH_CMP_LO
Scratch Register Compare Value Low

Field Name	Bit(s)	Description
SCRATCH_CMP_LOW	31:0	Least-Significant DWORD for Scratch Register Comparison This is programmed by the Driver before enabling the Scratch Compare Interrupt in the SCRATCH_CMP_CNTL register.

SCRATCH_CMP_CNTL
Scratch Register Compare Control

Field Name	Bit(s)	Description
COMPARE_ENABLE	31	Scratch Compare Enable. Default = 0.
Reserved	30:11	Reserved
TEST_SELECT	10:8	Test Select: 000 – A >= B (Default) 001 – A <= B 010 – A > B 011 – A < B 100 – A == B 101 – A != B Where: A = Scratch Register Pair B = Driver-Supplied Pair (SCRATCH_CMP_HI / LO)
Reserved	7:2	Reserved
PAIR_SELECT	1:0	Scratch Pair Select: 00 – Scratch Registers 0 (Low) & 1 (High) 01 – Scratch Registers 2 (Low) & 3 (High) 10 – Scratch Registers 4 (Low) & 5 (High) 11 – Scratch Registers 6 (Low) & 7 (High)

7.20 Command Debug Registers (Updated: 09-13-2002)

These registers are used to read the contents of the command cache queues that store the pre-parsed command and state data.

CP_CMD_INDEX (WO)
Command Index [ROQ]

Field Name	Bit(s)	Description
Reserved	31:18	Reserved
CMD_QUEUE_SEL	17:16	Selects queue to read: 00 – N/A 01 – CS Queue 10 – ST Queue 11 – ME Queue
Reserved	15:10	Reserved
CMD_INDEX	9:0	DWORD Read Address for the Command Queue. The address in the queue where the stream data is stored is dependant on the programming of the threshold values. The CMD_INDEX auto-increments after the CP_CMD_DATA register is read. See the CP_QUEUE_THRESHOLDS register.

CP_CMD_DATA (RO)
Command Data [ROQ]

Field Name	Bit(s)	Description
CMD_DATA	31:0	Data from the Command Queue, from location pointed to by the CP_CMD_ADDR register.

7.21 State Management Registers (Updated: 04-04-2003)

The following registers are used to read the contents of the storage elements in the CP for State Management. The following elements are read at the listed address ranges:

Index	Description
0x0	Sub-Block 0 Last Pointer [31:5] & Size [3:0]
0x1	Sub-Block 1 Last Pointer [31:5] & Size [3:0]
0x2	Sub-Block 2 Last Pointer [31:5] & Size [3:0]
0x3	Sub-Block 3 Last Pointer [31:5] & Size [3:0]
0x4	Sub-Block 4 Last Pointer [31:5] & Size [3:0]
0x5	Sub-Block 5 Last Pointer [31:5] & Size [3:0]
0x6	Sub-Block 6 Last Pointer [31:5] & Size [3:0]
0x7	Sub-Block 7 Last Pointer [31:5] & Size [3:0]
0x8	Vertex Shader Last Base [31:5]
0x9	Vertex Shader Last Size [13:0]
0xA	Pixel Shader Last Base [31:5]
0xB	Pixel Shader Last Size [13:0]
0xC	Valid Flags: Bit 0: Sub-Block 0 Valid (Default = 0) – Cleared on Soft Reset. Bit 1: Sub-Block 1 Valid (Default = 0) – Cleared on Soft Reset. Bit 2: Sub-Block 2 Valid (Default = 0) – Cleared on Soft Reset. Bit 3: Sub-Block 3 Valid (Default = 0) – Cleared on Soft Reset. Bit 4: Sub-Block 4 Valid (Default = 0) – Cleared on Soft Reset. Bit 5: Sub-Block 5 Valid (Default = 0) – Cleared on Soft Reset. Bit 6: Sub-Block 6 Valid (Default = 0) – Cleared on Soft Reset. Bit 7: Sub-Block 7 Valid (Default = 0) – Cleared on Soft Reset. Bit 8: Vertex Shader Last Valid (Default = 0) – Cleared on Soft Reset. Bit 9: Pixel Shader Last Valid (Default = 0) – Cleared on Soft Reset. Bit 10: ALU Constant Base Valid (Default=0) – Cleared on Soft Reset. Bit 11: Texture Constant Base Valid (Default=0) – Cleared on Soft Reset. Bit 12: Register Base Valid (Default = 0) – Cleared on Soft Reset. Bit 13: Boolean Constant Valid (Default = 0) – Cleared on Soft Reset. Bit 14: Loop Constant Valid (Default = 0) – Cleared on Soft Reset. Bits 31:15 – Reserved
0xD	Bits 7:0 : Constant Coherency Counter Bit 8 : ALU Constant Global Write Enable Bit 9 : Texture Constant Global Write Enable Bit 10 : Register Global Write Enable Bit 11 : Boolean Constant Write Enable Bit 12 : Loop Constant Write Enable Bits 28:15 – Reserved Bit 29 : 2D/3D Mode Flag. Bit 30: Real-time predicated test result. Bit 31: Non-real-time predicated test result.
0xE	Bits 19:0 : Indirect #1 Size FIFO Bits 23:20 : Reserved Bits 26:24 : Number of Outstanding Indirect #1 Requests Bits 31:27 : Reserved
0xF	Same as 0xE, but for Indirect #2
0x10	ALU_Constant_Base[31:13]
0x11	TEX_Constant_Base[31:13]
0x12	REG_Constant_Base[31:13]
0x13	BOOLEAN_Constant_Base[31:13]
0x14	LOOP_Constant_Base[31:13]
0x16 to 0x1F	Reserved

CP_STATE_DEBUG_INDEX (WO)
State Management Debug Index [PFP]

Field Name	Bit(s)	Description
Reserved	31:5	Unused
STATE_DEBUG_INDEX	4:0	Index. See above text for mapping. This index register auto increments when the data register is read.

CP_STATE_DEBUG_DATA (RO)
State Management Debug Data [PFP]

Field Name	Bit(s)	Description
STATE_DEBUG_DATA	31:0	Data. See above text for interpretation.

7.22 CP Status Registers (Updated: 03-12-2003)

CP_CSQ_RB_STAT (RO) Command Stream Queue Ring Buffer Status [ROQ]		
Field Name	**Bit(s)**	**Description**
Reserved	31:26	Reserved
CSQ_WPTR_PRIMARY	25:16	Current Write Pointer of the Primary Stream Data in the CSQ.
Reserved	15:10	Reserved
CSQ_RPTR_PRIMARY	9:0	Current Read Pointer of the Primary Stream Data in the CSQ.

CP_CSQ_IB1_STAT (RO) Command Stream Queue Indirect Buffer #1 Status [ROQ]		
Field Name	**Bit(s)**	**Description**
Reserved	31:26	Reserved
CSQ_WPTR_INDIRECT1	25:16	Current Write Pointer of the IB1 Stream data in the CSQ.
Reserved	15:10	Reserved
CSQ_RPTR_INDIRECT1	9:0	Current Read Pointer of the IB1 Stream data in the CSQ.

CP_CSQ_IB2_STAT (RO) Command Stream Queue Indirect Buffer #2 Status [ROQ]		
Field Name	**Bit(s)**	**Description**
Reserved	31:26	Reserved
CSQ_WPTR_INDIRECT2	25:16	Current Write Pointer of the IB2 Stream data in the CSQ.
Reserved	15:10	Reserved
CSQ_RPTR_INDIRECT2	9:0	Current Read Pointer of the IB2 Stream data in the CSQ.

CP_STQ_RT_STAT (RO) State Queue Real-Time Status [ROQ]		
Field Name	**Bit(s)**	**Description**
Reserved	31:26	Reserved
STQ_WPTR_RT	25:16	Current Write Pointer of the RT stream data in the STQ.
Reserved	15:10	Reserved
STQ_RPTR_RT	9:0	Current Read Pointer of the RT stream data in the STQ.

CP_STQ_ST_STAT (RO) State Queue State Data Status [ROQ]		
Field Name	**Bit(s)**	**Description**
Reserved	31:26	Reserved
STQ_WPTR_ST	25:16	Current Write Pointer of the State data in the STQ.
Reserved	15:10	Reserved
STQ_RPTR_ST	9:0	Current Read Pointer of the State data in the STQ.

CP_STQ_RT_ST_STAT (RO)
State Queue Real-Time State Data Status [ROQ]

Field Name	Bit(s)	Description
Reserved	31:26	Reserved
STQ_WPTR_RT_ST	25:16	Current Write Pointer of the RT State Data in the STQ.
Reserved	15:10	Reserved
STQ_RPTR_RT_ST	9:0	Current Read Pointer of the RT State Data in the STQ.

CP_MEQ_ STAT (RO)
Command Stream Queue Status [PFP]

Field Name	Bit(s)	Description
Reserved	31:26	Reserved
MEQ_WPTR	25:16	Current Write Pointer in the MEQ.
Reserved	15:10	Reserved
MEQ_RPTR	9:0	Current Read Pointer in the MEQ.

CP_NON_PREFETCH_CNTRS (RO)
Non Pre-Fetch Counters [PFP]

Field Name	Bit(s)	Description
Reserved	31:11	Reserved
IB2_COUNTER	10:8	Count of Indirect Buffer #2 Pre-Fetches In-Flight.
Reserved	7:3	Reserved
IB1_COUNTER	2:0	Count of Indirect Buffer #1 Pre-Fetches In-Flight.

CP_STAT (RO)
Command Processor Status Register [RBIU]

Field Name	Bit(s)	Description
MIU_WR_BUSY	0	Write Path in Memory Interface Unit is Busy (RT and Non-RT).
MIU_RD_REQ_BUSY	1	Read Request Path in Memory Interface Unit is Busy (RTEE, RT, or Non-RT)
MIU_RD_RETURN_BUSY	2	Memory Interface Unit is Expecting Data from Memory Hub (RTEE, RT and Non-RT)
RBIU_BUSY	3	The CP's Register Bus Interface Unit is Busy. This bit will be asserted when the CP_STAT register is read.
RCIU_BUSY	4	Read/Write Interface to RBBM is Busy with Non-RT Transactions.
CSF_RING_BUSY	5	Command Stream Fetcher has more to fetch for the Ring Buffer.
CSF_INDIRECTS_BUSY	6	Command Stream Fetcher has more to fetch for Indirect Buffers in the Ring.
CSF_INDIRECT2_BUSY	7	Command Stream Fetcher has more to fetch for Indirect Buffer #2's.
Reserved	8	Reserved.
CSF_ST_BUSY	9	Command Stream Fetcher has more to fetch for State Data.
CSF_BUSY	10	Command Stream Fetcher is Busy for Non Real Time.
RING_QUEUE_BUSY	11	Command Data Needs to Be Parsed for the Ring Buffer.
INDIRECTS_QUEUE_BUSY	12	Command Data Needs to be Parsed for the Indirect Buffers in the Ring Buffer.
INDIRECT2_QUEUE_BUSY	13	Command Data Needs to be Parsed for Indirect #2 Buffers in Indirect Buffer #1.
Reserved	15:14	Reserved
ST_QUEUE_BUSY	16	Pre-Fetched State Data Needs to be Taken by the Micro Engine.
PFP_BUSY	17	Pre-Fetch Parser is Busy Processing Either a Real-Time or Non-Real-Time Stream.
MEQ_BUSY	18	Micro Engine Queue Contains Data from the PFP.
Reserved	20:19	Reserved
MIU_WC_STALL	21	MIU is Waiting for Write Confirm from the MH.
CP_NRT_BUSY	22	CP is Busy Processing Non-Real-Time Streams, DMA Operation, or Scratch Memory Write. This status bit does ***not*** include pending read/write transactions from the RBBM.
3D_BUSY	23	There is at least one context reserved for 3D processing.
2D_BUSY	24	There is at least one context allocated for 2D processing.
Reserved	25	Reserved.
ME_BUSY	26	Non-Real-Time Micro Engine is Busy.
RTEE_BUSY	27	Real-Time Event Engine is Busy (Polling or Processing a Stream)
DMA_BUSY	28	DMA Engine is Busy.
ME_WC_BUSY	29	Micro Engine is waiting for write confirmations from the Memory Hub or the CP/SC write confirm counter is not zero.
MIU_WC_TRACK_FIFO_EMPTY	30	MIU's Write Confirm Tracking FIFO is Empty.
CP_BUSY	31	Any block in the CP is Busy (Non-Real-Time or Real-Time). This status bit will be asserted when the CP_STAT register is read.

CP_RT_STAT (RO)
Command Processor Real-Time Status Register [RBIU]

Field Name	Bit(s)	Description
Reserved	3:0	Reserved
RCIU_RT_BUSY	4	Read/Write Interface to RBBM is Busy with RT Transactions.
Reserved	8:5	Reserved.
CSF_RT_ST_BUSY	9	Command Stream Fetcher has more to fetch for Real-Time State Data.
CSF_RT_BUSY	10	Command Stream Fetcher is Busy for Real-Time.
Reserved	13:11	Reserved
RT_QUEUE_BUSY	14	Real-Time Re-Ordering Queue has Command Data.
RT_ST_QUEUE_BUSY	15	Real-Time State Data Queue has State Data.
Reserved	16	Reserved
PFP_RT_BUSY	17	Pre-Fetch Parser is Busy Processing Real-Time Stream.
Reserved	20:18	Reserved
MEQ_RT_BUSY	21	Micro Engine Output Queue has Real-Time Transactions.
Reserved	24:22	Reserved
ME_RT_BUSY	25	Real-Time Micro Engine is Busy
Reserved	26	Reserved
RTEE_RT_BUSY	27	Real-Time Event Engine is Busy Fetching a Real-Time Stream.
Reserved	30:28	Reserved
CP_RT_BUSY	31	CP is Processing a Real-Time Stream.

CP_MIU_TAG_STAT0 (RO)
Command Processor MIU Tag Return Status 0

Field Name	Bit(s)	Description
TAG_0_STAT	0	If set the MH has not returned this tag to the CP yet.
TAG_1_STAT	1	If set the MH has not returned this tag to the CP yet.
TAG_2_STAT	2	If set the MH has not returned this tag to the CP yet.
TAG_3_STAT	3	If set the MH has not returned this tag to the CP yet.
TAG_4_STAT	4	If set the MH has not returned this tag to the CP yet.
TAG_5_STAT	5	If set the MH has not returned this tag to the CP yet.
TAG_6_STAT	6	If set the MH has not returned this tag to the CP yet.
TAG_7_STAT	7	If set the MH has not returned this tag to the CP yet.
TAG_8_STAT	8	If set the MH has not returned this tag to the CP yet.
TAG_9_STAT	9	If set the MH has not returned this tag to the CP yet.
TAG_10_STAT	10	If set the MH has not returned this tag to the CP yet.
TAG_11_STAT	11	If set the MH has not returned this tag to the CP yet.
TAG_12_STAT	12	If set the MH has not returned this tag to the CP yet.
TAG_13_STAT	13	If set the MH has not returned this tag to the CP yet.
TAG_14_STAT	14	If set the MH has not returned this tag to the CP yet.
TAG_15_STAT	15	If set the MH has not returned this tag to the CP yet.
TAG_16_STAT	16	If set the MH has not returned this tag to the CP yet.
TAG_17_STAT	17	If set the MH has not returned this tag to the CP yet.
TAG_18_STAT	18	If set the MH has not returned this tag to the CP yet.
TAG_19_STAT	19	If set the MH has not returned this tag to the CP yet.
TAG_20_STAT	20	If set the MH has not returned this tag to the CP yet.
TAG_21_STAT	21	If set the MH has not returned this tag to the CP yet.
TAG_22_STAT	22	If set the MH has not returned this tag to the CP yet.
TAG_23_STAT	23	If set the MH has not returned this tag to the CP yet.
TAG_24_STAT	24	If set the MH has not returned this tag to the CP yet.
TAG_25_STAT	25	If set the MH has not returned this tag to the CP yet.
TAG_26_STAT	26	If set the MH has not returned this tag to the CP yet.
TAG_27_STAT	27	If set the MH has not returned this tag to the CP yet.
TAG_28_STAT	28	If set the MH has not returned this tag to the CP yet.
TAG_29_STAT	29	If set the MH has not returned this tag to the CP yet.
TAG_30_STAT	30	If set the MH has not returned this tag to the CP yet.
TAG_31_STAT	31	If set the MH has not returned this tag to the CP yet.

CP_MIU_TAG_STAT1 (RO)
Command Processor MIU Tag Return Status 0

Field Name	Bit(s)	Description
TAG_32_STAT	0	If set the MH has not returned this tag to the CP yet.
TAG_33_STAT	1	If set the MH has not returned this tag to the CP yet.
TAG_34_STAT	2	If set the MH has not returned this tag to the CP yet.
TAG_35_STAT	3	If set the MH has not returned this tag to the CP yet.
TAG_36_STAT	4	If set the MH has not returned this tag to the CP yet.
TAG_37_STAT	5	If set the MH has not returned this tag to the CP yet.
TAG_38_STAT	6	If set the MH has not returned this tag to the CP yet.
TAG_39_STAT	7	If set the MH has not returned this tag to the CP yet.
TAG_40_STAT	8	If set the MH has not returned this tag to the CP yet.
TAG_41_STAT	9	If set the MH has not returned this tag to the CP yet.
TAG_42_STAT	10	If set the MH has not returned this tag to the CP yet.
TAG_43_STAT	11	If set the MH has not returned this tag to the CP yet.
TAG_44_STAT	12	If set the MH has not returned this tag to the CP yet.
TAG_45_STAT	13	If set the MH has not returned this tag to the CP yet.
TAG_46_STAT	14	If set the MH has not returned this tag to the CP yet.
TAG_47_STAT	15	If set the MH has not returned this tag to the CP yet.
Reserved	30:16	Reserved.
INVALID_RETURN_TAG	31	Set if the MH ever returns a tag to the CP greater than 0x2F.

7.23 Real-Time Event Engine Control Registers (Update: 05-24-2002)

There are 16 sets (0-15) of the following registers. These registers control the operation of the Real-Time event engine in the CP.
Note: Interrupts from the Real-Time streams are done by submitting a CP_INTERRUPT packet in the stream.

CP_RT0_COMMAND
Real Time Stream 0 Command [RTEE]

Field Name	Bit(s)	Description
QUEUED_REG_PATH	31	Use queued path through RBBM to perform register polling. Default = 1.
Reserved	30	Reserved.
EXECUTED (RO)	29	Real-Time Stream has Executed as a Result of the Arm. Re-Arming and Reset (Soft and Hard) clears this status. When the RTS is activated, this status is set.
TRIGGERED_WAIT (RO)	28	Set by the CP if the RT stream has triggered and is waiting to be executed. This status is de-asserted when the RT stream event is executed (i.e. accepted by the RT stream fetcher). The ACTIVE status then reflects the status. This bit status is cleared by reset (Soft and Hard).
ACTIVE (RO)	27	Set if RT stream is being fetched by the CP.
ARM	26	0 => Disabled, 1=> Armed (Default is Disabled). This enables the Real-Time stream. The ARM flag is automatically disabled by the CP once the RT stream is executed. It stays set while the event has triggered but not executed (i.e. In TRIGGERED_WAIT) It is the responsibility of the RT stream content to re-arm itself if needed. Clearing this bit (i.e. writing '0') disables a real-time stream that has triggered and is waiting to be processed (i.e. TRIGGERED_WAIT). ***This bit is cleared when the CP is reset (Soft and Hard).***
Reserved	25:21	Reserved for Future Consideration
POLL_INTERVAL_SOURCE	20:19	Selects One of the Programmable Polling Interval Values: 00 – RT_POLL_INTERVAL0 01 – RT_POLL_INTERVAL1 1x – RT_POLL_INTERVAL2 The poll counters are free running; so all RT streams that select the same counter will try to poll at the same time. The arbitration for access to the RBBM/MH will serialize these polling operations. Default = 00.
TRIGGER_SOURCE	18:16	Selects source of Trigger: 000 – Signal. Need to Set SIGNAL_SOURCE and SIGNAL_CONDITIONING 001 – Register Location (Polling) 010 – Memory Location (Polling) 011 – Reserved. 100 – Display #0 End of Line / Start of Frame Counter 101 – Display #1 End of Line / Start of Frame Counter 110 – VIP End of Line / End of Frame Counter 111 – Reserved. Default = 000.
Reserved	15	Reserved for Future Considerations
SIGNAL_CONDITIONING	14:13	Signal Conditioning Select 00 – None. Event will trigger if signal is asserted. 01 – Rising Edge Detect 10 – Falling Edge Detect 11 – Invert. Event will trigger if signal is de-asserted. Default = 00.
SIGNAL_SOURCE	12:8	Signal Source Select Selects one of 32 Signals to Monitor when Trigger_Source = "000". The programmed value selects the bit (0 to 31) of the CP_rts_discretes[31:0] input of the CP. *See CP_RTS_Connections.doc for the bit-to-signal mapping.* Default = 00000.
Reserved	7:3	Reserved for Future Considerations
COMPARE_FUNCTION	2:0	Compare Function Select (Used for Polling Mode): 000 – Always (i.e. Do Now) 001 – Masked Value Less Than (<) Reference 010 – Masked Value Less Than or Equal (<=) to Reference 011 – Masked Value Equal (=) to Reference 100 – Masked Value Not Equal (!=) to Reference 101 – Masked Value Greater Than or Equal (>=) to Reference 110 – Masked Value Greater Than (>) Reference 111 – Always (i.e. Do Now) Default = 000

CP_RT0_POLL_ADDRESS
Real Time Stream 0 Polling Address [RTEE]

Field Name	Bit(s)	Description
POLL_ADDRESS	31:2	Address CP Polls for Register and Memory Trigger Sources if polling is used to trigger the RT stream. Note that for Register Polling, only bits 16:2 apply.
POLL_SWAP	1:0	Endian Swap Applied to Polling Memory Read Return Data. Default = 0.

CP_RT0_REFERENCE
Real Time Stream 0 Reference [RTEE]

Field Name	Bit(s)	Description
REFERENCE	31:0	Reference Value for Source Compare Function.

CP_RT0_MASK
Real Time Stream 0 Mask [RTEE]

Field Name	Bit(s)	Description
MASK	31:0	Mask Applied to Source Data before Comparison with Reference. Default = 0xFFFFFFFF

CP_RT0_BADR
Real Time Stream 0 Base Address [RTEE]

Field Name	Bit(s)	Description
BADR	31:5	External Memory Base address of the real-time stream.
Reserved	4:0	Reserved.

CP_RT0_SIZE
Real Time Stream 0 Size [RTEE]

Field Name	Bit(s)	Description
Reserved	31:20	Reserved.
SIZE	19:0	Size in DWORDs for the real-time stream.

Note: Register sets 1-15 are not listed but have identical format to the register set shown above.

CP_RT_POLL_INTERVAL (R/W)
Real-Time Polling Interval Count [RTEE]

Field Name	Bit(s)	Description
Reserved	31:30	Reserved
RT_POLL_INTERVAL2	29:20	Programmed with the time interval between polling operations for real-time commands. The counter decrements every 16 core clocks, so the maximum time is 1024 * 16 = 16384 clocks (40.96 uSec). Default = 0x3FF.
RT_POLL_INTERVAL1	19:10	Programmed with the time interval between polling operations for real-time commands. The counter decrements every 16 core clocks, so the maximum time is 1024 * 16 = 16384 clocks (40.96 uSec). Default = 0x3FF.
RT_POLL_INTERVAL0	9:0	Programmed with the time interval between polling operations for real-time commands. The counter decrements every 16 core clocks, so the maximum time is 1024 * 16 = 16384 clocks (40.96 uSec). Default = 0x3FF.

CP_VIP_EOL_EOF_COUNTER (R/W)
VIP End of Line / End of Frame Counter [RTEE]

Field Name	Bit(s)	Description
VIP_EOF_COUNTER	31:13	End of Frame Counter for VIP. This can be reset to any value and is used by the Real-Time Event Engine. It counts on every VIP_CP_EOF pulse from the VIP and can count 512K frames. The counter just rolls over at its terminal count. Default=0.
VIP_EOL_COUNTER	12:0	End of Line Counter for VIP. This can be reset to any value and is used by the Real-Time Event Engine. It counts on every VIP_CP_EOL pulse from the VIP and can count up to 8K lines. The counter resets on the VIP_CP_EOF pulse. Default=0.

CP_D1_EOL_SOF_COUNTER (R/W)
Display 1 End of Line / Start of Frame Counter [RTEE]

Field Name	Bit(s)	Description
DISP1_SOF_COUNTER	31:13	Start of Frame Counter for Display 1. This can be reset to any value and is used by the Real-Time Event Engine. It counts on every DISP1_CP_SOF pulse from the Display Engine and can count 512K frames. The counter just rolls over at its terminal count. Default=0.
DISP1_EOL_COUNTER	12:0	End of Line Counter for Display 1. This can be reset to any value and is used by the Real-Time Event Engine. It counts on every DISP1_CP_EOL pulse from the Display Engine and can count up to 8K lines. The counter resets on the DISP1_CP_SOF pulse. Default=0.

CP_D2_EOL_SOF_COUNTER (R/W)
Display 2 End of Line / Start of Frame Counter [RTEE]

Field Name	Bit(s)	Description
DISP2_SOF_COUNTER	31:13	Start of Frame Counter for Display 1. This can be reset to any value and is used by the Real-Time Event Engine. It counts on every DISP2_CP_SOF pulse from the Display Engine and can count 512K frames. The counter just rolls over at its terminal count. Default=0.
DISP2_EOL_COUNTER	12:0	End of Line Counter for Display 1. This can be reset to any value and is used by the Real-Time Event Engine. It counts on every DISP2_CP_EOL pulse from the Display Engine and can count up to 8K lines. The counter resets on the DISP2_CP_SOF pulse. Default=0.

7.24 “Viz Query A” Registers (Update: 09-13-2002)

The following register hold the visibility results from the Scan Converter to be used by the Viz Query function. See the VIZ_QUERY packet in the PM4 Specification for details on these bits.

CP_NV_FLAGS_0 (R/W)
Not Visible Flags for Viz Query [PFP]

Field Name	Bit(s)	Description
NV_15	31:30	Bit 31: DISCARD (Default = 0) – Resets on Soft Reset. Bit 30: END_RCVD (Default = 0) – Resets on Soft Reset.
NV_14	29:28	See NV_15 Description.
NV_13	27:26	See NV_15 Description.
NV_12	25:24	See NV_15 Description.
NV_11	23:22	See NV_15 Description.
NV_10	21:20	See NV_15 Description.
NV_9	19:18	See NV_15 Description.
NV_8	17:16	See NV_15 Description.
NV_7	15:14	See NV_15 Description.
NV_6	13:12	See NV_15 Description.
NV_5	11:10	See NV_15 Description.
NV_4	9:8	See NV_15 Description.
NV_3	7:6	See NV_15 Description.
NV_2	5:4	See NV_15 Description.
NV_1	3:2	See NV_15 Description.
NV_0	1:0	See NV_15 Description.

Note: CP_NV_FLAGS_1 through CP_NV_FLAGS_3 need to be added to the specification once the details of the Viz Query are established. The 4 NV flag registers assume that 64 NV flags are supported.

7.25 Predicate Registers (Update: 08-27-2003)

CP_BIN_MASK_LO (R/W)
Lower 32-bits of Bin Mask [PFP]

Field Name	Bit(s)	Description
BIN_MASK_LO	31:0	Current driver defined bin category. Default=0xffffffff

CP_BIN_MASK_HI (R/W)
Upper 32-bits of Bin Mask [PFP]

Field Name	Bit(s)	Description
BIN_MASK_HI	31:0	Current driver defined bin category. Default=0xffffffff

CP_BIN_SELECT_LO (R/W)
Lower 32-bits of Bin Select [PFP]

Field Name	Bit(s)	Description
BIN_SELECT_LO	31:0	Bin category of subsequent command stream data. Default=0xffffffff

CP_BIN_SELECT_HI (R/W)
Upper 32-bits of Bin Select [PFP]

Field Name	Bit(s)	Description
BIN_SELECT_HI	31:0	Bin category of subsequent command stream data. Default=0xffffffff

CP_RT_BIN_MASK_LO (R/W)
Lower 32-bits of Real-Time Bin Mask [PFP]

Field Name	Bit(s)	Description
RT_BIN_MASK_LO	31:0	Current driver defined bin category. Default=0xffffffff

CP_ RT_BIN_MASK_HI (R/W)
Upper 32-bits of Real-Time Bin Mask [PFP]

Field Name	Bit(s)	Description
RT_BIN_MASK_HI	31:0	Current driver defined bin category. Default=0xffffffff

CP_ RT_BIN_SELECT_LO (R/W)
Lower 32-bits of Real-Time Bin Select [PFP]

Field Name	Bit(s)	Description
RT_BIN_SELECT_LO	31:0	Bin category of subsequent command stream data. Default=0xffffffff

CP_ RT_BIN_SELECT_HI (R/W)
Upper 32-bits of Real-Time Bin Select [PFP]

Field Name	Bit(s)	Description
RT_BIN_SELECT_HI	31:0	Bin category of subsequent command stream data. Default=0xffffffff

7.26 2D Control Registers (Updated: 09-26-2002)

The following registers contain information that the CP uses to process the 2D PM4 packets.

CP_IB2D_BASE
2D Indirect Buffer Base [PFP]

Field Name	Bit(s)	Description
IB2D_BASE	31:5	2D Indirect Buffer Base Address. Double-Octword-Aligned.

CP_IB2D_BUFSZ
2D Indirect Buffer Size [PFP]

Field Name	Bit(s)	Description
IB2D_BUFSZ	19:0	2D Indirect Buffer Size in DWORDs. Writing to this register does not initiate the 2D indirect buffer fetch. The CP fetches the 2D indirect buffer when it encounters a 3D-to-2D mode switch.

CP_DEFAULT_PITCH_OFFSET (WO)
2D Default Pitch Offset [ME]

Field Name	Bit(s)	Description
DEFAULT_PITCH_OFFSET	31:0	2D Pitch and Offset Default. Written to ME's Scratch[0] See the PM4 Specification for the format of this register. Value can be read via the ME_STATMUX.

CP_DEFAULT2_PITCH_OFFSET (WO)
2D Default #2 Destination Pitch Offset [ME]

Field Name	Bit(s)	Description
DEFAULT2_PITCH_OFFSET	31:0	2D Pitch and Offset Default #2. Written to ME's Scratch[1] See the PM4 Specification for the format of this register. Value can be read via the ME_STATMUX.

CP_DEFAULT_SC_BOTTOM_RIGHT (WO)
2D Default Clipping Extents [ME]

Field Name	Bit(s)	Description
DEFAULT_SC_BOTTOM_RIGHT	31:0	2D Clipping Extent Default. Written to ME's Scratch[2] See the PM4 Specification for the format of this register. Value can be read via the ME_STATMUX.

CP_DEFAULT2_SC_BOTTOM_RIGHT (WO)
2D Default #2 Clipping Extents [ME]

Field Name	Bit(s)	Description
DEFAULT2_SC_BOTTOM_RIGHT	31:0	2D Clipping Extent Defaults #2. Written to ME's Scratch[3] See the PM4 Specification for the format of this register. Value can be read via the ME_STATMUX.

CP_2D_BRUSH_BASE (WO)
2D External Brush Base Address [ME]

Field Name	Bit(s)	Description
BRUSH_BASE	31:15	Brush Base Address in External Memory. Written to ME's Scratch[4] Value can be read via the ME_STATMUX.

CP_2D_PALETTE_BASE (WO)
2D External Palette Base Address [ME]

Field Name	Bit(s)	Description
PALETTE_BASE	31:15	Palette Base Address in External Memory. Written to ME's Scratch[5] Value can be read via the ME_STATMUX.

CP_2D_IMMD_BASE (WO)
2D External Immediate Data Base Address [ME]

Field Name	Bit(s)	Description
IMMD_BASE	31:19	Immediate Data Base Address in External Memory. Written to ME's Scratch[6] Value can be read via the ME_STATMUX.

CP_2D_BOOLEANS (WO)
2D Booleans [ME]
(Written to ME's Scratch[7])
Value can be read via the ME_STATMUX.

Field Name	Bit(s)	Description
B31_TO_B19	31:19	Written by Driver if anything other than zero is needed.
B18	18	AA Font Destination.
B17	17	AA Font.
B16	16	Source Alpha Blend.
B15	15	Constant Alpha Blend.
B14	14	Alpha Blend.
B13	13	Gradient Fill.
B12	12	Line. Set if the primitive is a line.
B11	11	Destination Rotate. Set if rotation is on.
B10	10	Source Rotate. Set on screen-to-screen blts if rotation is on.
B9	9	Embedded Source. Set for HostBLTs or TextBlts packets when data is viewed as a 1D packed array.
B8	8	Hilite. Set when the Color Compare function is Hilite in the CLR_CMP_CNTL register for the Trans_BitBLT packet.
B7	7	LUT8. Set when the source needs an 8-to-32 bit lookup.
B6	6	LUT. Set when the source needs some kind of lookup/gamma correction.
B5	5	Brush Present. Set when a (non-solid) brush is used.
B4	4	Source Fetch. Set when some kind of BLT or Text operation is being done.
B3	3	Source Mask. Set when the source is monochrome and 0 means don't draw.
B2	2	Brush Mask. Set when the brush is monochrome and 0 means don't draw.
B1	1	Monochrome Brush. Set when the brush is monochrome.
B0	0	Solid Brush.

7.27 DMA Control Registers (Updated: 05-09-2002)

CP_DMA_SRC_ADDR
DMA Engine Source Address [DMA]

Field Name	Bit(s)	Description
SRC_ADDR	31:0	Memory Address or Register-space Address where Source data begins, for the currently active descriptor that is being processed by the DMA engine. Written by the Non-RT Micro Engine.

CP_DMA_DST_ADDR
DMA Engine Destination Address [DMA]

Field Name	Bit(s)	Description
DST_ADDR	31:0	Memory Address or Register-space Address where Destination data begins, for the currently active descriptor that is being processed by the DMA engine. Written by the Non-RT Micro Engine.

CP_DMA_COMMAND (Initiator)
DMA Engine Command [DMA]

Field Name	Bit(s)	Description
COMMAND	31:0	Command data portion of descriptor. ***This register is an initiator for the DMA.*** The source and destination addresses used will be those written to the CP_DMA_SRC_ADDR and CP_DMA_DST_ADDR registers. Written by the Non-RT Micro Engine.

CP_RT_DMA_SRC_ADDR
Real-Time DMA Engine Source Address [DMA]

Field Name	Bit(s)	Description
SRC_ADDR	31:0	Memory Address or Register-space Address where Source data begins, for the currently active descriptor that is being processed by the DMA engine. Written by the Real-Time Micro Engine.

CP_RT_DMA_DST_ADDR
Real-Time DMA Engine Destination Address [DMA]

Field Name	Bit(s)	Description
DST_ADDR	31:0	Memory Address or Register-space Address where Destination data begins, for the currently active descriptor that is being processed by the DMA engine. Written by the Real-Time Micro Engine.

CP_RT_DMA_COMMAND (Initiator)
Real-Time DMA Engine Command [DMA]

Field Name	Bit(s)	Description
COMMAND	31:0	Command data portion of descriptor. ***This register is an initiator for the DMA.*** The source and destination addresses used will be those written to the CP_RT_DMA_SRC_ADDR and CP_RT_DMA_DST_ADDR registers. Written by the Real-Time Micro Engine.

CP_DMA_TABLE_ADDR (WO) / (Initiator)
DMA Engine Descriptor Table Address [DMA]

Field Name	Bit(s)	Description
TABLE_ADDR	31:4	Memory Address where the Descriptor Table begins. This register location is actually a port into the Descriptor Table Address Queue (DTAQ) FIFO of the DMA engine. Up to 8 pending table addresses can be stored in the DTAQ queue. ***This is a trigger register, which starts the DMA engine.*.**
Unused	3:1	Unused
CP_SYNC	0	Synchronize with the ***Non-Real-Time*** Micro Engine. Indicates that the Micro Engine cannot proceed to write anything to the register backbone while this DMA is running.

CP_DMA_SRC_ADDR_STAT (RO)
DMA Engine Source Address [DMA]

Field Name	Bit(s)	Description
SRC_ADDR	31:0	Memory Address or Register-space Address where Source data begins, for the currently active descriptor that is being processed by the DMA engine. Read Only.

CP_DMA_DST_ADDR_STAT (RO)
DMA Engine Destination Address [DMA]

Field Name	Bit(s)	Description
DST_ADDR	31:0	Memory Address or Register-space Address where Source data is being written, for the currently active descriptor that is being processed by the DMA engine. Read Only.

CP_DMA_COMMAND_STAT (RO)
DMA Engine Command (for Current Descriptor) [DMA]

Field Name	Bit(s)	Description
EOL	31	End Of List. Indicates that the currently active descriptor is the last one in the Descriptor Table.
INTDIS	30	Interrupt Disable. This value is a don't care if the EOL bit is '0'. 1=Disable the EndOfList interrupt. 0=Don't disable interrupt.
DAIC	29	Destination Address Increment Control. 0 = Increment the internal Destination Address with each data transfer. 1 = No increment.
SAIC	28	Source Address Increment Control. 0 = Increment the internal Source Address with each data transfer. 1 = No increment.
DAS	27	Destination Address Space. 0 = Destination Address is a memory-space address 1 = Destination Address is a register-space address
SAS	26	Source Address Space. 0 = Source Address is a memory-space address 1 = Source Address is a register-space address
DST_SWAP	25:24	Destination Endian Swap Control. 0 = No swap 1 = 16-bit swap: 0xAABBCCDD becomes 0xBBAADDCC 2 = 32-bit swap: 0xAABBCCDD becomes 0xDDCCBBAA 3 = Half-dword swap: 0xAABBCCDD becomes 0xCCDDAABB
SRC_SWAP	23:22	Source Endian Swap Control. 0 = No swap 1 = 16-bit swap: 0xAABBCCDD becomes 0xBBAADDCC 2 = 32-bit swap: 0xAABBCCDD becomes 0xDDCCBBAA 3 = Half-dword swap: 0xAABBCCDD becomes 0xCCDDAABB
QUEUED	21	Queued RBBM Path – Affects how DMA transactions through the RBBM are routed. 0 = DMA transactions go through the Non-Queued Path 1 = DMA transactions go through the Queued Path and are affected by the wait_until condition.
BYTE_COUNT	20:0	Number of Bytes remaining to be transferred from Source to Destination.

CP_DMA_STAT
DMA Engine Status [DMA]

Field Name	Bit(s)	Description
SWAP	23:22	Endian Swap Control for fetching the Descriptor Table. 0 = No swap (Default) 1 = 16-bit swap: 0xAABBCCDD becomes 0xBBAADDCC 2 = 32-bit swap: 0xAABBCCDD becomes 0xDDCCBBAA 3 = Half-dword swap: 0xAABBCCDD becomes 0xCCDDAABB
ACTIVE	21	Indicates that the DMA engine is currently working on a descriptor table. *The DMA engine remains active until the last write of the last descriptor for the DMA table is past the point of arbitration in the Memory Hub.* It also indicates whether the CURRENT_TABLE_NUM field is a valid entry in the Descriptor Table Address Queue. Read Only.
ABORT_EN	20	1 = Abort the Descriptor Table Address Queue entry indicated by the CURRENT_TABLE_NUM field, and halt operation. 0 = Re-initialize (invalidate all entries in) the Descriptor Table Address Queue, and allow the DMA engine to accept new commands. Note that the programmer must wait for the ACTIVE bit to be '0' before writing a '0' to the ABORT_EN bit; otherwise the DMA engine may hang. Default = 0.
CURRENT_TABLE_NUM	14:12	This is a pointer into the Descriptor Table Address Queue, indicating which queue entry the DMA engine is currently processing. Read Only.
LAST_TABLE_NUM	10:8	This is a pointer into the Descriptor Table Address Queue, indicating which queue entry was the last one to be written. Read Only.
DTAQ_AVAIL	3:0	The number of available entries in the Descriptor Table Address Queue. Read Only.

CP_DMA_ACT_DSCR_STAT (RO)
DMA Engine Active Descriptor Status [DMA]

Field Name	Bit(s)	Description
TABLE_ADDR	31:4	Memory Address of the most recently active descriptor.

7.28 BIOS Scratch Registers (Updated: 05-27-2002)

The BIOS scratch registers have been moved from the VIP to the CP in the CRAYOLA. These are general-purpose R/W registers.

BIOS_0_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_1_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_2_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_3_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_4_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_5_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_6_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_7_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_8_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_9_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_10_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_11_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_12_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_13_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_14_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

BIOS_15_SCRATCH
BIOS Scratch Register [RBIU]

Field Name	Bit(s)	Description
BIOS_SCRATCH	31:0	Scratch register used by the BIOS.

7.29 Coherency Registers (Updated: 08-27-2003)

COHER_BASE_PM4
PM4 Coherency Base Register [COHER]

Field Name	Bit(s)	Description
BASE	31:12	Default=0x0.

COHER_SIZE_PM4
PM4 Coherency Size Register [COHER]

Field Name	Bit(s)	Description
SIZE	31:12	Default=0x0.

COHER_STATUS_PM4
PM4 Coherency Status Register [COHER]

Field Name	Bit(s)	Description
MATCHING_CONTEXTS	7:0	Corresponding bit is set if the Base is used in that context (RO) Default=0x0.
COLOR0_BASE_ENA	8	If enabled, the scan logic will test the written base against all valid context for color 0 bases.
COLOR1_BASE_ENA	9	If enabled, the scan logic will test the written base against all valid context for color 1 bases.
COLOR2_BASE_ENA	10	If enabled, the scan logic will test the written base against all valid context for color 2 bases.
COLOR3_BASE_ENA	11	If enabled, the scan logic will test the written base against all valid context for color 3 bases.
DEPTH_BASE_ENA	12	If enabled, the scan logic will test the written base against all valid context for depth bases.
TILE_BASE_ENA	13	If enabled, the scan logic will test the written base against all valid context for tile bases.
VC_ACTION_ENA	14	If enabled, the Vertex Cache will get a start pulse when the PM4 Base is written.
TC_ACTION_ENA	15	If enabled, the Texture Cache will get a start pulse when the PM4 Base is written.
RC_ACTION_ENA	16	If enabled, the Render Common will get a start pulse when the PM4 Base is written.
STATUS	31	Designates whether the surface is ready to be used.

COHER_BASE_RT
RT Coherency Base Register [COHER]

Field Name	Bit(s)	Description
BASE	31:12	Default=0x0.

COHER_SIZE_RT
RT Coherency Size Register [COHER]

Field Name	Bit(s)	Description
SIZE	31:12	Default=0x0.

COHER_STATUS_RT
RT Coherency Status Register [COHER]

Field Name	Bit(s)	Description
MATCHING_CONTEXTS	7:0	Corresponding bit is set if the Base is used in that context (RO) Default=0x0.
COLOR0_BASE_ENA	8	If enabled, the scan logic will test the written base against all valid context for color 0 bases.
COLOR1_BASE_ENA	9	If enabled, the scan logic will test the written base against all valid context for color 1 bases.
COLOR2_BASE_ENA	10	If enabled, the scan logic will test the written base against all valid context for color 2 bases.
COLOR3_BASE_ENA	11	If enabled, the scan logic will test the written base against all valid context for color 3 bases.
DEPTH_BASE_ENA	12	If enabled, the scan logic will test the written base against all valid context for depth bases.
TILE_BASE_ENA	13	If enabled, the scan logic will test the written base against all valid context for tile bases.
VC_ACTION_ENA	14	If enabled, the Vertex Cache will get a start pulse when the PM4 Base is written.
TC_ACTION_ENA	15	If enabled, the Texture Cache will get a start pulse when the PM4 Base is written.
RC_ACTION_ENA	16	If enabled, the Render Common will get a start pulse when the PM4 Base is written.
STATUS	31	Designates whether the surface is ready to be used.

COHER_BASE_HOST
HOST Coherency Base Register [COHER]

Field Name	Bit(s)	Description
BASE	31:12	Default=0x0.

COHER_SIZE_ HOST
HOST Coherency Size Register [COHER]

Field Name	Bit(s)	Description
SIZE	31:12	Default=0x0.

COHER_STATUS_ HOST
HOST Coherency Status Register [COHER]

Field Name	Bit(s)	Description
MATCHING_CONTEXTS	7:0	Corresponding bit is set if the Base is used in that context (RO) Default=0x0.
COLOR0_BASE_ENA	8	If enabled, the scan logic will test the written base against all valid context for color 0 bases.
COLOR1_BASE_ENA	9	If enabled, the scan logic will test the written base against all valid context for color 1 bases.
COLOR2_BASE_ENA	10	If enabled, the scan logic will test the written base against all valid context for color 2 bases.
COLOR3_BASE_ENA	11	If enabled, the scan logic will test the written base against all valid context for color 3 bases.
DEPTH_BASE_ENA	12	If enabled, the scan logic will test the written base against all valid context for depth bases.
TILE_BASE_ENA	13	If enabled, the scan logic will test the written base against all valid context for tile bases.
VC_ACTION_ENA	14	If enabled, the Vertex Cache will get a start pulse when the PM4 Base is written.
TC_ACTION_ENA	15	If enabled, the Texture Cache will get a start pulse when the PM4 Base is written.
RC_ACTION_ENA	16	If enabled, the Render Common will get a start pulse when the PM4 Base is written.
STATUS	31	Designates whether the surface is ready to be used.

Host Programming Considerations

7.30 Starting the Indirect Streams & Restrictions (Updated: 05-08-2002)

The Indirect Buffers cannot be initiated via direct register writes. Only the Pre-Fetch Parser initiates indirect buffers.

The PFP writes to the CP_IB1_BUFSZ register to initiate fetching an indirect buffer from the Ring Buffer. The CP will fetch commands, starting at the address in the CP_IB1_BASE register, and continue until the CP_IB1_BUFSZ amount is exhausted. Then it will switch back to the Ring Buffer.

The PFP writes to the CP_IB2_BUFSZ register to initiate fetching an indirect buffer from the Indirect #1. The CP will fetch commands, starting at the address in the CP_IB2_BASE register, and continue until the CP_IB2_BUFSZ amount is exhausted. Then it will switch back to Indirect Buffer #1.

Here are the valid ways for the Driver to initiate indirect buffers:

1. Type-0 Packets – Must be a single Type-0 packet where the BASE_INDEX in the header is either the CP_IB1_BASE or CP_IB2_BASE register and it also writes the buffer size register. Note that initialization of an indirect buffer cannot be part of a larger Type-0 packet that writes other memory-mapped registers. Nor can it be two single-DWORD Type-0 packets. It is also assumed that the Driver will write both the base and size registers for every initiation.
2. Indirect_Buffer Packet – This is the preferred mode in CRAYOLA. The Indirect Buffer packet includes a "multi-pass" control flag that allows the indirect buffer to be fetched/executed more than once.

A Type-1 packet cannot be used to initiate an indirect buffer.

7.31 Writing to the Micro Engine RAM (Updated: 01-30-2003)

In order to change a location in the Micro Engine RAM (i.e. micro code), first load the CP_ME_RAM_WADDR Register with the address of the RAM into which data is to be written. Then write all the data to the CP_ME_RAM_DATA port. The RAM's write address auto increments after each write to the data port. Note that the number of DWORDs written must be a multiple of the width of a micro instruction – 3 DWORDs.

7.32 Reading from the Micro Engine RAM (Updated: 06-24-2002)

In order to read a location in the Micro Engine RAM, first load the CP_ME_RAM_RADDR Register with the address of the RAM from which data is to be read. This triggers the Command Processor to read the microcode value at that RAM location and transfer it to an internal holding register. Also, the RAM Read Address Register is auto-incremented to point to the next location in the RAM. Then read the CP_ME_RAM_DATA register. Consecutive reads from this register will provide the data for successive RAM locations. If the last RAM location is read, the address will wrap back to zero.

7.33 Writing to & reading from the Pre-fetch Parser's microcode RAM

There is a set of address & data registers for both the real-time & non-real-time PFP – CP_PFP_(RT_)UCODE_ADDR|DATA – for writing into and reading from the microcode RAMs. The HALT bit in CP_ME_CNTL needs be set any time these registers are being updated.

A write to the Address registers sets the 'address' for subsequent data writes into the RAM. Each write of the data (microcode) will cause the current address to be incremented to the next location in the RAM. A read will return the current address.

Reading the Data register returns the data (microcode) at the location pointed to by the current instruction pointer internal to the RISC processor. This value can be found by reading the PFPs debug bus. The instruction pointer cannot be set to a specific value, so the entire RAM may need to be read to get the contents of a specific location. Each read of the data will cause the instruction pointer to increment to the next location in the RAM and should therefore be used only in Debug mode.

7.34 Registers Requiring CP Idling Before Writing (Update: 07-01-2003)

Writing certain registers in the Command Processor (CP) cause internal resetting to its control logic. The Host needs to ensure that the CP is idle before writing these registers. The following registers fall under this category:

1. CP_RB_CNTL
2. CP_RB_BASE
3. CP_RB_RPTR_WR
4. CP_RB_WPTR_DELAY
5. CP_QUEUE_THRESHOLDS
6. CP_MEQ_THRESHOLDS
7. CP_DEBUG

8. External Interfaces

The Command Processor actually *does* communicate with other modules of the Graphics Controller device.

8.1 System Interface (Completed: 08-28-2002)

Pin Name	Vector	Type	Description	Notes
RBBM_regclk_active		I	Clock Gate for Register Bus.	RBBM asserts this signal for a pending transaction. The CP uses this signal to enable its gated clock for the logic on the register bus.
CP_MH_memreq		O	CP has a pending request for the Memory Hub.	The CP asserts this signal when it has a read/write request to the Memory Hub. It will wait until the MH_CP_memclk_active signal is asserted before issuing a request to the MH.
MH_CP_memclk_active		I	Clock Gate for Memory Bus	The Memory Hub asserts this signal to enable the clock to the flops connected to the memory return bus. The CP uses this signal to enable its gated clock. It waits for this signal to be active before issuing requests to the Memory Hub.
sclk		I	Permanent Core Clock	RTEE Uses for Non-Polling Logic (~190 flops) RBIU Uses for Read Return Data (66 Flops)
srst		I	Hard Reset – Resets all control logic and resets registers to their default values.	Reset (Synchronized to SCLK)
CG_CP_pm_enb		I	Power Management Enable	Active Low

8.2 CP -to- RBBM Interface (Updated: 08-28-2002)

The CP uses this interface to write to remote registers, which may reside in other functional blocks in the top-level chip. The same interface is used to transfer both Real-Time, Pre-Fetch Parser, and Non-Real Time Data to the RBBM.

Pin Name	Vector	Type	Description
CP_RBBM_send		O	CP Send to RBBM
CP_RBBM_rt_send		O	CP Send Real-Time Stream Transaction
CP_RBBM_pf_send		O	CP Send Pre-Fetch Parser Transaction
CP_RBBM_a	16:2	O	Register Address for Read/Write transaction.
CP_RBBM_op		O	Opcode. Specifies a read or write transaction. (0=Read, 1=Write).
CP_RBBM_nq		O	Use Non-Queued Path 0 = Transaction to go through "queued" path of RBBM. 1 = Transaction to go through "non-queued" path of RBBM.
CP_RBBM_wd	31:0	O	DWORD Write Data Bus
CP_RBBM_be	3:0	O	Byte Mask (Valid for 32-bit Transfers). Note: The byte enables are only provided for DMA operations. The Micro Engine does not support byte write operations.
RBBM_CP_rdy		I	Non-Real-Time Ready to Receive
RBBM_CP_rt_rdy		I	Real-Time Ready to Receive.
RBBM_CP_pf_rdy		I	Pre-Fetch Parser Ready to Receive (Writes Only).
RBBM_CP_valid		I	RBBM read data valid. Note: Only one read transaction can be active at any given time. CP should sample the read-return data (oRBBM_RDO) the first clock that the valid is high on a read transaction.
RBBM_rd	31:0	I	Read Return Data.
RBBM_CP_soft_reset		I	Soft Reset – Resets all control logic, but does not reset the register state. Control registers retain their values.

8.3 RBBM -to- CP Interface (RBIU) (Updated: 04-25-2002)

This interface is used when the Host wishes to access registers in the CP.

The CP's input FIFO generates the CP_RBBM_nrtrtr signal. The FIFO is deep enough so that up to 3 additional repeater flops can be inserted between the RBBM and CP without modifying the CP.

The CP does not use the byte enables, so these do not appear as inputs.

Pin Name	Vector	Type	Description	Note
RBBM_a	16:2	I	Register Address for Read/Write Transactions.	
RBBM_we		I	Write Enable (Address and Data are Valid)	
RBBM_wd	31:0	I	Write Data Bus.	
CP_RBBM_nrtrtr		O	Non-Real Time Ready to Receive	Up to 3 repeater flops can be added without modifying the CP
RBBM_re		I	Read Enable (Address is Valid, Data is "Don't Care"	
RBB_rsI		I	Read Return Strobe from Read Daisy Chain	
RBB_rdI	31:0	I	Read Return Data from Read Daisy Chain	
RBB_rsO		O	Output Read Return Strobe to Read Daisy Chain	
RBB_rdO	31:0	O	Output Read Return Data to Read Daisy Chain	
RBBM_out_a	16:2	O	Address output to be used for a repeater.	
RBBM_out_wd	31:0	O	Data output to be used for a repeater.	
RBBM_out_we		O	Write enable output to be used for a repeater.	
RBBM_out_re		O	Read enable output to be used for a repeater.	

8.4 CP –to- Memory Hub Interface (Updated: 03-26-2002)

8.4.1 Description

The CP uses this interface to read and write both the system and video (local) memories. The read return data can return out-of-order from how it was requested. The CP has re-ordering logic to handle out-of-order data returns.

The CP reads the following through this interface:

1. Ring Buffer Data
2. Indirect Buffer #1 and #2 Data
3. Real-Time Stream Data
4. Micro Engine Wait Semaphores (WAIT_MEM)
5. Micro Engine Code
6. State Sub-Block, Constant, and Shader Instruction Data
7. DMA Reads from Memory

The CP writes the following data through this interface:

1. Semaphores for Synchronization (Ring Buffer Read Pointer, Scratch Registers, etc)
2. Micro Engine Semaphore Writes (MEM_WRITE)
3. Constant State Data (From SET_CONSTANT PM4 Packet)
4. 2D Immediate Data (Brush, etc)
5. DMA Writes to Memory

Pin Name	Vector	Type	Description	Note
CP_MH_send		O	Send Active for Duration of Transfer (Address and Data Phases).	
CP_MH_write		O	Write Enable: 0 => Read, 1=> Write. Active only during address phase of the transfer.	
CP_MH_ad	31:0	O	Read Address & Command / Write Data For Address & Command: [31:4] – Octword Address of Request [3] -- Memory/Register 0 – Memory 1 – Register (Not Used by CP) [2] -- Transfer Size 0 – 128-bit Transfer (Not Used by CP) 1 – 256-bit Transfer [1:0] – Endian 00 : No Swap 01 : 16-bit Swap - 0xAABBCCDD → 0xBBAADDCC 10 : 32-bit Swap - 0xAABBCCDD → 0xDDCCBBAA 11 : WORD Swap - 0xAABBCCDD → 0xCCDDAABB	
CP_MH_tagbe	7:0	O	Read Tag / Write Byte Enable (BE) / Write Confirm (WC) For read operations the 8-bit bus contains tag information that is returned with the return data. The CP puts information in tag to identify where to store the return data. For write operations, the write byte enable (BE) is the lower 4 bits (3:0). The byte enables can change per clock within a 256-bit write in order to support DWORD writes. The CP's MIU however will group write requests if these fall into the same Double Octword. The most-significant bit (bit 7) of the tag is the "write confirm"	

			(WC) flag that indicates to the MH that a confirmation of the write is needed. The MH will de-assert the MH_CP_writeclean signal when the write transaction has been written to memory.	
CP_MH_phase		O	Phase: 0 = Data Phase 1 = Address & Control Phase	
MH_CP_rtr		I	Ready to Receive Requests. *This is guaranteed to be at least one request. The CP will at least register the RTRn before using this signal in order to improve timing.*	
CP_MH_rcd		O	Read Combine Disable The CP asserts this signal for all Ring Buffer read requests (i.e. Reads from AGP). It forces the MH not to combine the read requests so the CP does not get bad data from prior read operations.	
MH_CP_grb_send		I	Identifies CP as the Client to Receive Data	
MH_grb_data	31:0	I	Read Data.	
MH_grb_tag	7:0	I	Tag Returned. This is the same information that was sent on the CP_MH_tagbe interface when the read request was issued.	Wired to lower 8 bits of a global 16-bit bus.
MH_CP_writeclean		I	Write Clean from Memory Hub (MH). The MH will assert this signal when it receives a write transaction with the "write confirm" (WC) flag set in the tag. The MH will de-assert the MH_CP_writeclean signal when that write transaction is written to memory. The CP will stall consecutive write transactions with the WC flag set until the MH_CP_writeclean is de-asserted. Note that read operations and write operations without the WC flag set will be processed regardless of the status of the MH_CP_writeclean signal. This flag is used for 2D operations to confirm that Brush Data, Palette, and Immediate data has been written to memory. It is also used to confirm that Constant data is also written to memory.	
grb_out_data	31:0	O	Repeated Data for use as a Repeater.	
grb_out_tag	7:0	O	Repeated Tag for use as a Repeater.	

8.5 Renderer Common Interface (Updated 06-21-2002)

The Renderer Common (RC) sends the following signals to the CP for synchronization purposes.

Pin Name	Vector	Type	Description	Note
RC_CP_cache_flush		I	Pulse from Renderer Common (RC) to indicate that the cache flush operation is done. The CP will issue a DWORD write to memory as a result from the "cache flush done" event logic. ***The RC generates this signal as a result of receiving a CACHE_FLUSH_TS event initiator.***	
RC_CP_context_done		I	Pulse indicates that the RC is completed with a state context. The CP uses this to de-allocate an assigned context. The CP stalls until its assigned context is available. ***The RC generates as a result of receiving a CONTEXT_DONE event initiator.***	RC de-allocates state in-order.

8.6 Coherency Interface (Updated: 04-10-2002)

Pin Name	Vector	Type	Description	Note
CP_RBBM_coherency_busy		O	Coherency logic busy signal that asserts when there is a base match and the matching context is still valid.	
TC_START0		O	PM4 start signal to the TC to clean its cache of the written base.	
TC_START1		O	RT start signal to the TC to clean its cache of the written base.	
TC_START2		O	Host start signal to the TC to clean its cache of the written base.	
RC_START0		O	PM4 start signal to the RC to clean its cache of the written base.	
RC_START1		O	RT start signal to the RC to clean its cache of the written base.	
RC_START2		O	Host start signal to the RC to clean its cache of the written base.	
VC_START0		O	PM4 start signal to the VC to clean its cache of the written base.	
VC_START1		O	RT start signal to the VC to clean its cache of the written base.	
VC_START2		O	Host start signal to the VC to clean its cache of the written base.	
TC_CLEAN0		I	PM4 clean signal from the TC that its caches are clean of the written base.	
TC_ CLEAN1		I	RT clean signal from the TC that its caches are clean of the written base.	
TC_ CLEAN2		I	Host clean signal from the TC that its caches are clean of the written base.	
RC_ CLEAN0		I	PM4 clean signal from the RC that its caches are clean of the written base.	
RC_ CLEAN1		I	RT clean signal from the RC that its caches are clean of the written base.	
RC_ CLEAN2		I	Host clean signal from the RC that its caches are clean of the written base.	
VC_ CLEAN0		I	PM4 clean signal from the VC that its caches are clean of the written base.	
VC_ CLEAN1		I	RT clean signal from the VC that its caches are clean of the written base.	
VC_ CLEAN2		I	Host clean signal from the VC that its caches are clean of the written base.	

8.7 Scan Converter Interface

8.7.1 "Viz Query A" Interface (Updated: 03-22-2002)

The Scan Converter returns the "Discard" flags back to the CP through this interface.

Pin Name	Vector	Type	Description	Note
SC_CP_vq_snd		I	SC is returning the visibility flag now.	
SC_CP_vq_index	5:0	I	Index to identify one of 64 VQ flags stored in the CP.	

SC_CP_vq_discard		I	0=Keep, 1=Discard	

8.7.2 Multipass Flow Control (Updated: 03-22-2002)

The Scan Converter tells the CP to loop or continue through this interface.

Pin Name	Vector	Type	Description	Note
SC_CP_mp_snd		I	SC is sending the multi-pass command now.	
SC_CP_mp_loop		I	0=Continue, 1=Loop	

8.7.3 Write Confirm for Embedded Data (Updated: 07-20-2002)

The CP tells the Scan Converter (SC) that the write associated with 2D embedded immediate data is completed (i.e. confirmed by the Memory Hub). The SC uses this signal to increment a counter that it tests when waiting for a wait confirm event – sent by the CP. In turn, the SC sends back a decrement signal to the CP to indicate that it has processed the write confirmation.

The CP will stall processing packets that write to memory if its write confirm counter is at its maximum value.

Pin Name	Vector	Type	Description	Note
CP_SC_wc_inc		O	Confirmation has been received by the Memory Hub for 2D immediate data in the command stream.	
SC_CP_wc_dec		I	Confirmation has been processed by the Scan Converter.	

8.8 Status Interface (Updated: 08-05-2002)

The CP receives and sends status signals from/to other parts of the chip.

Signal Name	Vector	Type	Description	Note
CP_rts_discretes	31:0	I	Discrete Signals for Real-Time Stream Initiation.	See CP_RTS_Connections.doc
DISP1_CP_eol		I	End of Line from Display Engine #1 (Pulse)	Used in RTEE.
DISP2_CP_eol		I	End of Line from Display Engine #2 (Pulse)	Used in RTEE.
DISP1_CP_sof		I	Start of Frame from Display Engine #1 (Pulse)	Used in RTEE.
DISP2_CP_sof		I	Start of Frame from Display Engine #2 (Pulse)	Used in RTEE.
VIP_CP_eol		I	End of Line from VIP (Pulse)	Used in RTEE.
VIP_CP_eof		I	End of Frame from VIP (Pulse)	Used in RTEE.
RBBM_CP_nrt_idle		I	Non-Real-Time Processing is Done	Used to clear the context valid bits for surface coherency.
RBBM_CP_rt_idle		I	Real-Time Processing is Done	Used to clear context #0 valid bit for surface coherency if real-time is enabled.
CP_RBBM_int		O	Command Processor Interrupt.	
CP_RBBM_nrt_busy		O	Command Processor is Processing Non-Real-Time. This is the CP_NRT_BUSY signal in the CP_STAT with the addition of including any pending register transactions in the CP.	
CP_RBBM_rt_busy		O	Command Processor is Processing a Real-Time stream.	Does not Include the RTEE Polling Operations.
CP_RBBM_dma_busy		O	Command Processor's DMA Engine is Busy.	
CP_RBBM_rt_enable		O	Real-Time Streams are Enabled.	

8.9 Sequencer-to-CP Interface (Updated: 10-11-2002)

The invalidation of Vertex and Pixel shader code from the Sequencer is indicated over these interfaces. The Sequencer issues a pulse on the "event" signal and asserts the "eventid" signal to indicate the type of event. The encoding of the "eventid" is listed in the table below.

The Micro Engine (ME) will de-allocate the corresponding shader memory section when it receives a "Deallocate" event.

Signal Name	Vector	Type	Description
SQ_CP_vs_event		I	Vertex Shader Event
SQ_CP_vs_eventid	4:0	I	Vertex Shader Event ID 0x00 ➔ VsDeallocate (Result of a VS_DEALLOC Event Initiator) 0x02 ➔ VsDoneEvent (Result of a VS_DONE_TS Event Initiator) ** All other Event IDs are Ignored **
SQ_CP_ps_event		I	Pixel Shader Event
SQ_CP_ps_eventid	4:0	I	Pixel Shader Event ID 0x01 ➔ PsDeallocate (Result of a PS_DEALLOC Event Initiator) 0x03 ➔ PsDoneEvent (Result of a PS_DONE_TS Event Initiator) ** All other Event IDs are Ignored **

8.10 Debug Bus Interface (Updated: 02-03-2003)

This following interface is to connect to the debug bus.

Signal Name	Vector	Type	Description	Note
DEBUG_bus_in	11:0	I	Debug Bus Input	Async. Input
DEBUG_bus_out	11:0	O	Debug Bus Output	Async. Input
DEBUG_block_sel	5:0	I	Unit Select	CP = 0x03
DEBUG_group_sel	5:0	I	Selects Local Signals to Output	See the CP's Test Bus Specification.

9. Command Processor Implementation

9.1 Command Stream Fetcher (CSF) {Updated 05-01-2002}

The CSF for the CRAYOLA consists of five command stream fetcher(s):

1. Ring Buffer
2. Indirect Buffers in Ring Buffer
3. Indirect Buffers in Indirect #1
4. Non-Real-Time State Data Fetcher – State data for Ring, Indirect #1, and Indirect #2 Buffers
5. Real-Time State Data Fetcher

Also in the CSF is the Ring Buffer Manager (RBM) logic. The RBM's responsibility is to track the read and write pointers to the Ring Buffer, and to calculate if there is any data present in the Ring Buffer. It also generates DWORD write requests to the Memory Interface Unit (MIU) in order to update the Host's copy of the Read Pointer to the Ring Buffer.

The fetchers keep track of the available space in the return data queues and will not issue a request to the Memory Hub if there is no space for the return data.

For Micro Engine can also issue read requests to the Memory Hub. These requests do not go through the CSF and the return data is sent directly to the Micro Engine.

For Real Time Streams, the CP has a separate data/command cache. The reason is to prevent the Ring and Indirect Buffer data from blocking the RTS data. Real Time streams are fetched based on output from an Event Engine that monitors real-time events.

Figure 9-2 shows a top-level block diagram of the Command Stream Fetcher (CSF).

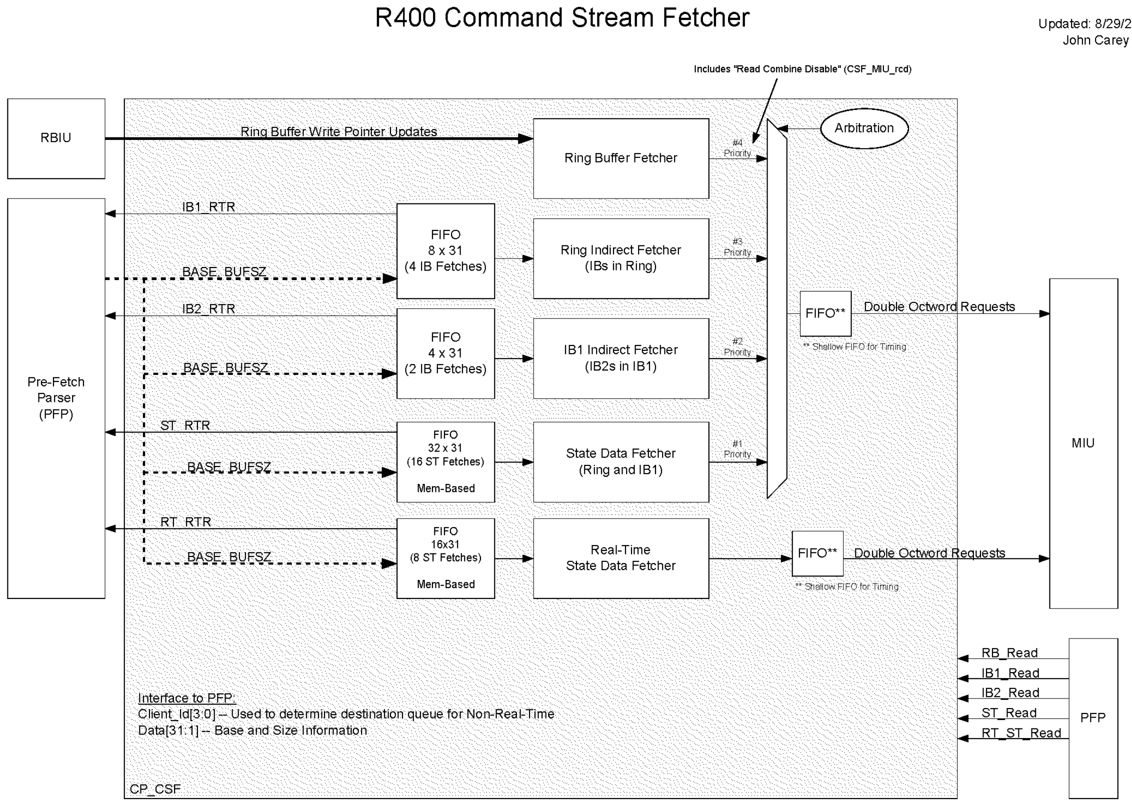


Figure 9-1: Block Diagram of Command Stream Fetcher

9.2 Real-Time Event Engine (RTEE)

9.2.1 General Description of Real-Time Streams (RTS) (Updated: 02-01-2002)

Support of Real-Time Streams is added to the CP for CRAYOLA. A Real-Time Stream (RTS) is defined as a high priority / low latency command stream that interrupts the non-RTS command processing. One typical use is Video Overlay Scaling. All of the CP micro-engine functionality is available and there is dedicated WAIT_UNTIL logic in the RBBM for Real-Time streams.

The data for the RTS bypasses all the setup and is written directly to the Scan Converter via a dedicated path through the RBBM. The SC is interrupted to process the RTS data. The RTS can also include state data that is written to state set zero which is dedicated for real-time stream processing.

For Real Time Streams, the CP has a separate data/command cache from the Ring and Indirect Buffers and an independent path through the Micro Engine (ME). A separate path is also provided through the RBBM to the Global Register Bus (GRB). The reason is to prevent the Ring and Indirect Buffer streams from blocking the RTS data. Note that an independent Micro Engine for Real-Time Streams is not required. Rather, the only requirement is to have a stoppable non-RTS pipeline through the ME.

Real Time Streams are fetched based on output from an Event Engine (RTEE) that monitors real-time events. The RTEE is armed via register writes. Up to 16 independent events are supported.

When an event is triggered the CP will arbitrate that stream's request against other pending Real-Time streams. When the RT stream wins the arbitration, the stream will be fetched, parsed by the PFP and Micro Engine, and the resulting commands will be sent to the rest of the graphics pipeline.

Note that even while a real-time stream is active, the RTEE is still in operation. Multiple events may trigger at any time. The CP determines the RTS to activate at the time the current RTS completes. The determination is a fixed priority where RTS #0 is the highest and RTS #15 is the lowest. So, the order that the events trigger does not determine the order of processing. The highest-priority RTS that has triggered at the time the fetch is ready is the one that will be processed. The non-RTS command stream will only re-start if no Real-Time Streams are active.

Real-Time streams are set-up via control registers per stream in the CP. These are defined elsewhere in this specification. It is also possible for an active RT stream to re-program its control registers.

The triggers for the Event Engine can be either unary signals (conditioned in the RTEE) or status that the CP reads (polls) from a location elsewhere in the chip or external memory. If polling is desired, then the real-time stream has the choice of three different time interval values. These are selected in the CP_RT*_COMMAND register. The time interval values are programmed in the CP_RT_POLL_INTERVAL register. Note that the interval counters are free-running and if multiple RT streams select the same counter, then all will trigger at the same time.

The RT stream automatically disarms itself once activated. It is the responsibility of the stream to re-arm itself if needed.

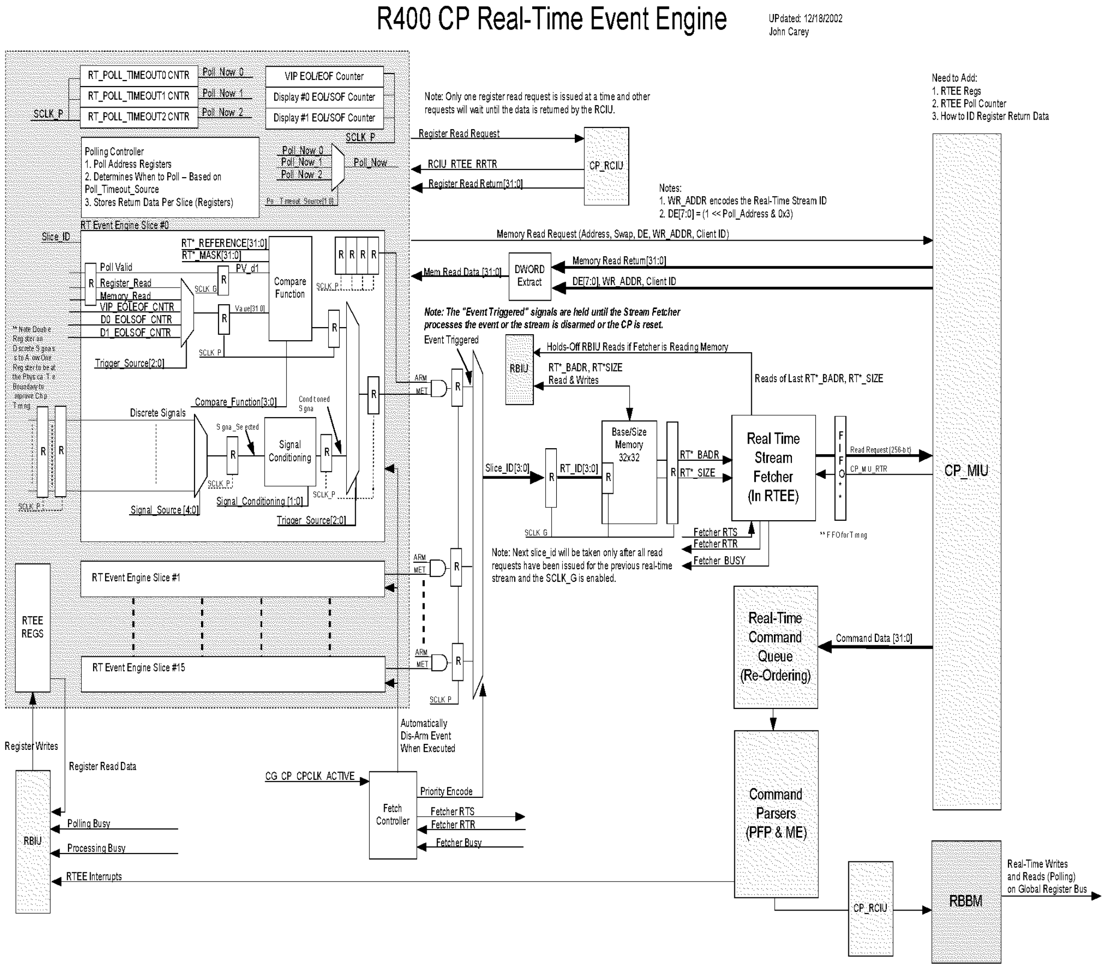
There are certain restrictions for PM4 command packets with a Real-Time stream. See the "Specification of PM4 Command Packets in CRAYOLA" document for details.

A typical Real-Time Stream would contain the following:

> WAIT_RTS_IDLE – RBBM holds-off RTS writes until prior RTS is completed in the RBBM.
>
> User's Real-Time Code:
>
>> Set_State (or Type-0 Packets)
>>
>> Set_Constant(s) (or Load_Constant_Context)
>>
>> Im_Load – Used to load Real-Time instruction code
>>
>> Type-0 Packet(s) (Primitive)

Note that it is the responsibility of the real-time Drivers to insert the WAIT_RTS_IDLE at the beginning of the RT command stream.

The following figure is a top-level diagram of the Real-Time Event Engine (RTEE).



R400 CP Real-Time Event Engine
UPdated: 12/18/2002
John Carey
RT_POLL_TIMEOUT0 CNTR
RT_POLL_TIMEOUT1 CNTR
RT_POLL_TIMEOUT2 CNTR
Poll Now 0
Poll Now 1
Poll Now 2
SCLK_P
VIP EOL/EOF Counter
Display #0 EOL/SOF Counter
Display #1 EOL/SOF Counter
Polling Controller
1. Poll Address Registers
2. Determines When to Poll – Based on Poll_Timeout_Source
3. Stores Return Data Per Slice (Registers)
Poll_Now
Poll_Timeout_Source[1:0]
RT Event Engine Slice #0
Slice_ID
RT*_REFERENCE[31:0]
RT*_MASK[31:0]
PV_d1
Poll Valid
Register_Read
Memory_Read
VIP_EOLEOF_CNTR
D0_EOLSOF_CNTR
D1_EOLSOF_CNTR
SCLK_G
Value[31:0]
Compare Function
R
Trigger_Source[2:0]
** Note Double Register on Discrete Signals to Allow One Register to be at the Physical Tile Boundary to improve Chip Timing
Compare_Function[3:0]
Discrete Signals
Conditioned Signal
Signal_Selected
Signal Conditioning
Signal_Source[4:0]
Signal_Conditioning [1:0]
RT Event Engine Slice #1
RT Event Engine Slice #15
RTEE REGS
ARM
MET
Note: Only one register read request is issued at a time and other requests will wait until the data is returned by the RCIU.
Register Read Request
RCIU_RTEE_RRTR
Register Read Return[31:0]
CP_RCIU
Notes:
1. WR_ADDR encodes the Real-Time Stream ID
2. DE[7:0] = (1 << Poll_Address & 0x3)
Memory Read Request (Address, Swap, DE, WR_ADDR, Client ID)
Memory Read Return[31:0]
DE[7:0], WR_ADDR, Client ID
Mem Read Data [31:0]
DWORD Extract
Note: The "Event Triggered" signals are held until the Stream Fetcher processes the event or the stream is disarmed or the CP is reset.
Event Triggered
RBIU
Holds-Off RBIU Reads if Fetcher is Reading Memory
RT*_BADR, RT*SIZE
Read & Writes
Reads of Last RT*_BADR, RT*_SIZE
Base/Size Memory 32x32
Slice_ID[3:0]
RT_ID[3:0]
RT*_BADR
RT*_SIZE
Real Time Stream Fetcher (In RTEE)
FIFO**
Read Request (256-bit)
CP_MIU_RTR
** FIFO for Timing
CP_MIU
Need to Add:
1. RTEE Regs
2. RTEE Poll Counter
3. How to ID Register Return Data
Fetcher RTS
Fetcher RTR
Fetcher BUSY
Note: Next slice_id will be taken only after all read requests have been issued for the previous real-time stream and the SCLK_G is enabled.
Real-Time Command Queue (Re-Ordering)
Command Data [31:0]
Command Parsers (PFP & ME)
Automatically Dis-Arm Event When Executed
CG_CP_CPCLK_ACTIVE
Fetch Controller
Priority Encode
Fetcher Busy
Register Writes
Register Read Data
Polling Busy
Processing Busy
RTEE Interrupts
RBBM
Real-Time Writes and Reads (Polling) on Global Register Bus

Figure 9-2: Real-Time Stream Event Engine Top-Level

9.2.2 Real-Time Stream State Management (Updated: 04-26-2001)

The CRAYOLA has a dedicated state context for Real-Time Streams. The state context for RTS will be "0", which will allow increasing the number of contexts late in the design for non-RTS state management. The state set is a subset of one of the other state sets in the chip because Real-Time Streams bypass setup and therefore do not require state associated with that portion of the pipeline. By default, real-time streams are disabled. A status bit in the CP_STATE_CNTL bit is provided to determine if RT streams are enabled. The ME_INIT packet is used to enabled/disable the RT stream processing.

The beginning section (i.e. Start address = 0) of the Instruction Memory is also dedicated to Real-Time Streams. The end of the RT stream instruction memory is programmable.

9.3 Register Backbone Interface Unit (RBIU) (Updated: 03-05-2002)

The Register Backbone Interface Unit (RBIU) contains the control logic to respond to transactions on the Global Register Bus (GRB) and contains a large number of the internal registers of the Command Processor. Registers in the CP are distributed. The RBIU forwards registers to other internal blocks of the CP via an internal register bus. The following diagram shows waveforms for a write and read transactions on this internal bus.

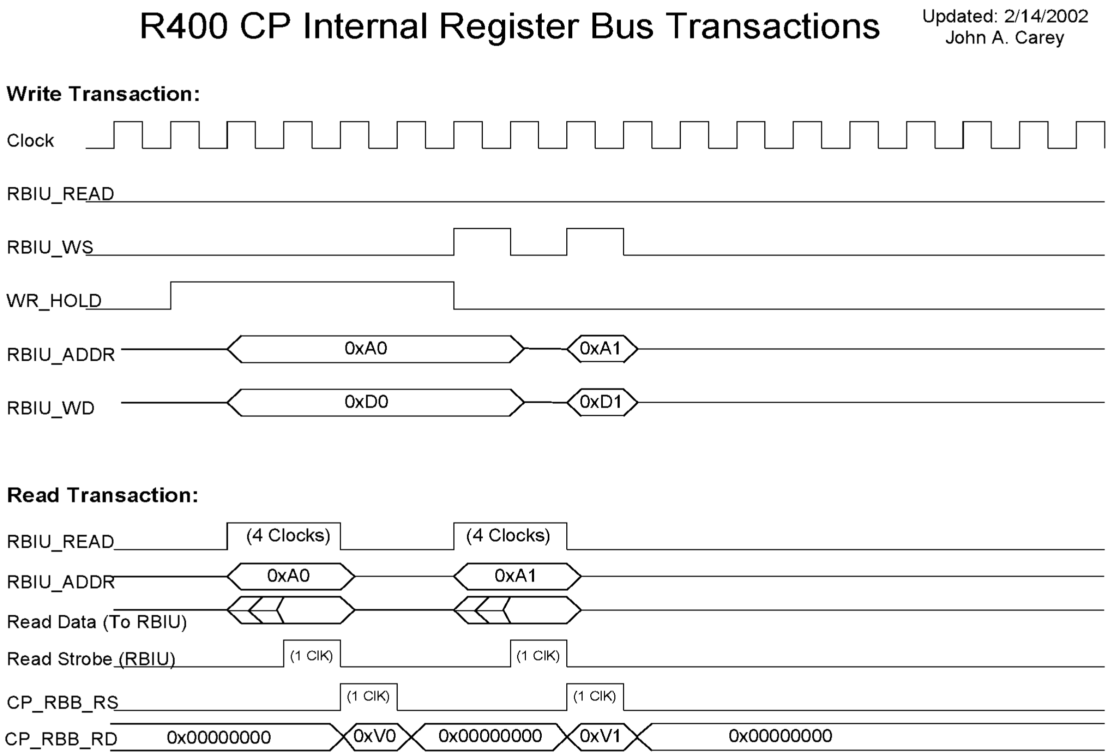


Figure 9-3: Internal CP Register Bus Transactions.

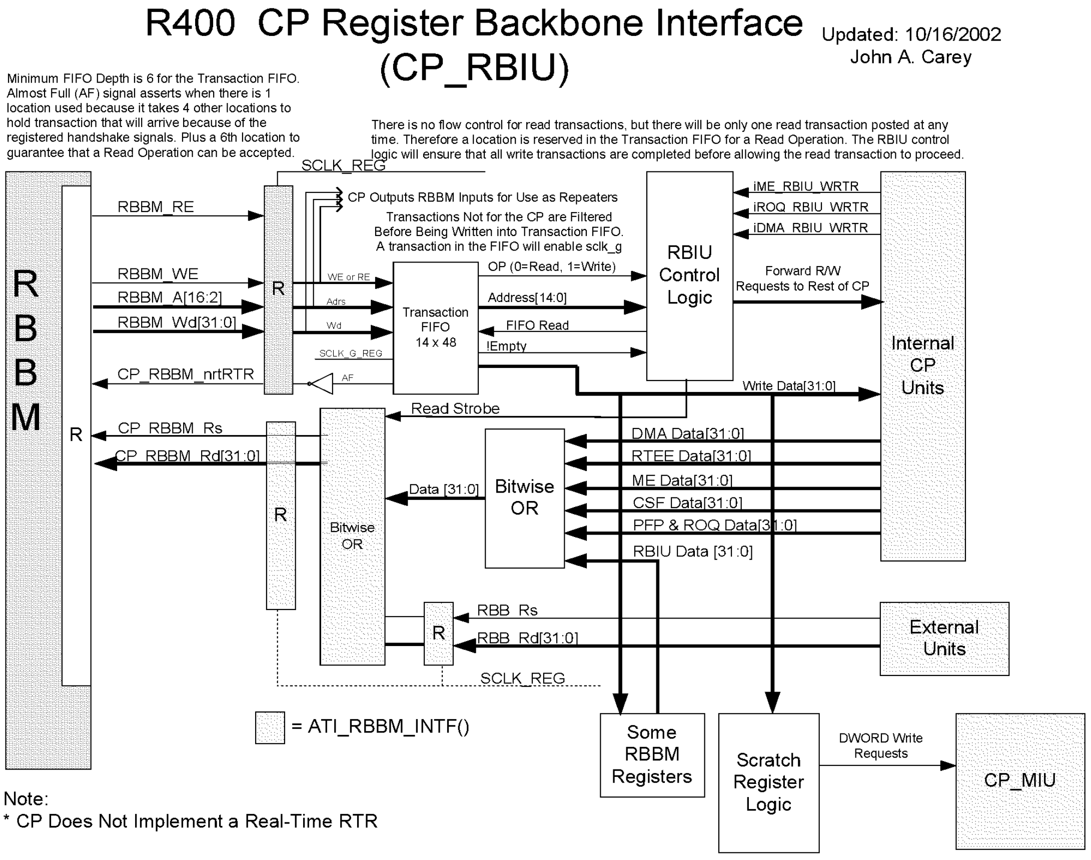
Register writes to internal CP clients are held-off (via WR_HOLD) for the following conditions:

1. Write to the VSD Address/Data FIFO when the FIFOs are full (In the ME).
2. Write to the PSD Address/Data FIFO when the FIFOs are full (In the ME).
3. Write to the CFD Address/Data FIFO when the FIFOs are full (In the ME).
4. Write to the Non-Real-Time Mem_Write registers when a write is pending (In the ME).
5. Write to the Real-Time Mem_Write registers when a write is pending (In the ME).
6. Write to the CP_DMA_TABLE_ADDR when the DTAQ FIFO is full (In the DMA).
7. Write to the CP_DMA_SRC_ADDR, CP_DMA_DST_ADDR, or CP_DMA_COMMAND registers when the ME has a DMA init pending (In the ME).
8. etc…

Register reads from the RTEE's base/size memory are held-off if the RTEE fetcher is reading the memory.

The RBIU also contains the Scratch Registers and the associated logic necessary to write their contents out to the Memory Interface Unit (MIU).

The RBIU also has the interface to the Read Data Return bus. Read data from external clients is registered, bitwise OR'd with CP internal read data, and output to the RBBM.



R400 CP Register Backbone Interface (CP_RBIU)

Updated: 10/16/2002
John A. Carey

Minimum FIFO Depth is 6 for the Transaction FIFO. Almost Full (AF) signal asserts when there is 1 location used because it takes 4 other locations to hold transaction that will arrive because of the registered handshake signals. Plus a 6th location to guarantee that a Read Operation can be accepted.

There is no flow control for read transactions, but there will be only one read transaction posted at any time. Therefore a location is reserved in the Transaction FIFO for a Read Operation. The RBIU control logic will ensure that all write transactions are completed before allowing the read transaction to proceed.

CP Outputs RBBM Inputs for Use as Repeaters

Transactions Not for the CP are Filtered Before Being Written into Transaction FIFO. A transaction in the FIFO will enable sclk_g

= ATI_RBBM_INTF()

Note:
* CP Does Not Implement a Real-Time RTR

Figure 9-4 : Register Backbone Interface Unit Block Diagram

9.4 Register Client Interface Unit (RCIU) (Updated: 03-07-2002)

The Register Client Interface Unit (RCIU) arbitrates among the internal CP requesters for reading/writing to the Global Register Bus (via the RBBM). The requesters include (In Priority Order):

1. Real-Time Stream – Processing transactions for Real-Time Stream from the Micro Engine.
2. Real-Time Event Engine (RTEE) – Polling of Registers within the Chip (Read-Only)
3. Pre-Fetch Parser – Orders DMAs of indices to the VGT (Write Only).
4. Micro-Engine – Constants, State Data, 2D Initiators, and Register Read/Writes.
5. DMA Engine – Mem->Reg and Reg->Reg Data Transfers.

For the ME and DMA transactions, a round-robin arbitration policy is implemented.

The RCIU also gets the read data from the RBBM and sends it to the appropriate requester within the CP. Only one of the unary RRTRs is asserted with the return data. The RCIU will stall consecutive read transactions to the RBBM when a prior read operation is in-progress.

Below is a top-level diagram of the RCIU.

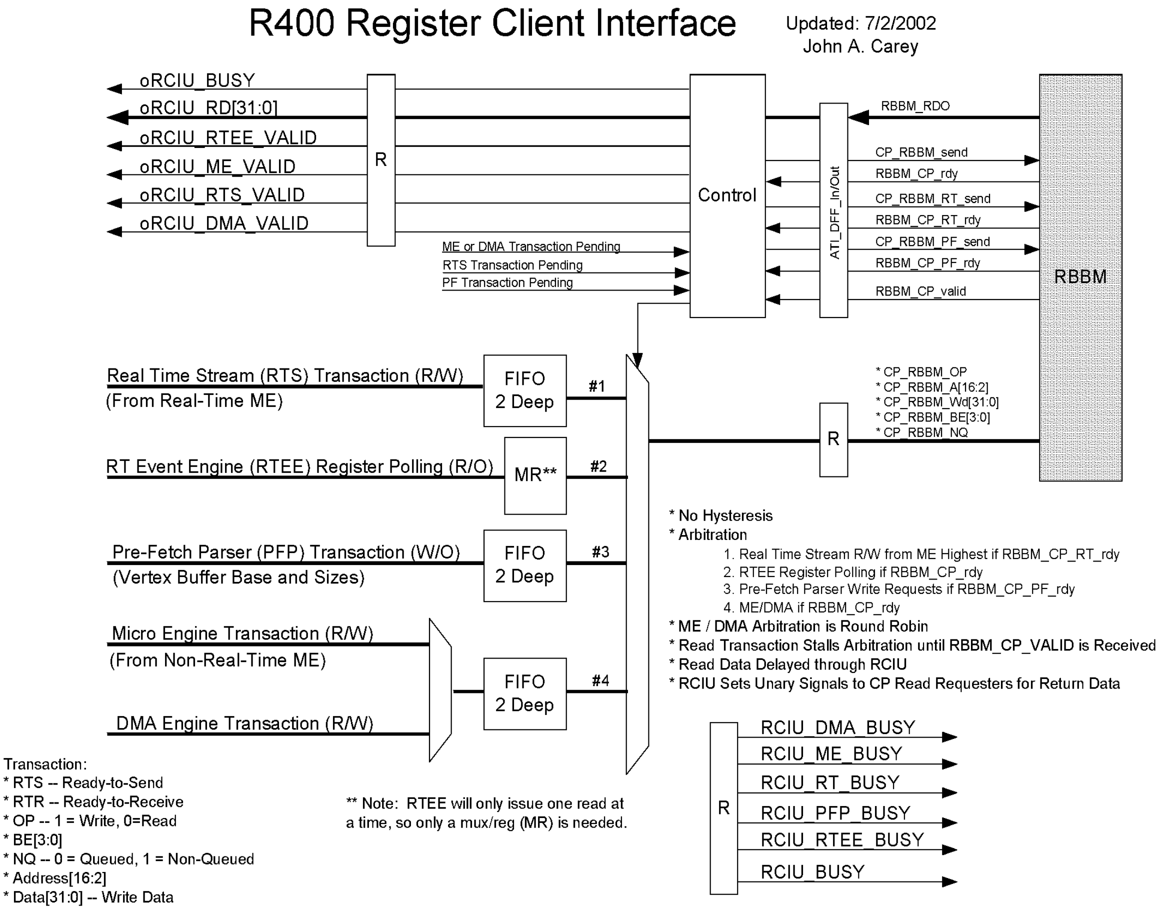

R400 Register Client Interface
Updated: 7/2/2002
John A. Carey
oRCIU_BUSY
oRCIU_RD[31:0]
oRCIU_RTEE_VALID
oRCIU_ME_VALID
oRCIU_RTS_VALID
oRCIU_DMA_VALID
R
Control
ME or DMA Transaction Pending
RTS Transaction Pending
PF Transaction Pending
ATI_DFF_In/Out
RBBM_RDO
CP_RBBM_send
RBBM_CP_rdy
CP_RBBM_RT_send
RBBM_CP_RT_rdy
CP_RBBM_PF_send
RBBM_CP_PF_rdy
RBBM_CP_valid
RBBM
Real Time Stream (RTS) Transaction (R/W)
(From Real-Time ME)
FIFO
2 Deep
#1
RT Event Engine (RTEE) Register Polling (R/O)
MR**
#2
Pre-Fetch Parser (PFP) Transaction (W/O)
(Vertex Buffer Base and Sizes)
FIFO
2 Deep
#3
Micro Engine Transaction (R/W)
(From Non-Real-Time ME)
FIFO
2 Deep
#4
DMA Engine Transaction (R/W)
* CP_RBBM_OP
* CP_RBBM_A[16:2]
* CP_RBBM_Wd[31:0]
* CP_RBBM_BE[3:0]
* CP_RBBM_NQ
R
* No Hysteresis
* Arbitration
1. Real Time Stream R/W from ME Highest if RBBM_CP_RT_rdy
2. RTEE Register Polling if RBBM_CP_rdy
3. Pre-Fetch Parser Write Requests if RBBM_CP_PF_rdy
4. ME/DMA if RBBM_CP_rdy
* ME / DMA Arbitration is Round Robin
* Read Transaction Stalls Arbitration until RBBM_CP_VALID is Received
* Read Data Delayed through RCIU
* RCIU Sets Unary Signals to CP Read Requesters for Return Data
Transaction:
* RTS -- Ready-to-Send
* RTR -- Ready-to-Receive
* OP -- 1 = Write, 0=Read
* BE[3:0]
* NQ -- 0 = Queued, 1 = Non-Queued
* Address[16:2]
* Data[31:0] -- Write Data
** Note: RTEE will only issue one read at a time, so only a mux/reg (MR) is needed.
R
RCIU_DMA_BUSY
RCIU_ME_BUSY
RCIU_RT_BUSY
RCIU_PFP_BUSY
RCIU_RTEE_BUSY
RCIU_BUSY

Figure 9-5: Register Client Interface Unit Block Diagram

9.5 Memory Interface Unit (MIU) (Updated: 04-22-2002)

The Memory Interface Unit (MIU) arbitrates between the internal CP's read requesters (Ring Buffer, Indirect Buffers, State Data, Real-Time Stream, Micro Engine, and DMA Engine) for reading from memory and between the CP's internal write requesters (Scratch Registers, Timestamps, Ring Buffer Read Pointer, DMA Engine, and Micro-Engine writes) for writing to memory.

The MIU also handles the conversion of the "flat" read/write requests from the internal CP units to the protocol and format that is required for interfacing to the Memory Hub.

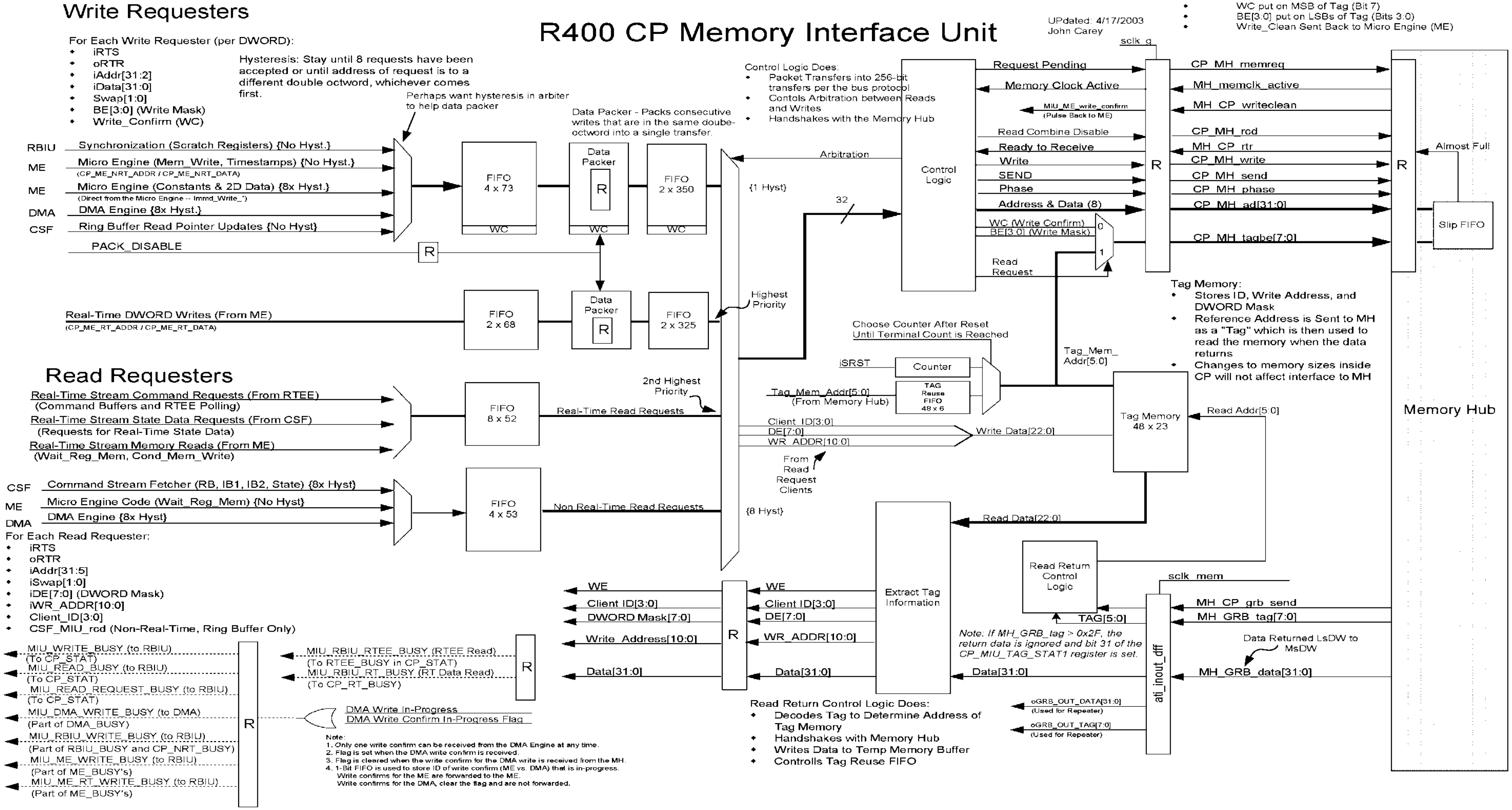
The MIU also helps with the coherency between data writes and processes that read the written data (i.e. Set_Constant and Load_Constant_Context). Writes from the micro engine can include a "write confirm" flag. The MIU will issue the write and generate a pulse back to the Micro Engine to indicate that the write was completed past the read/write arbiter in the Memory Hub. The MIU will stall consecutive "write confirm" transactions until there is not one in progress. Read operations and write operations without the "write confirm" flag set can be sent to the MH while a "write confirm" write operation is in-progress.

For write operations, the MIU has a "Data Packer" that will compress consecutive data writes that reside in the same double-octword into a single double-octword data transfer. The data packer is enabled by default, but can be disabled by setting a "pack disable" control bit in the CP_DEBUG register.

The MIU also is the receiving point of all the read return data from the Memory Hub (MH). The MIU decodes the TAG information to determine the destination of the data (i.e. Ring buffer, Indirect Buffers, Real-Time Stream Data, etc.).

The MIU can support up to 48 outstanding read requests. This is enough to cover the expected AGP latency. See other section of this document that discusses AGP latency issues.

A top-level diagram of the MIU is shown below.



- WC put on MSB of Tag (Bit 7)
- BE[3:0] put on LSBs of Tag (Bits 3:0)
- Write_Clean Sent Back to Micro Engine (ME)

Hysteresis: Stay until 8 requests have been accepted or until address of request is to a different double octword, whichever comes first.

Perhaps want hysteresis in arbiter to help data packer

Data Packer - Packs consecutive writes that are in the same doube-octword into a single transfer.

Control Logic Does:
- Packet Transfers into 256-bit transfers per the bus protocol
- Contols Arbitration between Reads and Writes
- Handshakes with the Memory Hub

Tag Memory:
- Stores ID, Write Address, and DWORD Mask
- Reference Address is Sent to MH as a "Tag" which is then used to read the memory when the data returns
- Changes to memory sizes inside CP will not affect interface to MH

Note: If MH_GRB_tag > 0x2F, the return data is ignored and bit 31 of the CP_MIU_TAG_STAT1 register is set.

Read Return Control Logic Does:
- Decodes Tag to Determine Address of Tag Memory
- Handshakes with Memory Hub
- Writes Data to Temp Memory Buffer
- Controlls Tag Reuse FIFO

Note:
1. Only one write confirm can be received from the DMA Engine at any time.
2. Flag is set when the DMA write confirm is received.
3. Flag is cleared when the write confirm for the DMA write is received from the MH.
4. 1-Bit FIFO is used to store ID of write confirm (ME vs. DMA) that is in-progress.
   Write confirms for the ME are forwarded to the ME.
   Write confirms for the DMA, clear the flag and are not forwarded.

Figure 9-6: Memory Interface Unit Block Diagram.

9.6 DMA Engine (Updated: 05-27-2002)

In prior CP designs, there were two DMA engines (GUI and VID). These were actually two instances of the same DMA design.

The CRAYOLA will have a single DMA engine that has the same functionality as one of the legacy DMA engines. The reasons for keeping a DMA engine in the CP are as follows:

1. R300 is considering using the DMA engine as a fast method to same chip state for an optimized Driver model. It is assumed that this could be used for CRAYOLA.
2. The DMA engines have been helpful for working-around bugs elsewhere in the chip.

9.6.1 DMA Engine Description

The DMA Engine has several processes running in parallel. First is the Descriptor Table Walker, which is an address generator, seeded with the value from the bottom of the Descriptor Table Address Queue (DTAQ). It fetches a descriptor from memory, and passes it along to the DMA Command Interpreter. The Descriptor Table Walker also snoops on the End-Of-List (EOL) bit of the Descriptor's command to determine if it is done with the current table, and should look to the DTAQ for more work.

The DMA Command Interpreter takes in the 4-dword Descriptor, and generates a sequence of source and destination addresses, and keeps track of the Transfer Count for the DMA transfer.

The DMA Engine can perform the following transfers:

1. Memory-to-Memory
2. Memory-to-Register
3. Register-to-Memory
4. Register to Registers

The DMA Engine will do byte alignment of the data from the source to destination surfaces if they are different.

9.6.2 Starting a DMA Operation

There are two methods to initiate a DMA operation – Descriptor Tables or Direct Descriptor Entry Register Writes.

To program a DMA operation via Descriptor Tables, the programmer has to build the table in the frame buffer first, being sure to mark the last entry of the list as "End Of List". Then, the programmer can write the starting address of the descriptor table into the Descriptor Table Address Queue (DTAQ) through the CP_DMA_TABLE_ADDR port. The action of writing the first "start" address into the DTAQ will trigger the DMA operation. The CP can store up to 8 pending table addresses in its DTAQ.

The type of transfer operation depends on the CP_DMA_COMMAND DWORD in the Descriptor. It controls such variables as: the length of the transfer, whether the Source/Destination addresses are in memory-space or register-space, whether the Source/Destination addresses auto-increment with each transfer, and whether an interrupt is generated when the entire Descriptor Table has been processed.

The second method - Direct Descriptor Entry Register Writes – involves writing the three DMA Entry registers directly. Three registers are provided for each of the DMA engines (CP_DMA_SRC_ADDR, CP_DMA_DST_ADDR, CP_DMA_COMMAND). The contents of these registers have the same fields as the SRC_ADDR, DST_ADDR, and COMMAND DWORDs of the descriptor table entry. Except that the EOL is hard-coded TRUE in the COMMAND DWORD. Writing to the CP_DMA_COMMAND register initiates a DMA operation using the descriptor described in all three registers. A table of descriptors can be built from multiple Type-0 packets each containing the SRC, DST, and COMMAND data.

9.6.3 *Overview of DMA Operation*

The DMA engine in the Command Processor fetches a command from the memory. This command tells the DMA Engine what to do. The command in memory is stored in a structure known as a *Descriptor*, having a four-DWORD format as shown below:

Ordinal	Name	Bit	Function
0	SRC_ADDR	31:0	Source address
1	DST_ADDR	31:0	Destination address
2	COMMAND	31:0	Command word. (See description below)
3	(Reserved)	31:0	

The COMMAND word has the following format:

31	EOL	End Of List Marker
30	INTDIS	End of List Interrupt Disable
29	DAIC	Destination Address Increment Control (0 → Increment, 1 → No Increment)
28	SAIC	Source Address Increment Control (0 → Increment, 1 → No Increment)
27	DAS	Destination Address Space (0 → Memory, 1 → Register)
26	SAS	Source Address Space (0 → Memory, 1 → Register)
25:24	DST_SWAP	Destination Endian Swap Control (0 → None, 1 → 16-bit, 2 → 32-bit, 3 → Half DWORD)
23:22	SRC_SWAP	Source Endian Swap Control (0 → None, 1 → 16-bit, 2 → 32-bit, 3 → Half DWORD)
21	QUEUED	Queued vs. Non-Queued Path Control (0 → Non-Queued, 1 →Queued)
20:0	BYTE_COUNT[20:0]	Byte Count of Transfer

There are some constraints on the programming of the Descriptor, as follows: If either the Source or the Destination is in the register address space, or is programmed to be non-incrementing, then the atomic transfer unit is assumed to be a DWORD. Namely, the bottom two bits of the BYTE_COUNT and the Address will be ignored (assumed “00”).

Note that a BYTE_COUNT of zero will perform no operation.

Multiple Descriptors may be stored contiguously in memory to make up a *Descriptor Table (DT)* (see Figure 9-8). The last Descriptor in the Descriptor Table must be marked as such so that the DMA engine knows when to stop consuming commands.

The programmer provides the DMA engine with a pointer to the beginning of the Descriptor Table, and the DMA engine fetches one Descriptor at a time, interprets the command to carry out a transfer, and then moves on to the next Descriptor in the table. As mentioned above, the DMA engine will stop when it reaches the last Descriptor in the table.

There is a bit called CP_SYNC in the Descriptor Address register (CP_DMA_TABLE_ADDR). If this bit is set, the DMA will “lock-out” the micro engine (ME) from performing any writes on the register backbone while the DMA is active.

A DMA channel may have its operation aborted by writing a ‘1’ to the ABORT_EN bit of the CP_DMA_STATUS register. It is important that the programmer then poll the ACTIVE bit of that same register, waiting for a value of ‘0’, before writing a ‘0’ to the ABORT_EN bit. Once the ACTIVE bit is ‘0’, the programmer is guaranteed to read-back stable state from all DMA registers.

The DMA engine generates an interrupt at the end of the DMA list (table) if the INTDIS is false. The DMA engine waits until the last write associated with the DMA has been accepted by the Memory Hub (MH) and is past the point of arbitration in the MH before raising the interrupt. The DMA raises the DMA_INT_STAT interrupt which is listed in the CP_INT_STATUS register. This will generate an interrupt out of the CP if the DMA_INT_MASK is set in the CP_INT_CNTL register.

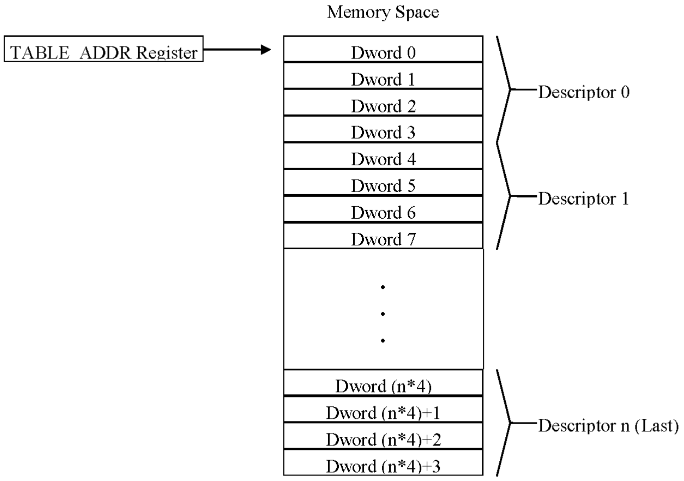


Figure 9-7: Descriptor Table Layout in Memory

An alternate method to writing the CP_DMA_TABLE_ADDR register to initiate a DMA operation is to write the descriptors directly to the CP. This saves the fetching of the descriptor table from memory.

Three registers are provided for each of the DMA engines (CP_DMA_SRC_ADDR, CP_DMA_DST_ADDR, CP_DMA_COMMAND). The contents of these registers have the same fields as the SRC_ADDR, DST_ADDR, and COMMAND DWORDs of the descriptor table entry described above. Except that the EOL is hard-coded TRUE in the COMMAND DWORD. Writing to the CP_DMA_COMMAND register initiates a DMA operation using the descriptor described in all three registers. A table of descriptors can be built from multiple Type-0 packets each containing the SRC, DST, and COMMAND data.

9.7 Pre-Fetch Parser (PFP)

9.7.1 Pre-Fetch Command Stream Queues (Updated 03-06-2002)

The PFP contains the memories that store the pre-fetched Ring Buffer Packets, Indirect Buffer Packets, Real-Time Stream Packets, and State Data. The associated control logic also performs the re-ordering the data from the Memory Hub. The outputs of the Ring, Indirect, and Real-Time command pre-fetch buffers go to the Pre-Fetch Parser. The output of the state data buffers go directly to the Micro Engine (ME) – they do not need to be interpreted. Each stream has a set priority for its access to the PFP.

Physically, there are two memories – one for Ring, Indirects in the Ring, and Indirect #2 data and a second for Ring/Indirect State, Real-Time, and Real-Time State data. Real-time streams are expected to be about 64 DWORDs in total.

Programmable thresholds for these memories allow the allocation to be changed to adjust the performance of the chip. See the CP_QUEUE_THRESHOLDS and CP_MEQ_THRESHOLDS registers.

9.7.2 Command Buffer Pre-Fetching (Updated: 06-07-2002)

The Pre-Fetch Parser (PFP) is a new function that is added to the CP design for CRAYOLA. One of the main functions of the PFP is to parse through the fetched command streams for future dispatches of Indirect Buffers. It also performs the pre-fetching of data for the Set_State, Load_Constant_Context, and Im_Load packets.

When an indirect buffer request is found, the PFP immediately issues the fetch request to the Command Stream Fetcher (CSF). The intention is to hide the latency of fetching indirect buffers behind the parsing of the other command streams. The PFP will continue to parse the command streams past the first pre-fetch. It will stop when the fetched indirect data is available, it encounters a Draw Packet, it encounters a state packet in the Ring Buffer, or the maximum number of pre-fetches is met. The PFP also stops when processing a "multi-pass" indirect buffer.

The INDIRECT_BUFFER packet has a disable bit that stops the PFP from immediately fetching the indirect buffers. When set the indirect buffer inits are pipelined out of the CP, through the RBBM and back to the CP. See the Pre-Fetch Disable.doc document for more details.

When an indirect buffer pre-fetch is issued, the PFP will consume the original packet that it parsed (Type-0 or INDIRECT_BUFFER) and insert an IB_PREFETCH_START packet into the command stream to the Micro Engine (ME). The IB_PREFETCH_START packet tells the ME to start consuming the indirect data from the next-highest-priority buffer. Another packet – IB_PREFETCH_END – identifies the end of the indirect buffer for the ME.

The PFP also parses the DRAW_INDX packet and will issue the request for indices to the VGT if the associated Viz Query test passes. It is expected that the indices will be available in the VGT by the time the associated DRAW_INITIATOR is written by the Micro Engine (ME). The PFP writes the INDEX_BASE and INDEX_SIZE to the VGT and writes the DRAW_INITIATOR as a Type-0 packet to the ME.

For state packets (Set_State, Load_Constant_Context, and Im_Load), the PFP remembers the last context's fetch pointers and sizes – Sub-Blocks, Vertex Shaders, Pixel Shaders, and constants. It compares the current pointers with the last context's and issues fetch requests to the CSF for the buffers that mismatch. Buffers that match the previous pointers are assumed to be identical and therefore will not be fetched. For every sub-block that is fetched, the PFP will insert a SUB-BLOCK_PREFETCH packet into the command stream to the ME. When instruction data is fetched, the PFP will insert an INSTR_PREFETCH packet into the command stream to the ME. When constant data is fetched, the PFP will insert an CONST_PREFETCH packet into the command stream to the ME.

The Pre-Fetch Parser also performs the 2D state default context fetch. When the PFP detects a 3D-to-2D transition, the 2D default state is fetched issuing an indirect buffer. The 2D Indirect Buffer's Base and Size are programmed into the CP as part of the chip initialization. The PFP will insert an IB_PREFETCH_START packet into the command stream to tell the Micro Engine to get data from the appropriate indirect buffer. See the "CP_Spec_2D_Appendix.doc" for how the PFP determines the 3D-to-2D transition.

9.7.3 PFP Multi-Pass Operation (Updated: 01-27-2002)

When the Pre-Fetch Parser (PFP) parses and INDIRECT_BUFFER packet with the MULTIPASS flag set, it stops pre-parsing the command stream and waits for a signal from the Scan Converter that indicates whether it should loop or continue. If the PFP is told to loop, it will re-fetch the indirect buffer and the Micro Engine will then re-process its contents. If the PFP is told to continue, the PFP will then just continue parsing the next instruction in the command stream.

9.7.4 PFP Constant Coherency Control (Updated: 04-22-2002)

The Pre-Fetch Parser (PFP) has an 8-bit counter that is incremented when it processes a Set_Constant packet with the CONST_WRITE_ENABLE bit set. A non-zero value on this counter indicates that there is a constant write pending in the Command Processor. The PFP will stall processing a Load_Constant_Context (LCC) packet if the counter is not zero. This ensures that the LCC packet will fetch the correct constant data that was previously written by the Set_Constant packet(s).

The PFP will likewise stall processing a Set_Constant packet if the counter has reached is maximum value.

The Micro Engine (ME) does the actual writing of the constants to memory and will initiate a “write confirm” for the last DWORD of the constant. The ME will get the write confirmation from the Memory Interface Unit (MIU) and issue a pulse on a signal to the PFP, which will decrement the counter mentioned above.

9.7.5 PFP Packet Processing Summary (Updated: 03-06-2002)

Here is the list of the packets that the PFP processes:

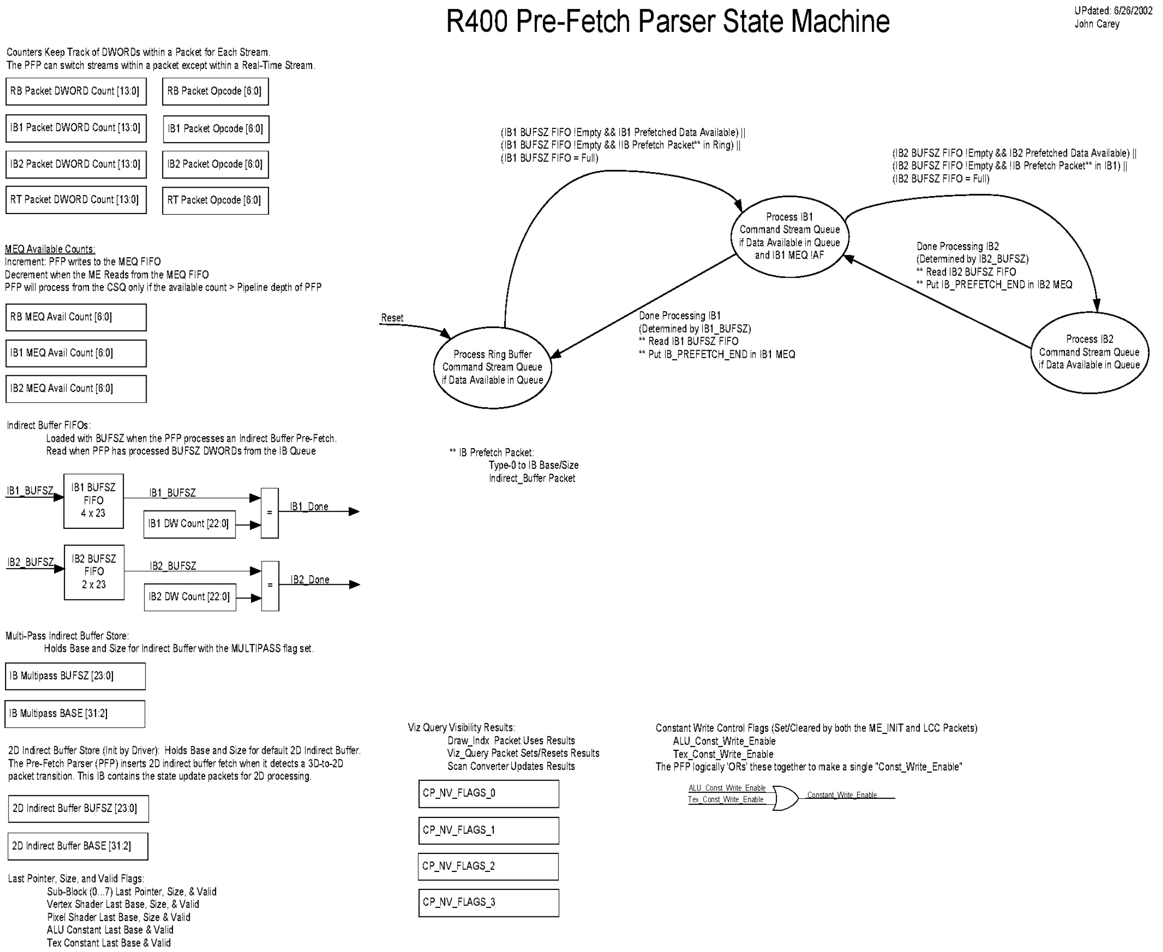
1. Type-0 Packets – If initiates Indirect Buffers they are replaced with IB_PREFETCH_START packet.
2. Type-1 Packets – Forwarded to the Micro Engine (Indirect Buffer Requests are Not Allowed via Type-1 Packets)
3. Type-2 Packets – Completely Consumed by the Pre-Fetch Parser.
4. Type-3 Packets:
   a. NOP – Completely Consumed by the Pre-Fetch Parser
   b. DRAW_INDX – Discarded if Not Visible, Otherwise DMA for Indices is Requested to the VGT
   c. Set_Constant – Clear Constant Valid Flags if Write-Back is Disabled. Also increment constant coherency counter, which is tested by the Load_Constant_Context packet.
   d. Set_State – Each Sub-Block is replaced with a SUB-BLOCK_PREFETCH packet and the VS/PS requests are replaced with INSTR_PREFETCH packets if the respective pointers mismatch.
   e. Im_Load – Code is fetched and Im_Load packet is replaced with INSTR_PREFETCH packet if pointers mismatch.
   f. Load_Constant_Context – Replaced with CONST_PREFETCH packet. The PFP waits for prior constant writes from Set_Constant packets to be confirmed.
   g. Invalidate_State – Consumed by the Pre-Fetch Parser. Clears selected valid bits for matching.
   h. Indirect_Buffer – Replaced with IB_PREFETCH packet. Also does MULTIPASS operation.
   i. ME_INIT – Clears all valid pointers for the rematching logic.

All other packets are just forwarded to the Micro Engine.

If an Indirect Buffer #2 request is in the Ring Buffer, the PFP will fetch it as an Indirect Buffer #1.

9.7.6 PFP Design (Updated: 02-27-2002)

The following diagram illustrates the top-level control that the PFP implements:



R400 Pre-Fetch Parser State Machine

UPdated: 6/26/2002
John Carey

Counters Keep Track of DWORDs within a Packet for Each Stream.
The PFP can switch streams within a packet except within a Real-Time Stream.

MEQ Available Counts:
Increment: PFP writes to the MEQ FIFO
Decrement when the ME Reads from the MEQ FIFO
PFP will process from the CSQ only if the available count > Pipeline depth of PFP

Indirect Buffer FIFOs:
Loaded with BUFSZ when the PFP processes an Indirect Buffer Pre-Fetch.
Read when PFP has processed BUFSZ DWORDs from the IB Queue

Multi-Pass Indirect Buffer Store:
Holds Base and Size for Indirect Buffer with the MULTIPASS flag set.

2D Indirect Buffer Store (Init by Driver): Holds Base and Size for default 2D Indirect Buffer. The Pre-Fetch Parser (PFP) inserts 2D indirect buffer fetch when it detects a 3D-to-2D packet transition. This IB contains the state update packets for 2D processing.

Last Pointer, Size, and Valid Flags:
Sub-Block (0...7) Last Pointer, Size, & Valid
Vertex Shader Last Base, Size, & Valid
Pixel Shader Last Base, Size & Valid
ALU Constant Last Base & Valid
Tex Constant Last Base & Valid

** IB Prefetch Packet:
Type-0 to IB Base/Size
Indirect_Buffer Packet

Viz Query Visibility Results:
Draw_Indx Packet Uses Results
Viz_Query Packet Sets/Resets Results
Scan Converter Updates Results

Constant Write Control Flags (Set/Cleared by both the ME_INIT and LCC Packets)
ALU_Const_Write_Enable
Tex_Const_Write_Enable
The PFP logically 'ORs' these together to make a single "Const_Write_Enable"

RB Packet DWORD Count [13:0]
RB Packet Opcode [6:0]
IB1 Packet DWORD Count [13:0]
IB1 Packet Opcode [6:0]
IB2 Packet DWORD Count [13:0]
IB2 Packet Opcode [6:0]
RT Packet DWORD Count [13:0]
RT Packet Opcode [6:0]
RB MEQ Avail Count [6:0]
IB1 MEQ Avail Count [6:0]
IB2 MEQ Avail Count [6:0]
IB1_BUFSZ
IB1 BUFSZ FIFO 4 x 23
IB1_BUFSZ
IB1 DW Count [22:0]
=
IB1_Done
IB2_BUFSZ
IB2 BUFSZ FIFO 2 x 23
IB2_BUFSZ
IB2 DW Count [22:0]
=
IB2_Done
IB Multipass BUFSZ [23:0]
IB Multipass BASE [31:2]
2D Indirect Buffer BUFSZ [23:0]
2D Indirect Buffer BASE [31:2]
(IB1 BUFSZ FIFO !Empty && IB1 Prefetched Data Available) ||
(IB1 BUFSZ FIFO !Empty && !IB Prefetch Packet** in Ring) ||
(IB1 BUFSZ FIFO = Full)
(IB2 BUFSZ FIFO !Empty && IB2 Prefetched Data Available) ||
(IB2 BUFSZ FIFO !Empty && !IB Prefetch Packet** in IB1) ||
(IB2 BUFSZ FIFO = Full)
Process IB1 Command Stream Queue if Data Available in Queue and IB1 MEQ !AF
Done Processing IB2
(Determined by IB2_BUFSZ)
** Read IB2 BUFSZ FIFO
** Put IB_PREFETCH_END in IB2 MEQ
Reset
Process Ring Buffer Command Stream Queue if Data Available in Queue
Done Processing IB1
(Determined by IB1_BUFSZ)
** Read IB1 BUFSZ FIFO
** Put IB_PREFETCH_END in IB1 MEQ
Process IB2 Command Stream Queue if Data Available in Queue
CP_NV_FLAGS_0
CP_NV_FLAGS_1
CP_NV_FLAGS_2
CP_NV_FLAGS_3
ALU_Const_Write_Enable
Tex_Const_Write_Enable
Constant_Write_Enable

Figure 9-8: Pre-Fetch Parser Top-Level Control Flow.

In the above drawing it is shown that the PFP must stop pre-parsing a stream under the following conditions:

8. Stop if it encounters any packet other than an indirect buffer pre-fetch packet after an indirect buffer has already been pre-fetched and not processed.
9. Stop pre-parsing ring when pre-fetched indirect buffer data is available.
10. Stop pre-parsing if it has filled up the pre-fetch queue in the Command Stream Fetcher (CSF).

9.7.7 PFP Connection to the Micro Engine (Updated: 02-26-2002)

The following diagram shows the Pre-Fetch Parser's connection to the Micro Engine.

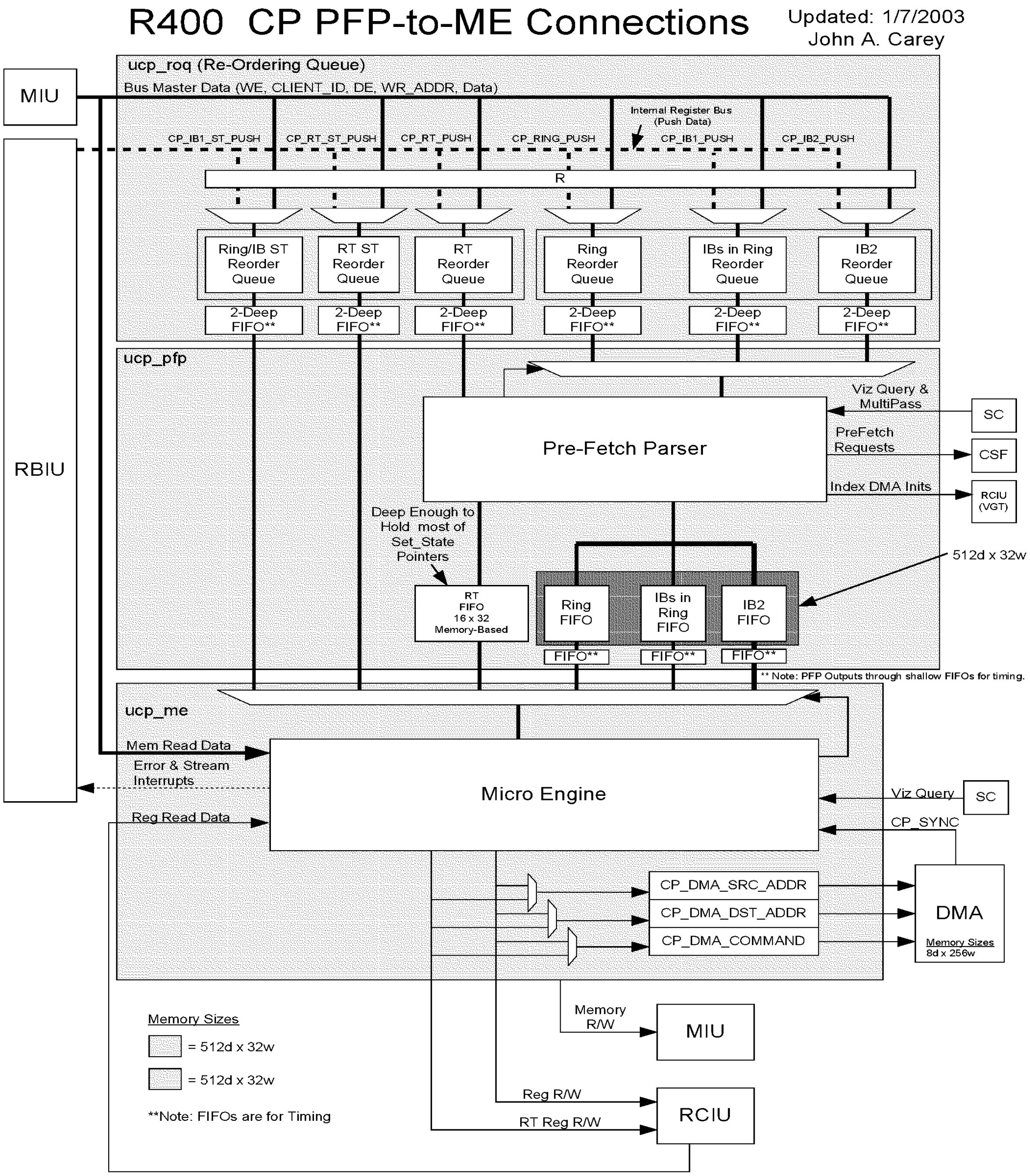


Figure 9-9: Pre-Fetch Parser Connection to Micro Engine.

9.8 Micro-Engine (ME)

9.8.1 Overview – CRAYOLA (Updated 05-03-2002)

The CP Micro-Engine (ME) is a programmable command parser. The compiled code for the ME is stored in an instruction cache/store on the chip and is loaded during chip initialization by discrete register writes.

Like prior designs, a custom compiler will be designed to support the compilation of code into the instruction code for the CP. The CRAYOLA CP ME design will support a super-set of the prior implementation's instructions and provide for a more general-purpose processor.

The ME operates on PM4 command packets that are documented in the "Specification of PM4 Command Packets CRAYOLA".

Other than the general-purpose functions, the ME has specific logic associated with State Management and 2D operations. The ME receives signals from the Renderer Common (RC) for de-allocating state sets and signals from the Sequencer for de-allocating locations in the Shader Instruction Memory. The Micro Engine holds-off writing data associated with a newly assigned state set until that state set is available. Similarly, the ME will hold-off updating the Instruction Memory until space is available for the shader update.

The ME has a long-word instruction format, broken into fields which control handshaking with the input and output stream as well as the internal functional units.

There two instances of the micro engine (ME) – one for non-real-time and one for real-time (RT) streams. The non-RT ME reads from the fetched Ring, Indirect #1, and Indirect #2 data queues. The real-time micro engine reads from the fetched real-time data queue.

Each ME can additionally transfer pre-fetched state data (Renderstate, Shader Code, or Constants). A dedicated DMA engine is used to perform these transfers. The ME calculates the starting address and size of the transfer and initiates the DMA engine. The data does not flow through the core of the Micro Engine. The DMA engine also handles the wrapping of the memory-mapped register addresses for shader code updates.

The ME can read/write any memory-mapped register or external memory location. This allows the ME to poll on the contents of an address, write semaphores to synchronize its operations with the Drivers, and to perform simple DMAs of data.

In addition, the ME can generate an interrupt for the purpose of synchronization with the Drivers. The interrupt is set by the processing of the CP_INTERRUPT PM4 command packet within a command stream.

For transactions to the memory-mapped registers, the ME has a 15-bit address bus [16:2], and a 32-bit data bus.

For transactions to the external memory locations, the CP supports a full 30-bit address A[31:2], a 2-bit swap code A[1:0], and a 32-bit data bus.

The Micro Engine has three register spaces that it can address: the Local register space, the Memory-Mapped register space, and the External Memory space. The Local register space includes all of the registers internal to the ME. The Memory-Mapped register space contains registers within the chip's register map. The External Memory space includes memory locations outside the chip (i.e. Local and System Memory). Not all of the "Local Registers" appear in the chip's register map, however an internal test bus allows visibility to these locations for debug purposes.

9.8.2 ME Error Checking (Updated: 04-15-2002)

The ME does the RESERVED_BIT, IB_START_CHECK, PROTECTED_MODE, Invalid Opcode, and Miscellaneous error checking. See the section on Error Checking / Command Buffer Validation elsewhere in this specification.

9.8.3 *Micro Engine Instruction Set (Updated: 07-01-2003)*

Below is the instruction set of the CP's Micro Engine (ME):

Data Transfer:

1. Load / Store:
   - Read & Write External Memory, Memory-Mapped Registers, and CP Internal Registers.
     - i. GPR ← External Memory (WAIT_REG_MEM, COND_MEM_WRITE)
     - ii. GPR ← Memory-Mapped Register (WAIT_REG_MEM, REG_TO_MEM)
     - iii. GPR ← Micro Engine Scratch Memory
     - iv. GPRa ← GPRb (Implementation: Src0=GPRa;SRC1=GPRb;Oper=shl;Immed=0x0)
   - Default Constants supported as embedded in CP machine code.
   - 15-bit Memory-Mapped Addresses [16:2] supported as embedded in CP machine code immediate field.
2. Conditional Writes – Based on bits set in Boolean register.
3. Word Swap: Swaps WORDs within a DWORD. Needed for Swapping DST_X and DST_Y terms in most of the 2D packets. This is implemented with a "rotate" by 16.

Integer Computations:

1. Addition/Subtraction (8-, 16-, 32-bit):
   - a. DST ← SRC0 ADD32 SRC1 –No Sign Extension
   - b. DST ← SRC0 ADD16 SRC1 – Bit 15 of Sources Sign Extended to Bit 31
   - c. DST ← SRC0 ADD8 SRC1 – Bit 7 of Sources Sign Extended to Bit 31
   - d. DST ← SRC0 SUB SRC1 –No Sign Extension
   - e. DST ← SRC0 SUB16 SRC1 – Bit 15 of Sources Sign Extended to Bit 31
   - f. DST ← SRC0 SUB8 SRC1 – Bit 7 of Sources Sign Extended to Bit 31
2. Min and Max Functions (16-bit Signed Value Comparison, 32-bit signed result):
   - a. DST[15:0] ← SRC0[15:0] MIN SRC1[15:0] – Bit 15 of result is sign extended to bit 31 of DST.
   - b. DST[15:0] ← SRC0[15:0] MAX SRC1[15:0] – Bit 15 of result is sign extended to bit 31 of DST.

3. Multiplication (14x14):
    a. DST[27:0] ← SRC0[13:0] MULT SRC1[13:0]
4. Logical (Bitwise):
    a. AND: DST← SRC0 AND SRC1
    b. OR: DST← SRC0 OR SRC1
    c. XOR: DST ← SRC0 XOR SRC1
    d. NOT (One's Complement): DST ← SRC0 NOT
5. Register Shift/Rotate (Left, Right)
    a. DST ← SRC0 shr <*value*>, DST ← SRC0 shl <*value*>, where *value* = {0,1,…,31}
    b. DST ← SRC0 ror <*value*>, DST ← SRC0 rol <*value*>, where *value* = {0,1,…,31}
6. Increment/Decrement Register Content:
    a. DST ← SRC0 INC (Implementation: Oper=Add32;Immed=0x00000001)
    b. DST ← SRC0 DEC (Implementation: Oper=Sub32;Immed=0x00000001)
7. Bit Replication (1bpp to 32bpp):
    a. DST ← SRC0 REPL <*value*>, where *value* is interpreted as the bit position in SRC0 to replicate (0 to 31).
8. Sign Extension
    a. DST ← SRC0 SIGNEXT <*Bit Position*>, where the bit the bit indicated is replicated up through bit 31.
9. Fix-to-Float: 8-bit integers to IEEE floating point integers – Not Normalized.
    a. The normalization to of the numbers is performed in the graphics engine.
    b. Implemented in output stage of the Micro Engine. The result can only be written to a Memory-Mapped Register.
    c. Selected by setting one of the upper bits in the ME's output address.
10. Time-Out Timer
    a. 16-bit loadable down-counter.
    b. Decrements every 16 core clocks when non-zero (2.62 ms at 400MHz).
    c. Microcode can test when the counter is zero via the Booleans register.
    d. Used for polling interval and general-purpose timer.

<u>Control Transfer:</u>

1. Subroutines – Support for up to 8 nested levels:
    a. CALL[Adrs]
        i. Stack ← IP+1
        ii. IP ← Adrs
    b. Return
        i. IP ← Stack
2. Jumps:
    - Absolute:
        i. Continue (IP ← IP+1)
        ii. Goto (IP ← Micro)
    - Conditional Branching & Comparisons
        i. Jump Boolean[n] Set (jbit)
        ii. Jump Boolean[n] Not Set (jnbit)

See the Micro Engine Register Set section for a listing of the Booleans.

Special Commands:

1. GMC Decode (oper = gmcdecode) – Decodes the 2D GUI_CONTROL DWORD to get the following parameters:
   a. Brush Foreground Color Present – Bit 31. Brush_Types = 0x0, 0x1, 0x6, 0x7, 0xD, 0xE
   b. Brush Backgound Color Present – Bit 30. Brush_Types = 0x0, 0x6
   c. Data_Format[5:0] – Bits 29:24 of Result. Based on the SRC_TYPE and DST_TYPE fields.
   d. Source Clip Disable = Bit 23. Set if the ROP code does not include a source: ROP = 0x00, 0x05, 0x0A, 0x0F, 0x50, 0x55, 0x5A, 0x5F, 0xA0, 0xA5, 0xAA, 0xAF, 0xF0, 0xF5, 0xFA, and 0xFF. The source clipping is disabled in the microcode for this condition.
   e. ROP[7:4] != ROP[3:0] Flag – Bit 22. Used to clear the C2.x Boolen.
   f. Pixels/DWORD[5:0] (PPDW) – Bits 21:16 of Result. Based on the SRC_TYPE and DST_TYPE fields.
   g. Line_32x1_Brush – Bit 15 of Result is set if Brush_Type = 0x6 or 0x7 (32 x 1 Mono Pattern)
   h. Number of Brush DWORDS[6:0] – Bits 14:8 of Result.
      i. Number Brush DWORDs = 0x02 for Brush_Types = 0x0 and 0x1
      ii. Number Brush DWORDs = 0x01 for Brush_Types = 0x6 and 0x7
      iii. Number Brush DWORDs = Function of DST_TYPE for Brush_Type = 0xA (See Table).
      iv. Number Brush DWORDs = 0x00 Otherwise.
   i. 2D Booleans – Bits 7:0 of the result are the respective 2D Booleans

The GUI_CONTROL DWORD is loaded as the SRC0 for the instruction.

SRC_TYPE = Bits [27,13:12]

DST_TYPE = Bits [11:8]

BRUSH_TYPE = Bits [7:4]

Byte_Pix_Order = Bit [14]

The Data_Format, Pixels/DWORD (PPDW), and 2D Booleans are generated as follows:

Data_Format[5:0]:

```
If (SRC_TYPE = Color)        // (i.e. Bits[27,13:12] = 01x)
   Data_Format = ‘0’ & DST_TYPE;  // SRC_TYPE=DST_TYPE
   PPDW = See Table for Calculating Based on DST_TYPE; // PPDW = Function (DST_TYPE)
//    Monochrome or TLU Color
else {
   If SRC_TYPE = 5 // 8bpp Source
     Data_Format = 0x02;       // Fmt_8 (TLU)
     PPDW = 0x04;              // 8-bpp
   Else If SRC_TYPE = 6 // 32bpp Source
     Data_Format = 0x06;       // FMT_8_8_8_8 (TLU)
     PPDW = 0x01;              // 32-bpp
   Else {
     If (Byte_Pix_Order == 0) {        \\ Mono Format (SRC_TYPE = 0 or 1)
       Data_Format = 0x00;   // Fmt_1_Reverse
       PPDW = 0x20;          // 1-bpp
     } else {
       Data_Format = 0x01;   // Fmt_1
       PPDW = 0x20;          // 1-bpp
     }
   }
}
```

Pixels/DWORD[5:0] (PPDW):

For Monochrome or TLU Color SRC_TYPE, see the Data_Format algorithm above.

For SRC_TYPE = DST_TYPE:

Destination Type (DST_TYPE[11:8])	Number of Brush DWORDs	Pixels / DWORD (PPDW)
0 :- Reserved	0x00	0x20
1 :- Reserved	0x00	0x20
2 :- 8 bpp pseudocolor	0x10	0x04
3 :- 16 bpp aRGB 1555	0x20	0x02
4 :- 16 bpp RGB 565	0x20	0x02
5 :- Reserved	0x00	0x01
6 :- 32 bpp aRGB 8888	0x40	0x01
7 :- Reserved	0x00	0x01
8 :- Y8 greyscale	0x10	0x04
9 :- RGB8 greyscale	0x10	0x04
10 :- Reserved	0x00	0x01
11 :- YUV 422 packed (VYUY)	0x20	0x02
12 :- YUV 422 packed (YVYU)	0x20	0x02
13 :- Reserved	0x00	0x01
14 :- aYUV 444 (8:8:8:8)	0x40	0x01
15 :- aRGB4444	0x20	0x02

2D Booleans [7:0]:

Boolean B4 is set by the microcode so is always output as zero.

Booleans B3, B6 and B7 are set based on the SRC_TYPE as follows:

Source Type (SRC_TYPE[27,13:12])	B3	B6	B7
0 :- Mono Opaque	0	0	0
1 :- Mono Transparent	1	0	0
2 :- Reserved	0	0	0
3 :- SRC_TYPE = DST_TYPE	0	0	0
4 :- Reserved	0	0	0
5 :- 8bpp TLU	0	1	1
6 :- 32 bpp TLU	0	1	0
7 :- Reserved	0	0	0

Boolean B0, B1, B2, and B5 are set based on the Brush_Type field in the GUI_CONTROL:

Brush_Type[7:4]	B0	B1	B2	B5
0 :- 8x8 Monochrome	0	1	0	1
1 :- 8x8 Monochrome	0	1	1	1
2 :- Reserved	0	0	0	0
3 :- Reserved	0	0	0	0
4 :- Reserved	0	0	0	0
5 :- Reserved	0	0	0	0
6 :- 32x1 Monochrome	0	1	0	1
7 :- 32x1 Monochrome	0	1	1	1
8 :- Reserved	0	0	0	0
9 :- Reserved	0	0	0	0
10 :- 8x8 Color	0	0	0	1
11 :- Reserved	0	0	0	0
12 :- Reserved	0	0	0	0
13 :- Solid	Set if SRC_TYPE != 0 or 1 (Set if SRC_TYPE != Mono)	0	0	0
14 :- Solid	Set if SRC_TYPE != 0 or 1 (Set if SRC_TYPE != Mono)	0	0	0
15 :- No Brush	Set if SRC_TYPE = 2,3,5,6	0	0	0

2. Color Repack (oper = clrrepack) – Repacks colors from format in Dst_Type to ARGB8888. This is used for converting the Foreground and Background packed colors when writing them to the constant memory.

```
SRC0 = Packed_Color[31:0]
SRC1 = Dst_Type[3:0]
Result[31:0]:
          If Dst_Type = 2 // 8bpp Pseudocolor
                    Result[31:24] = SRC0[7:0]    // Alpha
                    Result[23:16] = SRC0[7:0]    // Red
                    Result[15:08] = SRC0[7:0]    // Green
                    Result[07:00] = SRC0[7:0]    // Blue
          If Dst_Type = 3 // 16bpp aRGB1555
                    Result[24]    = SRC0[15]     // Alpha
                    Result[20:16] = SRC0[14:10]  // Red
                    Result[12:08] = SRC0[9:5]    // Green
                    Result[04:00] = SRC0[4:0]    // Blue
          If Dst_Type = 4 // 16bpp RGB565
                    Result[31:24] = 0x00         // Alpha
                    Result[20:16] = SRC0[15:11]  // Red
                    Result[13:08] = SRC0[10:5]   // Green
                    Result[04:00] = SRC0[4:0]    // Blue
          Else // 32bpp ARGB8888
                    Result[31:0]  = SRC0[31:0]   // aRGB8888
** Note: All bits in result not specified should be set to zero.
```

9.8.4 ME Tile Block Diagram

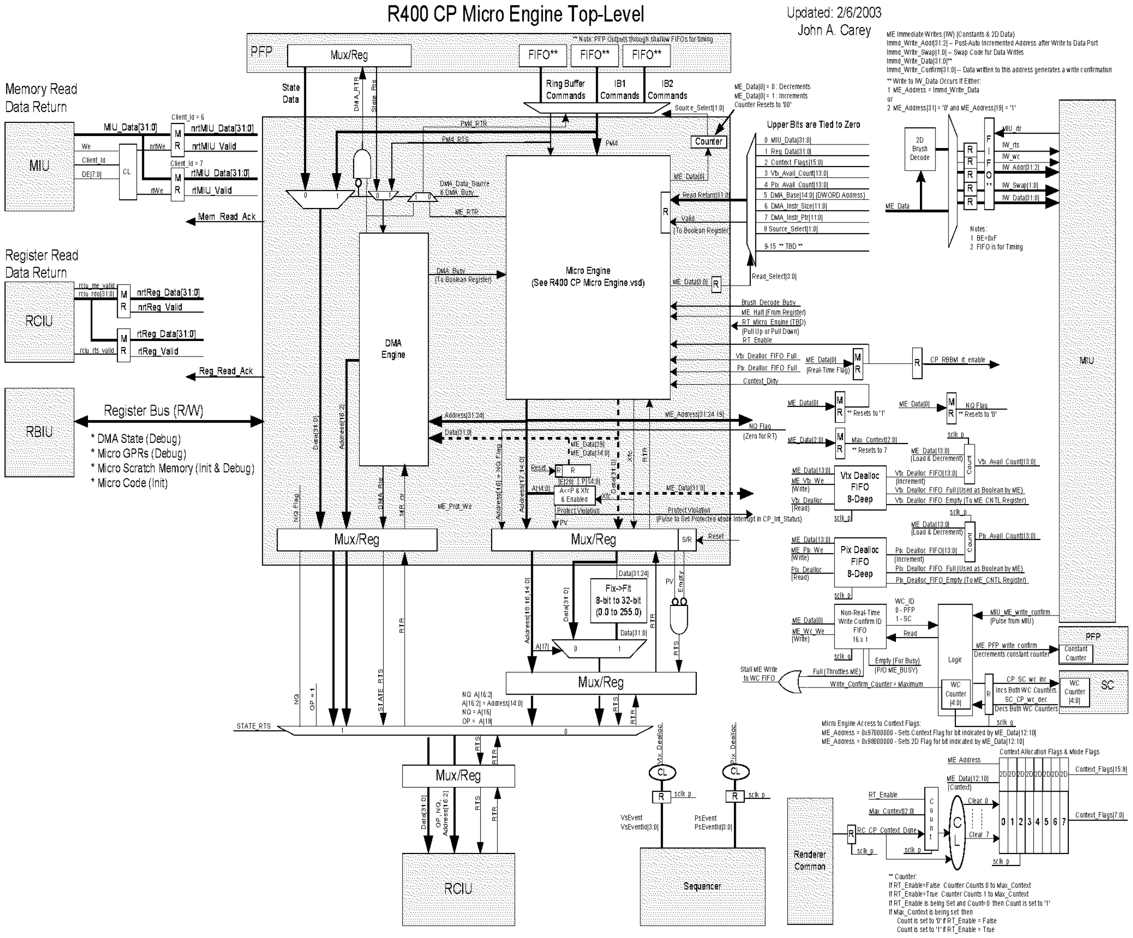
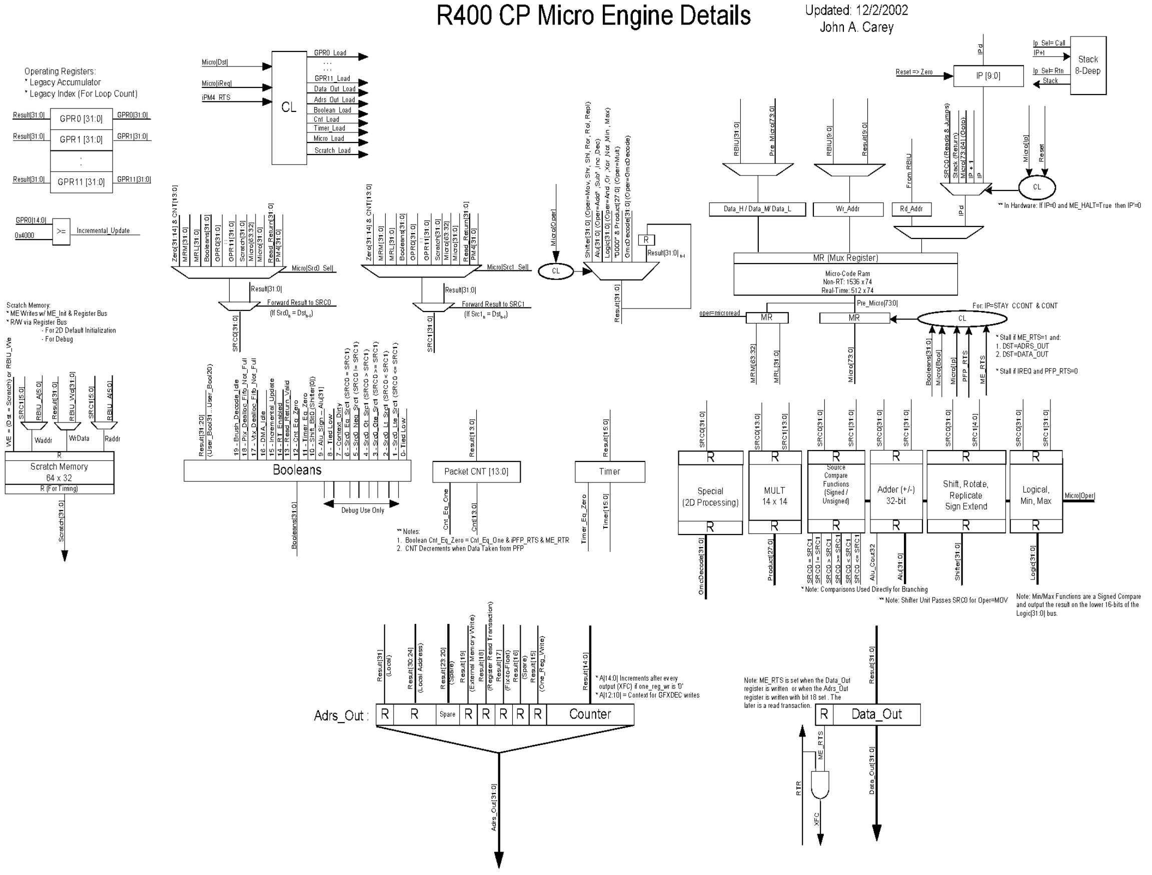


Figure 9-10 Micro Engine Tile Block Diagram

9.8.5 ME Details Block Diagram


R400 CP Micro Engine Details

Updated: 12/2/2002
John A. Carey

Operating Registers:
* Legacy Accumulator
* Legacy Index (For Loop Count)

Scratch Memory:
* ME Writes w/ ME_Init & Register Bus
* R/W via Register Bus
  - For 2D Default Initialization
  - For Debug

** Notes:
1. Boolean Cnt_Eq_Zero = Cnt_Eq_One & iPFP_RTS & ME_RTR
2. CNT Decrements when Data Taken from PFP

** In Hardware: If IP=0 and ME_HALT=True then IP'=0

For: IP=STAY CCONT & CONT

* Stall if ME_RTS=1 and:
1. DST=ADRS_OUT
2. DST=DATA_OUT

* Stall if IREQ and PFP_RTS=0

* Note: Comparisons Used Directly for Branching

** Note: Shifter Unit Passes SRC0 for Oper=MOV

Note: Min/Max Functions are a Signed Compare and output the result on the lower 16-bits of the Logic[31:0] bus.

* A[14:0] Increments after every output (XFC) if one_reg_wr is '0'
* A[12:10] = Context for GFXDEC writes

Note: ME_RTS is set when the Data_Out register is written or when the Adrs_Out register is written with bit 18 set . The later is a read transaction.

Figure 9-11: Block Diagram of Micro Engine

9.8.6 Register Set (Updated: 12-02-2002)

The Register State of the Micro Engine (ME) is composed of the following registers.

Register Name	Width	Description
IP	10	Instruction Pointer.
Packet CNT	14	Count Register Maintains a count of the number of DWORDs remaining in a packet. Increments when a DWORD from the command stream is consumed by the ME.
DATA_OUT	32	Data Output Bus – Common for ADRS_OUT and ADRS_MMR.
ADRS_OUT	32	ADRS_OUT[31:2] – DWORD address. ADRS_OUT[1:0] – Swap Field.
STACK	Same as IP	Instruction Pointer Stack – Instruction Pointer is pushed/popped from the stack to support micro-code subroutines. The stack is 8-deep.
GPR0	32	General-Purpose Register 0 – Source or Destination for a Micro Engine Operation.
:	:	: : :
GPR11	32	General-Purpose Register 11 – Source or Destination for a Micro Engine Operation.
BOOLEANS	32	Micro Engine's Control Flow Booleans: User Defined: User_Bool31..User_Bool20 – Managed by the Microcode Fixed Booleans: 19 – Brush_Decode_Idle – Brush Support Logic is Idle 18 – Pix_Dealloc_FIFO_Not_Full 17 – Vtx_Dealloc_FIFO_Not_Full 16 - DMA_Idle – The Micro Engine's DMA engine is Idle 15 – Incremental_Update 14 -RT_Enabled – Real-Time is Enabled 13 - Read_Return_Valid – Read Data Valid from either Memory or Register Bus 12 - Cnt_Eq_Zero – Packet Count Equals Zero 11- Timer_Eq_Zero – Micro Engine's Delay Timer 10 - Shift_Bit0 – Bit / Even (Odd) Tests 9- Alu_Sign – Sign of 32-bit ALU operation 8- Tied Low 7- Context_Dirty 6- Src0_Eq_Src1 5- Src0_Neq_Src1 4- Src0_Gt_Src1 3- Src0_Gte_Sr1 2- Src0_Lt_Src1 1 - Src0_Lte_Src1 0- Tied Low
TIMER	16	Timer used for poll interval and general-purpose waiting.

9.8.7 Scratch Memory (Updated: 08-06-2002)

The Micro Engine has access to a scratch memory for saving default values and temporary values. The GPRs can read/write these values. The Driver initializes some of these values with the ME_INIT packet during chip initialization.

The contents of the first locations in the Non-RT ME's scratch memory are fixed as these locations are written by direct register writes. The following table lists the contents of these fixed addresses.

Scratch Memory Address	Register Name	Description
0	DEFAULT_PITCH_OFFSET	2D Default Pitch Offset #1
1	DEFAULT2_PITCH_OFFSET	2D Default Pitch Offset #2
2	DEFAULT_SC_BOTTOM_RIGHT	2D Default Source Clipping Parameters #1
3	DEFAULT2_SC_BOTTOM_RIGHT	2D Default Source Clipping Parameters #2
4	2D BRUSH_BASE	Brush Base Address [31:15]
5	2D PALETTE_BASE	Palette Base Address [31:15]
6	2D IMMD_BASE	Immediate Data Base Address [31:19]
7	2D BOOLEANS	Default Boolean Values[31:19] to Control the 2D Shader.
8 to 48	N/A	Other Variables. The location of these variables is managed by the microcode.

9.8.8 *Instruction Format (Updated: 05-29-2003)*

A micro instruction has 75 bits, which are categorized into different fields. The definitions of the fields and related bits are given in the following table.

Num Bits	Bit Position	Field Name	Description	Choices
32-bit Field	31:0	IMMED	Immediate Data	@<Address> @<Value>
5-bit Field	36:32	SRC0	Source 0 for Operation. DST ← SRC0 [(*function*) SRC1]	PM4 READ_RETURN GPR0 : GPR11 MICROL – Lower DWORD of uRAM MICROM – Middle DWORD of uRAM SCRATCH (To GPR Only) BOOLEAN MRL – Micro Read Low MRM – Micro Read Medium CNT
5-bit Field	41:37	SRC1	Source 1 for Operation. See SRC0 Definition	See SRC0 Definition
5-bit Field	46:42	DST	Destination: Destination register for operation.	NONE (Default) – Fixed Booleans Updated GPR0 : GPR11 MICRO (Micro Code RAM) SCRATCH (Scratch Memory) BOOLEAN (User Booleans ←Result) ADRS_OUT DATA_OUT TIMER CNT
1-bit Field	47	Spare	Spare for Future Use	
5-bit Field	52:48	OPER	Operation: Selects operation to be performed on the two sources.	MOV (Default) – Shifer Passes SRC0 MULT ADD32 ADD16 – Bit 15 Sign Extended ADD8 – Bit 7 Sign Extended SUB32 (SRC0 – SRC1) SUB16 (SRC0 – SRC1) – Bit 15 Sign Extended SUB8 (SRC0 – SRC1) – Bit 7 Sign Extended AND OR XOR NOT SHR SHL ROR ROL REPL -- Replicate Selected Bit of Src0 MIN – 16-bit Signed Operation MAX – 16-bit Signed Operation MICROREAD SCOMP – 16-bit Signed Comparisions COMP – Unsigned Comparision SIGNEXT – Sign-Extend Selected Bit of Src0 GMCDECODE – See Spec for Details. CLRREPACK – See Spec for Detauls.
4-bit Field	56:53	IP	Instruction Pointer Select: Selects next instruction pointer. This field is modified by the Booleans.	STAY (IP ← IP) -- Default CONT (IP ← IP+1) CCONT If Bool_Sel=Cnt_Eq_Zero If Boolean(Bool_Sel)=1 and PFP_ME_XFC=1 Then IP ← IP+1 Else IP ← IP Else // Any other Boolean is selected If Boolean(Bool_Sel) = 1, IP ← IP+1 Else IP ← IP GOTO (IP ←Micro[8:0]) CALL (IP ← Micro[8:0], Stack ← IP+1) RTN (IP ← Stack[8:0]) JMPSRC0 (IP ← SRC0 – Computed Jumps) JNBIT If Bool(Bool_Sel) = 0, IP ← Micro[8:0] Else IP ← IP+1 JBIT If Bool(Bool_Sel) = 1, IP ← Micro[8:0] Else IP ← IP+1
5-bit Field	61:57	BOOL	Selects one of 32 Booleans to control the instruction flow.	31:18 - User_Bool31…UserBool19 18 – Pix_Dealloc_FIFO_Full 17 – Vtx_Dealloc_FIFO_Full 16 - DMA_Idle 15 – Incremental_Update 14 -RT_Enabled 13 - Read_Return_Valid 12 - Cnt_Eq_Zero 11- Timer_Eq_Zero 10 - Shift_Bit0 9- Alu_Sign 8- Alu_Cout32

Num Bits	Bit Position	Field Name	Description	Choices
				7- Context_Dirty 6- Src0_Eq_Src1 5- Src0_Neq_Src1 4- Src0_Gt_Src1 3- Src0_Gte_Sr1 2- Src0_Lt_Src1 1 - Src0_Lte_Src1 0- ME_Halt
1-bit Field	62	IREQ	Data from PFP is needed as source for instruction.	IREQ
1-bit Field	63	IRDY	Assert Ready-to-Receive to Pre-Fetch Parser	IRDY
10-bit Field	74:64	JUMP	Jump Address	Jump Address

Figure 9-12 Micro Engine Instruction Fields

9.8.9 Micro Engine Debugging Features (Updated: 11-08-2002)

When the ME_HALT (or ME_HALT_RT) bit set in the CP_ME_CNTL register, the ME is halted to allow observation of the internal status.

The ME_STATMUX field in the CP_ME_CNTL is a register test bus that provides the ability to read the Non-RT and Real-Time Micro Engine's operating registers. The selected data is read from the CP_ME_DEBUG_DATA register.

ME_STATMUX[14:0]	Description
0-63	Contents of the Micro Engine's Scratch Memory at Address=ME_STATMUX[14:0]
64	Packet CNT[13:0]
65	Timer[15:0
66	GPR0[31:0]
67	GPR1[31:0]
68	GPR2[31:0]
69	GPR3[31:0]
70	GPR4[31:0]
71	GPR5[31:0]
72	GPR6[31:0]
73	GPR7[31:0]
74	GPR8[31:0]
75	GPR9[31:0]
76	GPR10[31:0]
77	GPR11[31:0]
78	BOOLEAN[31:0]
79	Adrs_Out[31:0]
80	Data_Out[31:0]
81	IP[9:0] – Instruction Pointer
82	SP[9:0] – Stack Pointer
83	uCODE_INSTR[31:0]
84	uCODE_INSTR[63:32]
85	uCODE_INSTR[uC_WIDTH-1:64]
86	uCODE_RDLOW[31:0]
87	uCODE_RDMED[31:0]
88	qSCRATCH[31:0]
89	SRC0[31:0]
90	SRC1[31:0]
91	PRODUCT[27:0]
92	ALU[31:0]
93	SHIFTER[31:0]
94	LOGIC[31:0]
95	GMC_DECODE[31:0]
96	RESULT[31:0]
97	TIMER[15:0]
98	Sign-Extended SRC0

ME_STATMUX[14:0]	Description
99	Sign Extended SRC1
100	GMC_DECODE
101	GMC_PACK_COLOR
102	Read Return Data
103	Parser Debug Signals (MSB to LSB) in cp_me_parser.v: iME_HALT ME_IP_IDLE iRD_RTRN_VALID qRD_RTRN_VALID qME_RAM_WDATA_CNT[1:0] qME_RAM_RDATA_CNT[1:0] RAM_RADDR_INC qIP_START_CNT[1:0] PUSH_STACK_DATA POP_STACK_DATA NEXT_INSTR_SEL[1:0] NEXT_RAM_IP_SEL[2:0] iPFP_ME_RTS ME_PFP_RTR DATA_EXEC_RTR ADRS_EXEC_RTR BOOL MICRO_RD_AVAIL SUPPRESS_OUTPUT STALL_MULTIPLY FORWARD_SRC0 FORWARD_SRC1 DMA_SZ_WRITE DMA_SZ_WRITE_OUT qDMA_SZ_WRITE DMA_BUSY
104	Parser Debug Signals (MSB to LSB) in cp_me_parser.v: INCREMENTAL_UPDATE CNT_EQ_ZERO TIMER_EQ_ZERO CIN OPER16BIT OPER8BIT SRC1_SEL OPER_SEL REPLICATE_BIT SCOMP_SEL

ME_STATMUX[14:0]	Description
	SHIFT_OPER ALU_OPER LOGIC_OPER MULT_OPER GMCDEC_OPER ADRS_CNT_XFC 1'd0 qKEEP_RESULT PPDW_TABLE 2'd0
105	Parser Debug Signals (MSB to LSB) in cp_me_parser.v: GPR0_LOAD GPR1_LOAD GPR2_LOAD GPR3_LOAD GPR4_LOAD GPR5_LOAD GPR6_LOAD GPR7_LOAD GPR8_LOAD GPR9_LOAD GPR10_LOAD GPR11_LOAD DATA_EXEC_LOAD ADRS_EXEC_LOAD BOOLEAN_LOAD CNT_LOAD TIMER_LOAD MICRO_LOAD SCRATCH_LOAD qEXEC_SRC0 qEXEC_SRC1 qDATA_EXEC_LOAD qADRS_EXEC_LOAD WAIT_MICROREAD
106	Parser Debug Signals (MSB to LSB) in cp_me_parser.v: qEXEC_DEST qEXEC_OPER iME_BUS_RTR qME_OUT_RTS DATA_OUT_RTR ADRS_OUT_RTR

ME_STATMUX[14:0]	Description
	SCRATCH_READ SCRATCH_RE SCRATCH_RADDR SCRATCH_WE SCRATCH_WADDR RBIU_SCRATCH_VALID qINSTR_SEL 1'd0
107	Flow Control Debug Signals

9.8.10 Micro Engine Brush Decode Support Logic (Updated: 09-12-2002)

The CP needs to expand the compressed brush for 2D and write the uncompressed data to external memory. The CP uses the BRUSH_TYPE field in the GUI_CONTROL register of 2D packets to determine how to uncompress the data.

The decompression of the data is done by support hardware to the Micro Engine. The ME just writes the data as follows:

1. Brush_External_Address – External Address where the uncompressed brush data will be written. The address includes the current context.
2. Brush_GUI_Control – GUI Control DWORD from the 2D PM4 packet. The support hardware uses the BRUSH_TYPE field (bits 7:4) to determine how to uncompress the data.
3. Brush_Data – The ME writes the uncompressed** data to this local register to send it to the brush support logic.

Refer to the 2D Implementation spec and the PM4 spec for details of how the brush data is uncompressed.

** Note that for Brush Types 0x6 and 0x7 (32x1 patterns), the CP just writes the single DWORD to memory and does not convert the pattern to 32bpp.

9.8.11 ME Error Checking (Updated: 02-22-2002)

The Micro Engine will raise an interrupt if it detects an illegal Opcode. The ME will simply stop when this happens. This is implemented entirely in the micro code's jump table.

10. Power Management Support from CP (Updated: 02-03-2003)

The CP gets a permanent SCLK (sclk_global) from the Clock Generator (CG) and internally gates it for power management. The common "Master Clock Gater" circuit is instantiated several times to create gated clock domains in the CP. The diagram below shows the clock domains in the CP.

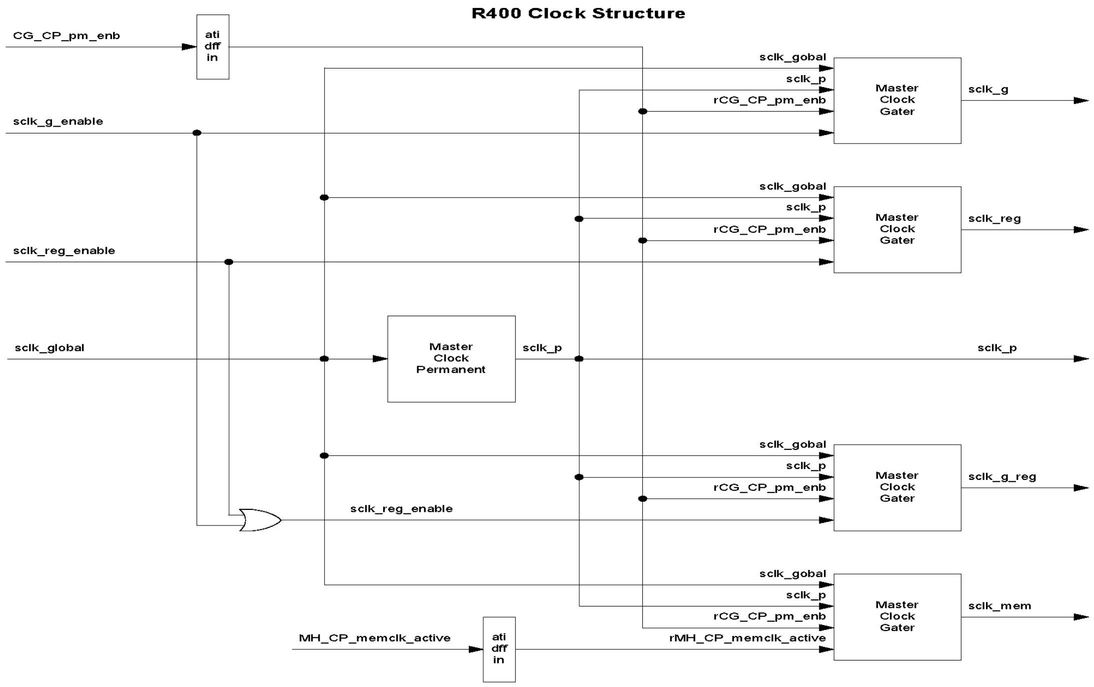


Figure 10-1 CP Clock Structure

Each of the clock gaters has an "enable" which also includes the DYNAMIC_CLK_DISABLE signal from the CP_DEBUG register. This can be used to disable the clock gating (i.e. Keep the clocks always on in the CP).

The overall power management function of the CRAYOLA can also be assisted by the CP. The CP receives status (i.e. "Idle" signals) from other parts of the chip and can poll registers for the purpose of initiating command streams (i.e. Real-Time Command Streams). Within these command streams it is possible for the CP to do the following via register read-modify-write operations:

1. Program the CG to turn on/off/reprogram clock branches.
2. Control of power switches to minimize leakage current in the chip.
3. Reprogramming voltage regulator on board.
4. Monitor an embedded temperature sensor in the chip and adjust settings of clocks.

There may be other possibilities, but they all involve using the CP's event-initiated command streams (i.e. RT streams) and its ability to perform read-modify-write operations on registers/memory locations within the chip.

11. Verification Plan (Updated: 01-18-2002)

The RBBM and CP are unit-tested with the same combined test bench. This test bench has a BIF model that supplies host transactions (Read & Write). The CP interfaces with a model of the Memory Hub (MH). Trackers are used at the output of the RBBM and CP to verify the transactions from both the BIF and CP are correctly forwarded to the register backbone.

The order of CP packet bring-up should be:

1. Debug with below packets:

    Type 0,1,2 Packets

    Type 3 Packets:

        SET_CONSTANT

        IM_LOAD

2. Continue debugging the following Type-3 packets

    LOAD_CONSTANT_CONTEXT

    SET_STATE

12. Performance Monitoring

The CP has a 48-bit counter that can be used to count the number of cycles that certain signals are asserted. This is used to determine if there are any performance issues within the CP. The following is a list of items that can be monitored by programming the CP_PERFCOUNTER_SELECT register:

```
6'h00 : // Always Count
6'h01 : // RBIU Transaction FIFO FUll
6'h02 : // RBIU Transaction Almost FIFO
6'h03 : // PFP Transaction is Waiting for RBBM in RCIU
6'h04 : // Real-Time Transaction is Waiting for RBBM in RCIU
6'h05 : // Real-Time Event Engine Transaction is Waiting for RBBM in RCIU
6'h06 : // Non-Real-Time Transaction is Waiting for RBBM in RCIU
6'h07 : // CSF Real-Time State Fetcher Waiting on MIU
6'h08 : // CSF Non-Real-Time State Fetcher Waiting on MIU
6'h09 : // CSF PFP I1 Request FIFO is FUll
6'h0a : // CSF PFP I2 Request FIFO is FUll
6'h0b : // CSF PFP State Request FIFO is FUll
6'h0c : // CSF PFP Real-Time State Request FIFO is FUll
6'h0d : // Ring Reorder Queue is Full
6'h0e : // I1 Reorder Queue is Full
6'h0f : // I2 Reorder Queue is Full
6'h10 : // State Reorder Queue is Full
6'h11 : // Real-Time State Reorder Queue is Full
6'h12 : // MIU Tag Memory is Full
6'h13 : // MIU WriteClean is In-Progress
6'h14 : // Real-Time Write Request Stalled by MIU Input FIFO
6'h15 : // Real-Time Read Request Stalled by MIU Input FIFO
6'h16 : // Non-Real-Time Write Request Stalled by MIU Input FIFO
6'h17 : // Non-Real-Time Read Request Stalled by MIU Input FIFO
6'h18 : // Write Confirm FIFO is FULL
6'h19 : // Vertex Shader Dealloc FIFO is FULL
6'h1a : // Pixel Shader Dealloc FIFO is FULL
6'h1b : // Vertex Shader Event FIFO is FULL
6'h1c : // Pixel Shader Event FIFO is FULL
6'h1d : // Cache Flush Event FIFO is FULL
6'h1e : // Micro Engine's RB Processing Starved by PFP
6'h1f : // Micro Engine's I1 Processing Starved by PFP
6'h20 : // Micro Engine's I2 Processing Starved by PFP
6'h21 : // Micro Engine's ST Processing Starved by PFP
6'h22 : // Any of the RT Slices are Armed and Waiting for a Trigger
6'h23 : // Any of the RT Slices are Triggered and Waiting for Execution
6'h24 : // Real-Time Slice 7 is Armed and Waiting for a Trigger
6'h25 : // Real-Time Slice 7 is Triggered and Waiting for Execution
6'h26 : // Real-Time Slice 8 is Armed and Waiting for a Trigger
6'h27 : // Real-Time Slice 8 is Triggered and Waiting for Execution
```

13. Design for Test

Signals in the CP are connected to the Test Bus for observation at the pins of the chip. The CP decodes select 0x03. Refer to the CRAYOLA CP test bus document for details on the wiring.

14. Physical Aspects

14.1 Hard Macro Requirement (Updated: 09-23-2002)

Description	Memory	Cycle Time (ns)	Unit Area (umsq)
48 x 23 RAM for the Tag Memory in the Memory Interface Unit. (MIU)	rfsd2_48x23cm1sw0	1.055	22,374
48 x 6 RAM for the Tag Re-Use FIFO in the Memory Interface Unit (MIU)	rfsd2_48x6cm4sw0	0.924	12,454
228 x 128 RAM for the Re-Ordering Queues (CSQ, STQ, MEQ)	TBD (3)	TBD	TBD
1408 x 75 RAM for Micro Engine Micro Code. (ME) Note: Bit 74 is not used by the Micro Engine.	High Density 1408 x 25(3)	TBD	TBD
384 x 74 RAM for Real-Time Micro Engine Micro Code (ME)	TBD	TBD	TBD
48 x 32 RAM/Register File for Micro-Engine (ME) Scratch Memory	rfsd2_48x32cm1sw0 (2)	TBD	TBD
32 x 32 RAM for Real-Time Stream Base/Size Pairs (RTEE)	rfsd2_32x32cm1sw0	0.841	21,020
8 x 256 RAM for the DMA Engine (DMA)	rfsd2_8x256cm1sw0	1.134	61,805
16 x 32 RAM for RT (MEQ)	rfsd2_16x32cm1sw0	0.769	15,854
32 x 31 RAM for State Data Base/Size Fetcher FIFO (CSF)	TBD	TBD	TBD
16 x 31 RAM for Real-Time Base/Size Fetcher FIFO (CSF)	TBD	TBD	TBD

15. Appendices

15.1 2D Implementation in CP

Refer to the CP_Spec_2D_Appendix.doc file for details of how 2D is implemented in the CP for CRAYOLA.

AUTHOR: John Carey

ISSUE TO	COPY NO.

Specification of PM4 Command Packets Crayola

THIS DOCUMENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATI TECHNOLOGIES INC. THROUGH UNAUTHORIZED USE OR DISCLOSURE.

APPROVALS

Name/Dept	Signature/Date

Remarks:

1. **Preserve this document's integrity:**
   - ⇒ **Do not reproduce any portions of it.**
   - ⇒ **Do not separate any pages from this cover.**

2. **This document is issued to you alone. Do not transfer it to another person.**

3. **Store this document in a locked cabinet or secure underground bunker. Do not leave it unattended.**

4. **When you no longer need this document, return it to Kathleen Bishop at ATI Research Inc. Do not discard it.**

"Copyright 2001, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished work created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this unpublished work. The copyright notice is not an admission that publication has occurred. This work contains confidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or transmitted in any form or by any means without the prior written permission of ATI Technologies Inc."

1 Table of Contents

2 Revision History

24 Oct 2001	Version 0.01	Baseline from Version 1.12 from Khan (R300). Information in 0.01 CRAYOLA PM4 spec is all subject to change. Added STATE_LOAD and LOAD_EXECUTE command packets as a proposal. -- J. Carey.
8 Nov. 2001	Version 0.03	Removed packets that are no longer needed for CRAYOLA & Added Others. Version 0.02 was not officially released.
28 Feb. 2002	Version 0.04	Set_State and Load_Constant_Context can be used within Real-Time Streams. Write-back of constants does not occur for a Set_Constant packet within a Real-Time stream. MPEG packet needs update. Added back 3D_DRAW_INDX_2 packet. Updated 2D packet description section. Added diagram for character mapping for Small_Text packet. Added SMALL_TEXT_CRAYOLA and HOSTDATA_BLT_CRAYOLA packets. Clarified sizes of fields in the Set_State packet. Removed AGP_FENCE packet; a Type-0 packet can be used instead. Removed the RT_WAIT_* packets; the WAIT_* packets can just be used within a real-time stream. Removed INVALIDATE_ALL_STATE; the INVALIDATE_STATE can be used to invalidate all of the state remembered by the CP. Sizes of sub-blocks are in the Set_State Packet. Removed MULTIPASS start and end packets. The multi-pass operation is performed with the INDIRECT_BUFFER packet. Remove the Data_Buffer packet. Added instruction immediate data packet for 2D processing.
07 Mar. 2002	Version 0.05	Renamed 2D Scratch Memory to 2D Defaults Memory. Update to the IB_PREFETCH_START packet format. Set_State and Load_Palette packets are "2D State" packets and will only assign a new context if they are the first packets after a 2D/3D "draw" packet.
15 Mar 2002	Version 0.06	Updates to Internal CP Packets for the PFP-to-ME communication. Moved Type-3 NOP packet to the miscellaneous section. Added MICRO_PREFETCH packet for Load_Execute command.
04 April. 2002	Version 0.07	Updates from the PM4 Design Review and 2D Discussions with Andy Gruber. Added VGT_INDX_OFFSET to 3D Draw Packets. Updated GUI Control with Renderer Backend registers that will be written. Update to POLYLINE packet – CP breaks into multiple draw packets to do brush offset modulation. Updated brush conversion table for 32x1 brushes – brush is written twice to memory to fill the entire 64-DWORD brush. Fix equations in Load_Constant_Context packet. Clarifications for 2D GMC table w.r.t. C0 and C1 constants. Updated CP_Interrupt packet to accommodate micro engine design. Updated order of invalidates for Invalidate_State packet. Remove support of "single" ring mode for Instruction Memory management. Updated packets for "write confirmation".
July 2nd 2002	Version 0.08	HostData_Blt_Pntr only supports one BIGCHAR per 2D design review. Added new 2D packets from J. Cheng. Set_State supports 8 sub-blocks with a size encoding as a multiple of 8. Added Set_CF_Constant packet. Updated ME_INIT packet. Instruction load packets support a size to load the full 12,288 DWORDs of instruction memory. Updated number formats for 2D packets. Updated compare function bit width for Cond_Mem_Write and Wait_Reg_Mem. Text Updates to MPEG_Index packet. Updates to 2D packets for 4KByte surface alignment. Changed COND_MEM_WRITE to COND_WRITE. Added GMC to new packets. Set_State and Load_Constant_Context packets should not be within the 2D stream as they are used to determine the 2D-to-3D mode switch. Strike through out-of-date packets.
Oct. 3 2002	Version 0.09	Update Draw_Indx text for Pre-Fetch Parser algorithm. Updated ME_INIT packet. Updated pseudocode related to the issuance of the event initiators to the Scan Converter. Updated the pitch format to be Pixels/32 and update the 2D packets with source to set the B4 (Source Fetch) 2D Boolean. Added sign extension note for the PLY_NEXTSCAN packet. Updates to AlphaBlend packet to add source data format.
Nov. 4 2002	Version 0.10	Update format of the MPEG_INDX packet to support the use of rectangle lists.
Nov. 5 2002	Version 0.11	Updated format of the Grad_Fill Packet.
Nov. 13 2002	Version 0.12	Im_Load and Im_Load_Immediate packets write the SQ_PS_PROGRAM register for real-time.
Nov. 18 2002	Version 0.13	Set RB_BLENDCONTROL.COLOR_DITHER_MODE to DITHER_LUT for Grad_Fill packets with DST_TYPE != 32bpp.
Nov. 20 2002	Version 0.14	Add Register Read/Mod/Write Packet. Add Note to Load_Palette Packet.
Nov. 21 2002	Version 0.15	Update to Register Read/Mod/Write Packet.
Dec. 3 2002	Version 0.16	Update IB_BASE in the Indirect_Buffer PM4 packet to be 31:5.
Dec. 13 2002	Version 0.17	Dummy values in IB_Prefetch packets are 0xdeadbeef by the pre-fetch parser.
Dec. 18 2002	Version 0.18	AA_Font and AlphaBlend PM4 Packet Updates.
Jan. 6 2003	Version 0.19	Clarification to Invalidate_State packet w.r.t. use of Mem_Write.
Jan 8 2003	Version 0.20	Update Reg_RMW packet to include register source for and_mask and or_mask. Add note that scan_count should be greater that zero for the Polyscanlines packet.
Jan. 13 2003	Version 0.21	Add new packet to control endian mode for 2D.
Jan. 17 2003	Version 0.22	Updated Booleans and Description for AlphaBlend packet.
Jan. 20 2003	Version 0.23	Added note to Ply_NextScan and NextChar about requiring a proceeding packet with a GMC to set-up the 2D state.
Jan. 20 2003	Version 0.24	Update AlphaBlend with equations and new RB_BlendControl settings. Update AAFONT for setting the B6 Boolean.
Jan. 22 2003	Version 0.25	Update to 2D_ENDIAN_MODE packet. AlphaBlend update for sources without Alpha term.
Jan. 24 2003	Version 0.26	Add note to LCC packet for the constant write enable behavior.
Jan. 29 2003	Version 0.27	HostData_Blt and PolyScanLines can specify a number of zero.
Jan. 30 2003	Version 0.28	ME_INIT in non-RT stream invalidates matching pointers and constant write enables.
Feb. 11 2003	Version 0.29	Removed SRC=DST (H/W) restriction for non-stretch Alphablend packets.
Feb. 23 2003	Version 0.30	Write Confirm Interval is for Future Use Only.
March 12, 2003	Version 0.31	Add Pre-Fetch Disable to the Indirect_Buffer packet. Added note to the HostData_Blt2 and HostData_Blt_Pntr packets indicating that the CP outputs a rectangle primitive per scan line to improve texture fetch performance.
March 20, 2003	Version 0.32	Added new Indirect_Buffer_PFD packet which does the pre-fetch disable implied instead of a control bit in the packet.
April 2, 2003	Version 0.33	RB_CLRCMP_MSK_HI and RB_CLRCMP_DST_HI written by 2D indirect buffer and not the microcode. The RB_CLRCMP_LO format is based on the pixel type.
April 4, 2003	Version 0.34	Add matching logic for the Loop and Boolean Constants in the LCC packet. Set_Constant also writes to memory.
April 14, 2003	Version 0.35	Document some setting restrictions to BitBlt and HostData_Blt packets.
April 17, 2003	Version 0.36	Addition to ME_INIT to allow checking for Type-0 and Type-1 packets in Indirect Buffers.
May 12, 2003	Version 0.37	Added New Type-3 to view 2D coherency rectangle generated by CP.
May 14, 2003	Version 0.38	Miscellaneous Clarifications.
May 19, 2003	Version 0.39	Correct values that ME_INIT sets the 2D coherency rectangle.
May 20, 2003	Version 0.40	Correction to GradFill's primitive type for rectangles.

3 Open Issues / Items to Complete (Updated: 05-07-2002)

1. Possible addition of 2nd default DST_PITCH_OFFSET value for the CRTC #2. *A: Yes.*
2. In PM4 design review (03-13-2002) S. Morein stated that the Sub-Block size encoding may change. On 03-28-2002, Steve Morein stated that the sub-block size issue will be resolved within 2 weeks (by 04-24-2002). *A: 05-06-2002: Set_State will support 8 sub-blocks & the Size encoding is (N+1)*8*.
3. May define more 2D packets that use the 3D engine resources so that CP does not initiate 2D Indirect Buffer between the 2D's use of the 3D pipe – *2D_GRADFILL, 2D_ALPHABLEND, 2D_AAFONT.*
4. For a POLYLINE packet with a single line segment, if X0=Y0 the entire packet should be discarded (J. Cheng). *A: 04-23-2002. No. The graphics pipeline hardware will take care of filtering this condition.*
5. How is the swap field supplied for Index Buffer in the Draw_Index packet? *A: It is provided as the upper bits of ordinal #6.*
6. A Conditional_Write_Mem packet was discussed at the PM4 design review (03-13-2002) by S. Morein. *A: The COND_MEM_WRITE packet has been added.*
7. Need to determine if the Ib_Preamble packet should contain anything more complicated than a fixed signature value. *A: This will be a fixed value for CRAYOLA.*
8. Does the CP need to perform the bytes-to-pixels conversion of the pitch? *A: 03-15-2002: No. The pitch supplied will be in terms of pixels.*
9. MPEG packet needs update. The registers it wrote for R300 may not exist in the CRAYOLA. *A:03-13-2002: The MPEG packet will be the same format as R300. The mask however needs to be zero.*
10. The Load_Palette packet may not be needed. Jeffrey Cheng will look into existing Driver code to see if it is ever used. It is believed that it is not used in the 9x/NT Drivers. *A: The LOAD_PALETTE packet is used for bitmaps with 4bpp or 8bpp, so it is needed.*
11. Reasons for removal of the Data_Buffer (a.k.a. Indx_Buffer) packet: *DRAW_INDX can be used to fetch indices in separate buffer, Im_Load can be used to fetch shader code, and Load_Constant_Context can be used to fetch constants.*
12. Definition of a Type-3 packet that is placed at the beginning of buffers to validate the buffer as "valid". *A: IB_PREAMBLE packet.*
13. Possibly the DRAW_INDX packet may need to support having the CP fetch the indices instead of the VGT. This will either be a separate packet or a new bit in the 2nd ordinal of the packet. Tim Kelley will provide feedback. *A: No.*
14. Need a definition of the "Null" packet for Viz Query. *A: Event initiators indicate Viz Query Begin and End.*
15. Need a review of the Multi-Pass packets. I think there are difficulties with these packets considering that we are now doing pre-fetching of command streams. *A: INDIRECT_BUFFER packet has a flag to indicate multi-pass.*
16. What is the Base and Size restrictions for Index Buffers? *A: Same as prior designs.*
17. Get confirmation on the need for a LOAD_EXECUTE packet. *A: CP will support this packet.*
18. Can the sub-block sizes be a power of 2? *A: 01-22-2002 Yes per Phil Rogers. The sizes are also embedded within the Set_State packet.*
19. Do we need separate Real-Time equivalent packets for the WAIT_REG/WAIT_MEM packet? *A: No. The packet can be used in the Real-Time stream as well.*
20. Should we have a separate Set_State packet equivalent for Real-Time Streams? *A: No. The Set_State packet can be used in a Real-Time stream with some restrictions.*
21. Since the CP still has a general-purpose DMA engine, is there still a need for the Reg_to_Mem packet? *A: Sure.*
22. Is there a need for the "fine" grain Invalidation control (Invalidate_State packet)? *A: Yes. In fact the INVALIATE_STATE packet can be used to invalidate all state, so the INVALIDATE_ALL_STATE packet can be removed.*

23. Can the Invalidate_All_State packet be removed and just use the Invalidate_State packet with all its control bits set? *A: Yes.*

24. Which register in the BIF does the “AGP FENCE” command packet write to? Why cannot the “AGP Fence” packet be just replaced with a Type-0 packet? *A: AGP Fence packet removed.*

25. Information needs to be added to the Draw Calls to associate them to a Per-Draw-Call Constant set. A: 10-30-2001: Per Steve Morein, there is no longer a requirement for PDCCs.

26. Command Packet Set_State_Base_Ptr should be implemented as a Type-0 command packet (E-mailed question to S. Morein on 10-01-2001)? A: This packet is not needed. All the sub-block pointers are full memory addresses, so they do not need an base address.

27. Is there a need to support the Type-1 packet? A: Yes for legacy reasons.

28. Is there a need for the “3D_Clear_*” PM4 packets A: 11-13-2001: No. Instead, primitives will be rendered with parameters set such that the CRAYOLA memories will be cleared. This is faster than discrete writes (8x8 can be processed per clock).

29. What form are the 2D packets for CRAYOLA? Are they the same and the CP does the translation to shader format or does the driver send new packets that have the data in a native CRAYOLA format? *A: The POR is that the command packets are the same. Steve Morein will work through the PM4 conversion requirements which we will review on September 19th.*

30. Because the Plan-Of-Record (POR) is that the CP does not have immediate data, a plan needs to be determined as to how this data now gets into the chip for the following packets: 3D_DRAW_IMMD & 3D_DRAW_INDX & Host_Blit & Small_Text. *A: CP would perhaps fetch the data, but then skip over the data instead of parsing. The CP instead would send a command do the Vertex/Index fetcher to fetch the data. If the data set is too large, the CP stream fetcher may also skip over fetching the data.*

31. Proposed new Type-3 packet used for dispatching Indirect Buffers for the Pre-Fetch Parser in the CP. E-mail sent on 09-24-2001. A: 09-25-2001 – Approved by Phil Rogers.

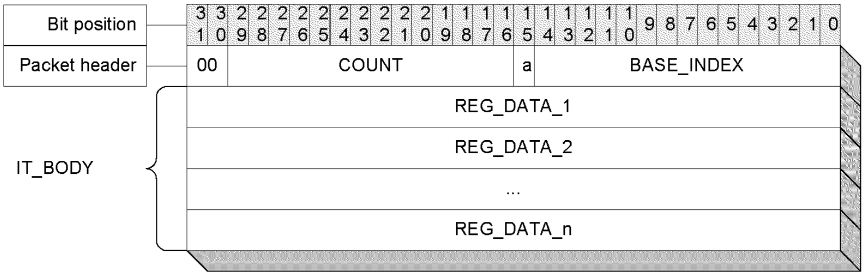
32. Can the existing opcodes that are not being used be re-assigned to the new packets being added for CRAYOLA? A: 09-25-2001 Yes. Per Phil Rogers.

4 Packet Types

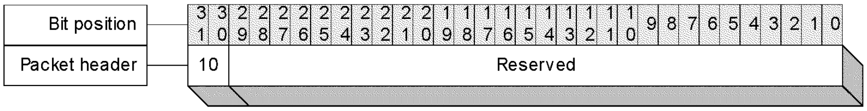
When programming in the PM4 mode, we do not need to write directly to registers to carry out drawing operations on the screen. Instead, what we need to do is to prepare data in the format of PM4 *Command Packets* in either system or video (a.k.a. local) memory, and let the Micro Engine to do the rest of the job.

Three types of PM4 command packets are currently defined. They are types 0, 2 and 3 as shown in the following figure. A PM4 command packet consists of a *packet header*, identified by field HEADER, and an *information body*, identified by IT_BODY, that follows the header. The packet header defines the operations to be carried out by the PM4 micro-engine, and the information body contains the data to be used by the engine in carrying out the operation. In the following, we use brackets [.] to denote a 32-bit field (referred to as DWORD) in a packet, and braces {.} to denote a size-varying field that may consist of a number of DWORDs. If a DWORD consists of more than one field, the fields are separated by '|'. The field that appears on the far left takes the most significant bits, and the field that appears on the far right takes the least significant bits. For example, DWORD [HI_WORD | LO_WORD] denotes that HI_WORD is defined on bits 16-31, and LO_WORD on bits 0-15. A C-style notation of referencing an element of a structure is used to refer to a sub-field of a main field. For example, MAIN_FIELD.SUBFIELD refers to the sub-field SUBFIELD of MAIN_FIELD.

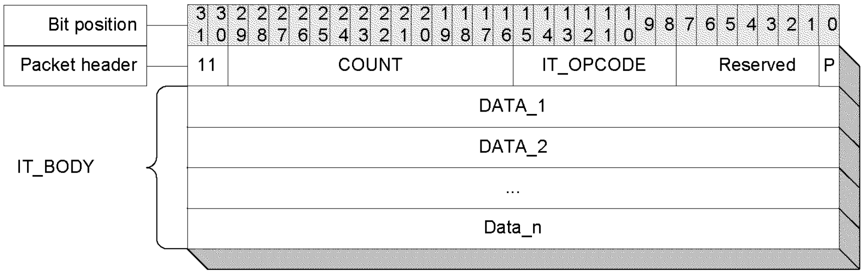
Type-0 packet



Type-2 packet



Type-3 packet



4.1 TYPE-0 PACKET

Functionality

Write *N* DWORDs in the information body to the *N* consecutive registers, or to the register, pointed to by the BASE_INDEX field of the packet header.

Format

Ordinal	Field Name
1	[HEADER]
2	[REG_DATA_1]
3	[REG_DATA_2]
…	…
N+1	[REG_DATA_N]

Header Fields

Bit(s)	Field Name	Description
14:0	BASE_INDEX	The BASE_INDEX[14:0] correspond to byte address bits [16:2]. The BASE_INDEX is the DWORD Memory-mapped address. This field width supports a memory map up to 32K DWORDs (128K Bytes).
15	ONE_REG_WR	0:- Write the data to N consecutive registers. 1:- Write all the data to the same register.
29:16	COUNT	Count of DWORDs in the information body. Its value should be N-1 if there are N DWORDs in the information body.
31:30	TYPE	Packet identifier. It should be zero.

Note: **Symbol ':-' reads "defined as."**

Information Body

Bit(s)	Field Name	Description
31:0	REG_DATA _x	The bits correspond to those defined for the relevant register. Note the suffix x of REG_DATA_x stands for an integer ranging from 1 to N.

Comment

The use of this packet requires the complete understanding of the registers to be written.

4.2 TYPE-2 PACKET

Functionality

This is a filler packet. It has only the header, and its content is not important except for bits 30 and 31. It is used to fill up the trailing space left when the allocated buffer for a packet, or packets, is not fully filled. This allows the CP to skip the trailing space and to fetch the next packet.

Format

Ordinal	Field Name
1	[HEADER]

Header fields

Bit(s)	Field Name	Description
29:0	Reserved	Reserved
31:30	TYPE	Packet identifier. It should be 2.

4.3 TYPE-3 PACKET

Functionality

Carry out the operation indicated by field IT_OPCODE.

Format

Ordinal	Field Name
1	[HEADER]
2	{IT_BODY}

Header fields

Bit(s)	Field Name	Description
0	PREDICATE	Predicated version of packet when bit 0 is set.
7:1	Reserved	This field is undefined, and is set to zero by default. ***** 3D Packet Additions After Chip Release must Set bit 7.***
15:8	IT_OPCODE	Operation to be carried out. See section 6 for details.
29:16	COUNT	Number of DWORDs -1 in the information body. It is N-1 if the information body contains N DWORDs.
31:30	TYPE	Packet identifier. It should be 3.

Information Body

The information body IT_BODY will be described extensively in the following sections.

5 Packet Restrictions

5.1 Packet Location Restrictions (

The following lists the restrictions for the usage of the packets:

1. Real-Time Packet Limitations:
   a. SET_STATE – Allowed in a Real-Time Stream, but no pointer matching occurs. All the sub-blocks are unconditionally fetched if not disabled. The vertex and pixel shader pointers are ignored. See Im_Load and Im_Load_Immediate packets for loading real-time shader code.
   b. IM_LOAD – Allowed in a Real-Time Stream, but no pointer match occurs. The shader code is unconditionally fetched.
   c. SET_CONSTANT – Allowed in a Real-Time Stream, but no write-back to memory occurs.
   d. LOAD_CONSTANT_CONTEXT – Allowed in a Real-Time Stream, but no matching occurs. The constant data is unconditionally fetched.

2. Packets Not Allowed in a Real-Time Stream:
   a. Any 2D Packets
   b. DRAW_INDX
   c. 3D_DRAW_INDX_2
   d. WAIT_FOR_IDLE -- No NQ WAIT function in RBBM for Real-Time Streams
   e. VIZ_QUERY
   f. INDIRECT_BUFFER -- Includes Multi-Pass Operations
   g. INDIRECT_BUFFER_PFD
   h. MPEG_PACKET
   i. IB_PREAMBLE – As a Result of No INDIRECT_BUFFER
   j. INVALIDATE_STATE – No “Last Pointers” for Real-Time
   k. EVENT_WRITE

3. Packets Not Allowed in Indirect Buffer #2
   a. INDIRECT_BUFFER – Not a 3rd Level of Command Buffers
   b. Any 2D Packet (Requires a 2D Indirect Buffer Fetch)

4. Packets not allowed in a 2D stream:
   a. LOAD_CONSTANT_CONTEXT – Used for 2D-to-3D mode switch determination.
   b. SET_STATE – Used for 2D-to-3D mode switch determination.

5. Only the following packets can be used to load the instruction memory: Set_State, Im_Load, and Im_Load_Immediate. The use of Type-0 packets to load instruction memory is not allowed, as the Driver does not have visibility to the pointers that the CP maintains within its microcode.

6. The following methods can be used to initiate indirect buffers:
   a. Type-0 Packets – Must be a single Type-0 packet where the BASE_INDEX in the header is either the CP_IB1_BASE or CP_IB2_BASE register and it also writes the buffer size register. The Header’s COUNT field needs to be 0x01 and the Header’s ONE_REG_WR field needs to be ‘0’.
   b. INDIRECT_BUFFER Packet – This is the preferred mode in CRAYOLA. The Indirect Buffer packet includes a “multi-pass” control flag that allows the indirect buffer to be fetched/executed more than once.

5.2 Restrictions for Packet Additions after Chip Release (Updated: 06-17-2002)

The programmability of the CP allows the addition of PM4 packets after the chip is produced.
Because 2D packet translation is performed in the Command Processor, the following restrictions are imposed on new packets:

1. New Type-3 2D packets must have bit 15 set in the Header (i.e. GMC Processing Required).

6 Definition of Type-3 Packets

Type-3 packets have a common format for their headers. However, the size of their information body may vary depending on the value of field IT_OPCODE. The size of the information body is indicated by field COUNT. If the size of the information is *N* DWORDs, the value of COUNT is *N-1*. In the following packet definitions, we will describe the field IT_BODY for each packet with respect to a given IT_OPCODE, and omit the header. The MSB of the IT_OPCODE identifies whether this packet requires the GUI_CONTROL field (described later). A "1" in the MSB of the IT_OPCODE indicates that GUI control is required. A "0" in the MSB of the IT_OPCODE indicates that the GUI_CONTROL should be omitted.

Opcode assignment POR: For CRAYOLA, all packets that are used from prior designs will keep their existing opcodes. All new packets that are being defined for CRAYOLA will re-use opcodes from packets that are "retired" (i.e. packets have not been used since before Rage 6).

6.1 TYPE-3 PACKET SUMMARY

Packet Name	IT_OPCODE	Description	Status
NOP	0x10	Skip N DWORDs to get to the next packet.	Supported
PAINT	0x91	Paint a number of rectangles with a color brush.	Supported
SMALL_TEXT	0x93	Draw a string of small characters on the screen.	Supported
BITBLT	0x92	Copy a source rectangle to a destination rectangle.	Supported
HOSTDATA_BLT	0x94	Draw a string of large characters on the screen, or copy a number of bitmaps to the video memory.	Supported
POLYLINE	0x95	Draw a polyline (lines connected with their ends).	Supported
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
POLYSCANLINES	0x98	Draw polyscanlines or scanlines.	Supported
NEXTCHAR	0x19	Print a character at a given screen location using the default foreground and background colors.	Supported
PLY_NEXTSCAN	0x1D	Draw polyscanlines using current settings.	Supported
SET_SCISSORS	0x1E	Set up scissors.	Supported
[redacted]	[redacted]	[redacted]	
PAINT_MULTI	0x9A	Paint a number of rectangles on the screen with one color. The difference between this function and PAINT is the representation of parameters.	Supported
BITBLT_MULTI	0x9B	Copy a number of source rectangles to destination rectangles of the screen respectively.	Supported
TRANS_BITBLT	0x9C	2D transparent bitblt operation.	Supported
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
WAIT_FOR_IDLE	0x26	Wait for the IDLE state of the engine.	Supported
LOAD_PALETTE	0x2C	Load a palette for 2D scaling.	Supported
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
3D_DRAW_INDX_2	0x36	Same as 3D_DRAW_INDX, but without VAP_VTX_FMT	Modified: Draw_Initiator replaces Vf_Cntl.

Packet Name	IT_OPCODE	Description	Status
MPEG_INDEX	0x3A	MPEG Packed Register Writes and Index Generation	Format Redefined for Crayola
REG_RMW	0x21	Register Read/Modify/Write	New for Crayola
DRAW_INDX	0x22	Initiate Fetch of Index Buffer	New for Crayola
VIZ_QUERY	0x23	Begin/End Initiator for Viz Queury Extent Processing	New for Crayola
LOAD_EXECUTE	0x24	Load Custom ME Code and Execute	Experimental Use Only.
SET_STATE	0x25	Fetch State Sub-Blocks and Initiate Shader Code DMAs	New for Crayola
IM_LOAD	0x27	Load Sequencer Instruction Memory (Pointer-Based)	New for Crayola
IM_LOAD_IMMEDIATE	0x2B	Load Sequencer Instruction Memory (Code Embedded in Packet)	New for Crayola
SET_CONSTANT	0x2D	Load Constant into Chip and to Memory	New for Crayola
CONST_PREFETCH	0x49	Internal Packet Used Only by CP	New for Crayola
LOAD_CONSTANT_CONTEXT	0x2E	Load Constants from a Location in Memory	New for Crayola
INVALIDATE_STATE	0x3B	Selective Invalidation of State Pointers	New for Crayola
WAIT_REG_MEM	0x3C	Wait Until a Register or Memory Location is a Specific Value.	New for Crayola
WAIT_REG_EQ	0x52	Wait Until a Register Location is equal to a Specific Value.	New for Crayola
WAIT_REG_GTE	0x53	Wait Until a Register Location is greater than or equal to a Specific Value.	New for Crayola
MEM_WRITE	0x3D	Write DWORD to Memory For Synchronization	New for Crayola
COND_MEM_WRITE	0x45	Conditional Write to Memory	New for Crayola
EVENT_WRITE	0x46	Generate an Event that Creates a Write to Memory when Completed	New for Crayola
EVENT_WRITE_SHD	0x58	Generate a VS|PS_Done Event.	Supported
EVENT_WRITE_CFL	0x59	Generate a Cach Flush Done Event.	Supported
EVENT_WRITE_SER	0x5A	Generate a Screen Extent Report Event.	Supported
EVENT_WRITE_ZPD	0x5B	Generate a Z-Pass Done Event.	Supported
REG_TO_MEM	0x3E	Reads Register in Chip and Writes to Memory	New for Crayola
INDIRECT_BUFFER	0x3F	Indirect Buffer Dispatch – Pre-fetch parser uses this packet type in determining to pre-fetch the indirect buffer.	Supported
INDIRECT_BUFFER_PFD	0x37	Indirect Buffer Dispatch – Same as Indirect_Buffer, but init is pipelined.	Supported
CP_INTERRUPT	0x40	Generate Interrupt from the Command Stream	New for Crayola
HOSTDATA_BLT2	0xC1	Same as HOSTDATA_BLT, but one BIG_CHAR and CP Skips Data	New for Crayola
HOSTDATA_BLT_PNTR	0xC2	Same as HOSTDATA_BLT, but embedded data is replaced with a pointer.	New for Crayola
IB_PREAMBLE	0x43	Preamble Packet for IBs for Error Checking	New for Crayola
IB_PREFETCH_START	0x16	Internal Packet Used Only by CP	New for Crayola
IB_PREFETCH_END	0x17	Internal Packet Used Only by CP	New for Crayola
SUB-BLK_PREFETCH	0x1F	Internal Packet Used Only by CP	New for Crayola
INSTR_PREFETCH	0x20	Internal Packet Used Only by CP	New for Crayola
INSTR_MATCH	0x47	Internal Packet Used Only by CP	New for Crayola
MICRO_PREFETCH	0x44	Internal Packet Used Only by CP	Experimental Use Only.
INCR_UPDATE_STATE	0x55	Internal Packet Used Only by CP	New for Crayola
INCR_UPDATE_CONST	0x56	Internal Packet Used Only by CP	New for Crayola
INCR_UPDATE_INSTR	0x57	Internal Packet Used Only by CP	New for Crayola
ME_INIT	0x48	Initialize CP's Micro Engine	New for Crayola
FIX2FLT_REG	0x4D	Convert Fixed Integer Value to Float (0.0 to 255.0) and Write to Register	New for Crayola
GRADFILL	0xCA	Draw Gradient Filled Primitive	New for Crayola
ALPHABLEND	0xCB	Do Alpha Blending Blit	New for Crayola
AAFONT	0xCC	Draw an AA font string.	New for Crayola
2D_ENDIAN_MODE	0x4E	Update the 2D SRC and DST Endian Swapping Controls	New for Crayola
DRAW_2D_COHER_RECT	0xFF	Draw 2D Coherency Rectangle Maintained by CP (Debug Use Only)	New for Crayola
MEM_WRITE_CNTR	0x4F	Write CP_PROG_COUNTER value to memory	New for Crayola
SET_BIN_MASK	0x50	Sets the 64-bit BIN_MASK register in the PFP	New for Crayola
SET_BIN_SELECT	0x51	Sets the 64-bit BIN_SELECT register in the PFP	New for Crayola

6.2 2D PACKETS -- LEGACY

The information body IT_BODY of 2-D packets may have the following format:

Ordinal	Field Name
1	{SETTINGS}
2	{DATA_BLOCK}

SETTINGS

This field consists of 2 sub-fields, GUI_CONTROL and SETUP_BODY.

Ordinal	Field Name
1	[GUI_CONTROL]
2	{SETUP_BODY}

- **SETTINGS.GUI_CONTROL**

This field will be used to setup the register DP_GUI_MASTER_CNTL, and it also decides the content of SETTINGS.SETUP_BODY.

The CP has 2D default registers. If the default is requested in the GUI_CONTROL, the CP will use the values from these registers, otherwise the data must be supplied in the command packet. Note that values that override the defaults will not overwrite the default values. The default values are loaded into the CP via Type-0 packets and/or direct register writes by the Driver. They are stored in the same format as they are written.

Where the CP is writing constant data, the fields constant (ALU or Texture Fetch) do not have to be modified will have default values embedded in the CP microcode.

Refer to the CP's power-up initialization section of the CP unit spec for details the minimum 2D initialization.

The figure below illustrates the meaning of some of the fields related to 2D packets:

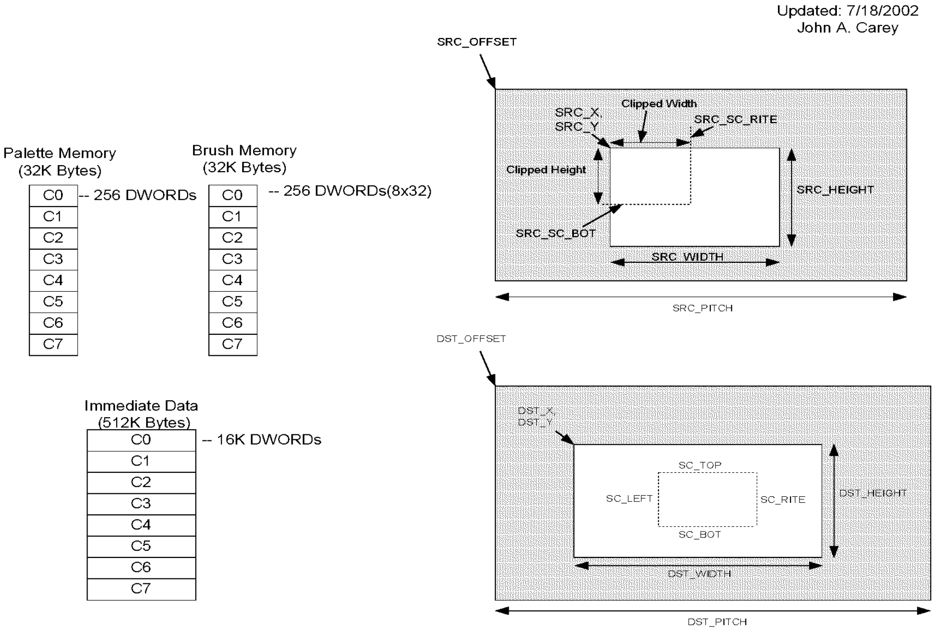


Figure 6-1: 2D Field Illustration

Bit(s)	Field Name	Description	Notes
0	SRC_PITCH_OFF	The bit controls the pitch and offset of the blitting source. 0:- Use the DEFAULT_PITCH_OFFSET from CP's 2D Defaults Memory, and no datum [SRC_PITCH_OFFSET] is supplied in SETUP_BODY. 1:- Use the datum [SRC_PITCH_OFFSET] supplied in SETUP_BODY to set up a new pitch offset.	CP writes the Pitch and Tile to DW#0 of the T0 constant. It also sets the "2D Texture" flag in T0 if it is not a Small_Text or HostData_Blt* packet. The CP writes the Offset[21:0] value to bits [31:12] of DW#1 of the T0 constant. The CP also calculates and writes the DATA_FORMAT field in DW#1 of the T0 constant.
1	DST_PITCH_OFF	The bit controls the pitch and offset of the blitting destination. 0:- Use the DEFAULT_PITCH_OFFSET from CP's 2D Defaults Memory, and no datum [DST_PITCH_OFFSET] is supplied in SETUP_BODY. 1:- Use the datum [DST_PITCH_OFFSET] supplied in SETUP_BODY. The pitch may mean the bitmap pitch and the offset may points the off-screen area of the video memory.	CP writes this value to the RB_2D_BASE_PITCH register in the RB. The CP does not change the format of the value that it writes to the RB.
2	SRC_CLIPPING	This bit controls the clipping parameters of the blitting source. 0:- Use the default clipping parameters from the CP_DEFAULT_SC_BOTTOM_RIGHT register and no relevant clipping data supplied in SETUP_BODY. 1:- Use datum [SRC_SC_BOTTOM_RIGHT] supplied in SETUP_BODY to set up the bottom and right edges of the clipping rectangle.	The CP stores the DEFAULT_SC_BOTTOM_RIGHT in the Micro Engine's Scratch Memory for use with source clipping (BitBLT, BITBLT_MULTI, and Trans_Blt packets).
3	DST_CLIPPING	This bit controls the clipping parameters of the blitting destination. 0:- Use the default clipping parameters from the CP_DEFAULT_SC_BOTTOM_RIGHT register and no relevant clipping data supplied in SETUP_BODY. Top/Left is set to (0,0). 1:- Use data [SC_TOP_LEFT] and [SC_BOTTOM_RIGHT] supplied in SETUP_BODY to set up a new clipping rectangle.	The CP writes (0,0) to the PA_SC_SCREEN_SCISSOR_TL register and the DEFAULT_SC_BOTTOM_RIGHT value to the PA_SC_SCREEN_SCISSOR_BR register in the PA.
7:4	BRUSH_TYPE	Types of brush used in drawing. The type code determines how to supply data to the sub-field BRUSH_PACKET in SETUP_BODY. See detailed definition of BRUSH_TYPE in the following.	The CP writes the DATA_FORMAT field of DW#1 of the T1 constant based on the Brush Type. The CP also writes the external memory base address of the brush data for the assigned Brush offset to the BASE_ADDRESS field in DW#1 of the T1 constant. The FG and BG colors that are associated with the brush are unpacked, converted to floating point, and are written to the Pixel ALU constants C0 and C1 respectively. See the Brush Packet Content section later in this document. The raster data associated with the brush is written to external memory at a pitch of 32 texels This means that the brush conversion function writes the first 8 values then skips the next 24 DWORDs to get to the next line. See the Brush Table later in this document for details. The CP also sets the following brush Booleans depending on the BRUSH_Type: B0, B1, B2, B5. See the Brush Table later in this document for details.
11:8	DST_TYPE	The pixel type of the destination. 0--1 :- (reserved) 2 :- 8 bpp pseudocolor 3 :- 16 bpp aRGB 1555 4 :- 16 bpp RGB 565 5 :- reserved 6 :- 32 bpp aRGB 8888 7 :- 8 bpp RGB 332 **(Not Supported in CRAYOLA..Not since R128)** 8 :- Y8 greyscale 9 :- RGB8 greyscale (8 bit intensity, duplicated for all 3 channels. Green channel is used on writes) 10 :- (reserved) 11 :- YUV 422 packed (VYUY)	CP writes this GUI Control DWORD to the RB_2D_FORMAT_ROP3 register in the Renderer Backend. This takes care of initializing the DST_TYPE and ROP3 code. The field is also used in determining the DATA_FORMAT field of that is written to the T0 constant.

		12 :- YUV 422 packed (YVYU) 13 :- (reserved) 14 :- aYUV 444 (8:8:8:8) 15 :- aRGB4444 (Intermediate format only. Not understood by the Display Controller) **Note:** Choices 7-15 are only supported by BitBlt, BitBlt_Multi, Trans_BitBlt, HostData_Blt, HostData_Blt2, and HostData_Blt_Pntr.	
13:12	SRC_TYPE (See bit 27 for 3rd bit)	The field indicates the pixel type of blitting source. 0:- The source data type is mono opaque, and the fore- and back-ground colors need to be redefined. 1:- The source data type is mono transparent, and only the foreground color needs to be redefined. 2:- Reserved. 3:- The source pixel type is the same as that given in field DST_TYPE. If bit 27 (SRC_TYPE) is one then the following sources are available: 4:- 4bpp source **(Not Supported in CRAYOLA)** 5:- 8bpp source (Boolean LUT in Shader Code) 6:- 32 bpp source (Boolean LUT in Shader Code) 7:- 64 bpp Obuffer blit **(Not Supported in CRAYOLA)** **Note:** In mode 3, all source formats are supported as the destination formats, where they are supported as the source.	CP analyses the SRC_TYPE[2:0] and as a result sets the following Booleans: Set B3 (Src Mask) if SRC_TYPE=1 (Mon Transparent) Set B6 (LUT) if SRC_TYPE=5 or 6 Set B7 (LUT8) if SRC_TYPE=5
14	BYTE_PIX_ORDER	The bit decides the order of bits (or pixels) in DWORD to be consumed. Only applicable to the monochrome mode. 0 :- Bits to be consumed from the Most Significant Bit (MSB) to the Least Significant Bit (LSB). 1 :- Bits to be consumed from LSB to MSB.	The CP uses the BYTE_PIX_ORDER field in determining the DATA_FORMAT field of the Texture T0 constant.
15	DEFAULT_SEL	Selects which set of default values to use when processing the GMC. DEFAULT_PITCH_OFFSET vs DEFAULT2_PITCH_OFFSET DEFAULT_SC_BOTTOM_RIGHT vs. DEFAULT2_SC_BOTTOM_RIGHT	1 => Use Default #2
23:16	ROP3	This field tells the GUI engine how the raster operation to be carried out. The code of this field follows the ROP3 code defined by Microsoft. See WIN31 DDK for reference.	See the DST_TYPE field defined above.
26:24	DP_SRC_SOURCE	The field indicates where the source data come from. 0,1 :- Reserved 2 :- Loaded from the video memory (rectangular trajectory) 3 :- Loaded through the HOSTDATA registers (linear trajectory) 4 :- Loaded through the HOSTDATA registers (linear trajectory & byte-aligned) Note that during 3D/Scale Operations (whenever SCALE_3D_FCN@MISC_3D_STATE_REG is non-zero), this field is ignored and data is always loaded from the 3D/Scaler pipeline.	DP_SRC_SOURCE = 4 only supported for color sources. CRAYOLA does not support DP_SRC_SOURCE = 4 for Monochrome sources.
27	SRC_DATATYPE	Third bit of SRC_TYPE	See bits 13:12
28	CLR_FCN_DIS	0 :- No change to CLRCMP_FCN_SRC and CLRCMP_FCN_DST fields. 1 :- Clear CLRCMP_FCN_DST and CLRCMP_FCN_SRC to zero.	If set the CP will write the RB_CLRCMP_CONROL register to zero. This will clear the CLRCMP_FCN_SRC and CLRCMP_FCN_DST fields. The CLRCMP_FCN_SEL will also be cleared, but this is a “don’t care” in this case.
29	Reserved	Reserved	Reserved
30	WR_MSK_DIS	0 :- No Change to the RB_PLANE_MASK and RB_CLRCMP_MSK_LO 1 :- Set RB_PLANE_MASK to 0xffffffff RB_CLRCMP_MSK_LO according to pixel type. Note: RB_CLRCMP_MSK_HI is zero’d in the 2D Indirect Buffer.	If set, the CP will write the RB_PLANE_MASK and RB_CLRCMP_MSK_LO registers in the Renderer Backend (RB).

31	BRUSH_FLAG	This field indicates whether there is a field BRUSH_Y_X field in the SETTINGS.SETUP_BODY.
		0:- No such a field in SETTINGS.SETUP_BODY.
		1:- There is a field in SETTINGS.SETUP_BODY.
			If the BRUSH_FLAG is set, the BRUSH X and Y values are unpacked, converted to floating point (0.0 to 31.0), and written to the Vertex ALU C0.xy constant:
			Brush_X = C0.x
			Brush_Y = C0.y
			The Brush_Offset is also initialized from bits 20:16 in the CP.

- **SETTINGS.SETUP_BODY**

This field may contain the following sub-fields. Their presence depends on the bits 0-7 of **SETTINGS.GUI_CONTROL.**

Ordinal	Field Name	Description
1	[SRC_PITCH_OFFSET]	Bit 31: Select between untiled(0) and tiled (1) Bits 30:22 Pitch in (Pixels/32) {Up to 16,384 Pixels} bits 21:0 Offset in units of 1KB, 0 to 4GB-1K ***Note: In CRAYOLA, surfaces are 4KByte aligned so bits 1:0 of the offset need to be zero.***
2	[DST_PITCH_OFFSET]	Bit 31: Select between untiled(0) and tiled (1) Bits 30:22 Pitch in (Pixels/32) {Up to 16,384 Pixels} Bits 21:0 Offset in units of 1KB, 0 to 4GB-1K ***Note: In CRAYOLA, surfaces are 4KByte aligned so bits 1:0 of the offset need to be zero.***
3	[SRC_SC_BOTTOM_RIGHT]	The parameters are used to setup the clipping area of the source. The implied coordinate of the top-left corner of the clipping rectangle is the same as the source. ***Both parameters have the range: -8,192 to +8,191.*** [15:0] :- X-coordinate of the right edge of the clipping rectangle (in number of pixels). [31:16] :- Y-coordinate of the bottom edge of the clipping rectangle (in number of scan lines).
4	[SC_TOP_LEFT] [SC_BOT_RITE]	The parameters are used to setup the clipping area of destination. ***See the PA_SC_SCREEN_SCISSOR_TL and PA_SC_SCREEN_SCISSOR_BR definitions in the PA register spec for the format of the SC_TOP_LEFT and SC_BOT_RITE values respectively.***
5	[BRUSH_PACKET]	The content of this field is determined by field `SETTINGS.GUI_CONTROL.BRUSH_TYPE`. See the following table for the possible content.
6	[BRUSH_Y_X]	[2:0] :- X-coordinate for brush alignment (0.0 to 7.0). [10:8] :- Y-coordinate for brush alignment (0.0 to 7.0). [20:16] :- BRUSH_X Initial Value (0.0-31.0). The CP uses the BRUSH_X Initial Value as a starting offset for the brush when processing the first line segment of a POLY_LINE PM4 packet. The brush for lines is 32x1.

- **SETTINGS.SETUP_BODY.BRUSH_PACKET**

All Brush Types but 6 and 7 are not available for lines, and 6 and 7 are only usable for lines.

The CP sets Booleans based on the Brush Type. The Brush Booleans are defined as follows:

B0 – Solid Brush
B1 – Mono Brush
B2 – Brush Mask
B5 – Brush Present

For Monochrome, the brush data is converted to 32bpp and written to external memory. Note that the pitch of the brush data is set to 32, so after the 8 DWORDs are written, the CP skips 24 DWORDs to the next brush line in external memory.

BRUSH_TYPE	Description of the brush	Boolean Settings	BMP Conversion & Source Type	Packet size	Packet content
0	A 8 x 8 mono pattern with the foreground and background colors specified in the packet. Here the matrix is represented in the format *column-by-row*.	B0 = 0 B1 = 1 B2 = 0 B5 = 1	CP converts to 32bpp for a total of 64 DWORDs. Conversion is MSB to LSB within each byte, where the MSB of the byte is the left-most pixel. DATA_FORMAT = 6 in the DW #1 of the T1 Const.	4 DWORDs	[BKGRD_COLOR] [FRGRD_COLOR] [MONO_BMP_1] [MONO_BMP_2]
1	A 8 x 8 mono pattern with the foreground color specified in the packet and the background color the same as that of the area to be painted. Background left alone.	B0 = 0 B1 = 1 B2 = 1 B5 = 1	CP converts to 32bpp for a total of 64 DWORDs. Same conversion as Brush_Type 0. DATA_FORMAT = 6 in the DW #1 of the T1 Const.	3 DWORDs	[FRGRD_COLOR] [MONO_BMP_1] [MONO_BMP_2]
2	Reserved	N/A	N/A	N/A	Not Applicable
3	Reserved	N/A	N/A	N/A	Not Applicable
4	Reserved	N/A	N/A	N/A	Not Applicable
5	Reserved	N/A	N/A	N/A	Not Applicable
6	A 32 x 1 mono pattern with the foreground and background colors specified in the packet. This pattern corresponds to the PEN of Win95 DDK and is only usable for lines.	B0 = 0 B1 = 1 B2 = 0 B5 = 1	CP converts to 32 bpp. Same conversion as Brush_Type 0. The CP writes the converted brush *twice* for a total of 64 DWORDs, where the second 32 DWORDs are identical to the first 32 DWORDs. DATA_FORMAT = 6 in the DW #1 of the T1 Const.	3 DWORDs	[BKGRD_COLOR] [FRGRD_COLOR] [MONO_BMP_1]
7	A 32x1 mono pattern with the foreground color specified in the packet and the background color the same as that of the area to be painted. This is PEN as well. And is only usable for lines.	B0 = 0 B1 = 1 B2 = 1 B5 = 1	CP converts to 32 bpp. Same conversion as Brush_Type 0. The CP writes the converted brush *twice* for a total of 64 DWORDs, where the second 32 DWORDs are identical to the first 32 DWORDs. DATA_FORMAT = 6 in the DW #1 of the T1 Const.	2 DWORDs	[FRGRD_COLOR] [MONO_BMP_1]
8	Reserved	N/A	N/A	N/A	Not Applicable
9	Reserved	N/A	N/A	N/A	Not Applicable
10	A 8x8 color pattern. The pixel type = DST_TYPE	B0 = 0 B1 = 0 B2 = 0 B5 = 1	CP just writes 16*N DWORDs of brush to external memory. DATA_FORMAT in the DW #1 of the T1 Const equals the DST_TYPE.	16* N DWORDs, where N stands for the number of bytes per pixel with exception that a 24-BPP pixel is still represented by 4 bytes.	[COLOR_BMP_1] [COLOR_BMP_2] ... [COLOR_BMP_16*N]
11	Reserved	N/A	N/A	N/A	Not Applicable
12	Reserved	N/A	N/A	N/A	Not Applicable
13	Use the color specified in the packet as the	B0 = 1	No Brush Written to Memory.	1	[FRGRD_COLOR]

BRUSH_TYPE	Description of the brush	Boolean Settings	BMP Conversion & Source Type	Packet size	Packet content
	solid (plain) color for the brush (i.e. a color brush without a pattern).	B1 = 0 B2 = 0 B5 = 0	DATA_FORMAT = 6 in the DW #1 of the T1 Const.	DWORD	
14	Use the color specified in the packet as the solid (plain) color for the brush (i.e. a color brush without a pattern). Used for Lines.	B0 = 1 B1 = 0 B2 = 0 B5 = 0	No Brush Written to Memory. DATA_FORMAT = 6 in the DW #1 of the T1 Const.	1 DWORD	[FRGRD_COLOR]
15	No brush used.	B0 = 0 B1 = 0 B2 = 0 B5 = 0	N/A	0	N/A

Brush Packet Content

Field Name	Description
[BKGRD_COLOR]	The background color of the text in the DST_TYPE format. Formats 2 (8bpp), 3 (16bpp aRGB1555), 4 (16bpp RGB565), and 6 (32bpp aRGB8888) are supported. If the DST_TYPE is not 2, 3, 4, 6, then aRGB8888 is assumed. The CP converts each of these integer terms to floating point (0.0 to 255.0). For the 8bpp format, the CP replicates the same value for the Red, Green, and Blue channels and sets Alpha to 255. For the 16bpp RGB565 format, the Alpha term is set to zero. Bits [7:0] :- Intensity of Blue – Written to Pixel ALU C1: Logical #0, DW#6 Bits [15:8] :- Intensity of Green – Written to Pixel ALU C1: Logical #0, DW#5 Bits [23:16] :- Intensity of Red – Written to Pixel ALU C1: Logical #0, DW#4 Bits [31:24] :- Reserved – Written to Pixel ALU C1: Logical #0, DW#7 (ALPHA Term) Upper bits of "Reserved" field assumed to be zero for float conversion.
[FGGRD_COLOR]	The foreground color of the text in the DST_TYPE format. Formats 2 (8bpp), 3 (16bpp aRGB1555), 4 (16bpp RGB565), and 6 (32bpp aRGB8888) are supported. If the DST_TYPE is not 2, 3, 4, 6, then aRGB8888 is assumed. The CP converts each of these integer terms to floating point (0.0 to 255.0). For the 8bpp format, the CP replicates the same value for the Red, Green, and Blue channels and sets Alpha to 255. For the 16bpp RGB565 format, the Alpha term is set to zero. The foreground color of the text in the DST_TYPE format. Bits [7:0] :- Intensity of Blue – Written to Pixel ALU C0: Logical #0, DW#2 Bits [15:8] :- Intensity of Green – Written to Pixel ALU C0: Logical #0, DW#1 Bits [23:16] :- Intensity of Red – Written to Pixel ALU C0: Logical #0, DW#0 Bits [31:24] :- Reserved – Written to Pixel ALU C0: Logical #0, DW#3 (ALPHA Term) Upper bits of "Reserved" field assumed to be zero for float conversion.
[MONO_BMP_x]	Raster data of monochrome pixels. One bit represents one pixel. If the number of pixels for the field is less than 32, the pixels take the lower bits. The remaining bits should be filled with 0's.
[COLOR_BMP_x]	Raster data of color pixels. The representation depends on the pixel type.

DATA_BLOCK

The composition of this field depends on the operation code IT_OPCODE given in the header. Section B.2 gives details of DATA_BLOCK with respect to IT_OPCODE. In the following, the field SETTINGS may appear in the definition of a packet, but will not be described further.

6.2.1 *PAINT (Updated: 07-10-2002)*

Functionality

Paint a number of rectangles with a color brush.

Set geometry base address to the first DWORD of data block, and select the paint vertex shader.

The paint vertex shader fetches two pairs of 16-bits for every vertex output, and outputs a rectangle point sprite with the fetched coordinates.

Pseudocode:

1. Send single DRAW_INITIATOR to the *VGT_DRAW_INITIATOR* register representing all of the paint rectangles in the packet. The primitive is set as a 2D_FILL_RECT_LIST and the NUM_INDICES is computed as follows:
    a. NUM_INDICES = ((COUNT + 1) ÷ 2) * 3

    Where COUNT is the N-1 count of DWORDs after the SETTINGS field is processed.

2. For every paint rectangle, the Micro Engine (ME) computes and outputs the Upper Left, Upper Right, and Lower Left compound indices to the VGT. See the "Specification of the CP's 2D Implementation in CRAYOLA" document for details.
    a. DST_X = LEFT
    b. DST_Y = TOP
    c. WIDTH = RIGHT – LEFT
    d. HEIGHT = BOTTOM - TOP

Format:

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

`DATA_BLOCK`

Ordinal	Field Name	Description
1	[TOP_1 | LEFT_1]	The coordinates of the top-left corner of the 1st rectangle to be painted. LEFT_1: [15:0]:- x-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. TOP_1: [31:16]:- y-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
2	[BOTTOM_1| RIGHT_1]	The coordinates of the bottom-right corner of the 1st rectangle to be painted. RIGHT_1: [15:0]:- x-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. BOTTOM_1: [31:16]:- y-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
…	…	…
2n-1	[TOP_n| LEFT_n]	The coordinates of the top-left corner of the n-th rectangle to be painted.
2n	[BOTTOM_n| RIGHT_n]	The coordinates of the bottom-right corner of the n-th rectangle to be painted.

6.2.2 SMALL_TEXT (Updated: 09-18-2002)

Functionality

Print a string of characters on the screen in the format of the bit-packed Small Glyph.

All characters in the SMALL_TEXT packet are processed within the same context. Because the amount of raster data per character is cannot be determined by the HEADER of the packet, a DRAW_INITIATOR is issued for each of the character.

Note that the Driver will never supply a brush for a Small_Text packet.

Microcode:

1. Unpack and Convert the FRGD_COLOR to floating point and write the values to the ALU C0 constant.
2. Set B9 (Embedded Source) Boolean.
3. If ROP[7:4] == ROP[3:0], then set B4 and C2.x (Source Fetch Booleans).
4. Write the BASE_ADDRESS field in the DW#1 of the Texture T0 with the base address of the small text data in external memory for the assigned context.
5. Write the 2D Booleans to the assigned context.
6. For every Small Text Character:
    a. Calculate and send the UL, UR, LL compound indices to the VGT. See the "Specification of the CP's 2D Implementation in CRAYOLA" document for details.
    b. Send an EVENT_INITIATOR with ID = SC_WAIT_WC to the Scan Converter to tell it to wait for the write confirmation before processing the subsequent draw initiator.
    c. Write the raster data to external memory. The last DWORD is written with a write confirmation.
    d. Send single DRAW_INITIATOR to the *VGT_DRAW_INITIATOR* register. The primitive is set as a 2D_COPY_RECT_LIST_SRC_V0. The *NOT_EOP* flag in the VGT_DRAW_INITIATOR is set for every initiator but the last.
    e. Calculate DST_X, DST_Y, and SRC_OFFSET for the next character's Uper-Left Corner.
        i. $DST\_X_{n+1} = DST\_X_n + \Delta X_{n+1}$ (Accumulated per Character)
        ii. $DST\_Y_{n+1} = BAS\_Y - \Delta Y_{n+1}$ (No Accumulation)
        iii. $SRC\_OFFSET_{n+1} = SRC\_OFFSET_n + (RASTER\_COUNT_n * 32)$ (Raster Count Accumulates per Character)

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[FRGD_COLOR]	The foreground color of the text in the RGBQUAD format. BLUE: [7:0] :- Intensity of the blue component. GREEN: [15:8] :- Intensity of the green component. RED: [23:16] :- Intensity of the red component. Bits [31:24] :- Reserved.
2	[BAS_Y | BAS_X]	The base coordinates of the text rectangle in the screen coordinate system. See the following illustration for details. BAS_X: [15:0] :- X-coordinate. –8,192 to +8,191, Sign Extended to 16 bits. BAS_Y: [31:16] :- Y-coordinate. –8,192 to +8,191, Sign Extended to 16 bits.
3	{SMALLCHAR_1}	The 1st character of the text.
…	…	…
n+2	{ SMALLCHAR _n}	The n-th character of the text, i.e. the last character.

- **DATA_BLOCK.SMALLCHAR_x**

Ordinal	Field Name	Description
1	[H | W |ΔY | ΔX]	The geometry of the bitmap and the deviation of its top-left corner from the base coordinates. ΔX: [7:0] :- Deviation from the BAS_X-coordinate of the preceding glyph. ΔY: [15:8] :- Deviation from the BAS_Y-coordinate to the Upper Left Corner. W: [23:16] :- Width of the character bitmap (Can be Zero for Space Characters). H: [31:24] :- Height of the character bitmap (Can be Zero for Space Characters). ΔX and ΔY have the range: -128 to 127, where the 8th bit is the sign.
2	[RASTER_1]	The 1st DWORD of the mono bitmap data.
...	…	…
m+1	[RASTER_m]	The m-th DWORD of the mono bitmap data.

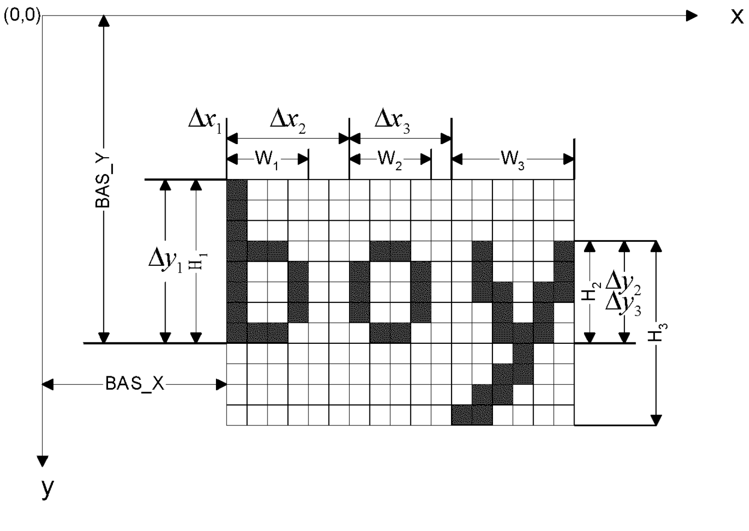
Parameters H, W, ΔY and ΔX

The relationship between the parameters and the reference coordinates BAS_X and BAS_Y is shown in the following figure. In the figure, the starting position of text is at (*BAS_X, BAS_Y*). The actual sizes of characters 'b', 'o' and 'y' respectively are 4×8, 4×5, and 6×9. Therefore, the related parameters are:

$$H_1 = 8, W_1 = 4, \Delta x_1 = 0, \text{ and } \Delta y_1 = +8$$

$$H_2 = 5, W_2 = 4, \Delta x_2 = 6, \text{ and } \Delta y_1 = +5$$

$$H_3 = 9, W_3 = 6, \Delta x_3 = 5, \text{ and } \Delta y_1 = +5$$



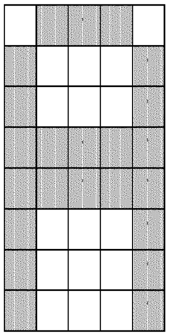
- RASTER_m

Raster_m represents the data block of a mono bitmap. The bitmap represents the raster image of a character

The raster data is mapped as follows:

R400 Small Text Character Packing

Updated: 3/1/2002
John A. Carey



H = 8

W = 5

0	1	1	1	0
1	0	0	0	1
1	0	0	0	1
1	1	1	1	1
1	1	1	1	1
1	0	0	0	1
1	0	0	0	1
1	0	0	0	1

Key = $DW\#_{bit\#}$

0_{07}	0_{06}	0_{05}	0_{04}	0_{03}
0_{02}	0_{01}	0_{00}	0_{15}	0_{14}
0_{13}	0_{12}	0_{11}	0_{10}	0_{09}
0_{08}	0_{23}	0_{22}	0_{21}	0_{20}
0_{19}	0_{18}	0_{17}	0_{16}	0_{31}
0_{30}	0_{29}	0_{28}	0_{27}	0_{26}
0_{25}	0_{24}	1_{07}	1_{06}	1_{05}
1_{04}	1_{03}	1_{02}	1_{01}	1_{00}

DW0: 1100 0110 1111 1111 0110 0011 0111 0100
DW0: C6 FF 63 74

DW1: 0000 0000 0000 0000 0000 0000 0011 0001
DW1: 00 00 00 31

$RASTER\_COUNT_0$ = int [((H * W) + 31) / 32] = ((8 * 5) + 31) / 32 = 2
SRC_X_0 = 0

SRC_X_1 = 64 {$RASTER\_COUNT_0$ * 32}

6.2.3 BITBLT (Updated: 04-14-2003)

Functionality

Copy a source rectangle to a destination rectangle of the screen. It is assumed that the geometry of the destination is identical to its source.

Settings Restrictions:

A BitBlt with a mono opaque source, SRC_TYPE=0, or a mono transparent source, SRC_TYPE=1, and a ROP code set to source copy, 0xCC, is not supported.

Pseudocode:

See microcode description for BITBLT_MULTI. The BITBLT opcode jumps to the same microinstructions as BITBLT_MULTI.

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[SRC_X1 | SRC_Y1]	The coordinates of the top-left corner of the 1st source bitmap. SRC_Y1: [15:0]:- y-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. SRC_X1: [31:16]:- x-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
2	[DST_X1 | DST_Y1]	The coordinates of the top-left corner of the 1st destination. The definition of bits is the same as SRC_X1 and SRC_Y1.
3	[SRC_W1| SRC_H1]	The width and height of the 1st source bitmap, expressed in unsigned integers. SRC_H1: [13:0]:- Height. SRC_W1: [29:16]:- Width.

6.2.4 HOSTDATA_BLT (Updated: 04-14-2003)

Functionality

Copy a number of bit-packed bitmaps to the Video Memory. It can be used to print a string of large characters on the screen or load texture maps.

All BIGCHARs in the HOSTDATA_BLT packet are processed in the same context.

Settings Restrictions:

If the ROP is set to source copy, 0xCC, then the "settings" portion of this packet should not include a brush.

Microcode:

1. Unpack and Convert the FRGD_COLOR to floating point values and write them to the ALU Constant C0.
2. Unpack and Convert the BKGD_COLOR to floating point values and write them to the ALU Constant C1.
3. Set the B9 (Embedded Source) Boolean.
4. If ROP[7:4] == ROP[3:0], then set B4 and C2.x (Source Fetch Booleans).
5. Write the BASE_ADDRESS field of the Texture T0 Constant with the base address of the raster data in external memory where the BIGCHAR data will be written for the assigned context. Bits [31:19] is the base address and bits [18:16] is the current context.
6. For Every Big Character:
    a. Send an EVENT_INITIATOR with ID = SC_WAIT_WC to the Scan Converter to tell it to wait for the write confirmation before processing the subsequent draw initiator.
    b. Write the raster data to external memory. The last DWORD is written with a write confirmation. The next big character in the packet is written to the next consecutive address.
    c. Write the DRAW_INITIATOR to the *VGT_DRAW_INITIATOR* register. The PRIM_TYPE is set to 2D_COPY_RECT_LIST_V0. The NOT_EOP flag in the VGT_DRAW_INITIATOR is set for every initiator but the last. Note also that the raster order is arbitrary, as the source data will not overlap the destination for this primitive.
    d. Calculate the UL, UR, LL compound indices and send them to the VGT. See the "Specification of the CP's 2D Implementation in CRAYOLA" document for details.

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[FRGD_COLOR]	Foreground color in the RGBQUAD format. For mono-to color expansion only. The field is ineffective if field SRC_TYPE at SETTINGS.GUI_CONTROL is set to a type other than mono opaque or mono transparent (0 or 1).
2	[BKGD_COLOR]	Background color in the RGBQUAD format. For mono-to color expansion only. The field is ineffective if field SRC_TYPE at SETTINGS.GUI_CONTROL is set to a type other than mono opaque or mono transparent (0 or 1).
3	{BIGCHAR_1}	Data block of the 1st character.
...	...	...
m+2	{BIGCHAR _m}	Data block of the m-th character.

- **`DATA_BLOCK.BIGCHAR_x`**

Ordinal	Field Name	Description
1	[BaseY | BaseX]	The coordinate of the top-left corner of the character's bitmap. BaseX: [15:0] :- X-coordinate. –8,192 to +8,191, Sign Extended to 16 bits. BaseY: [31:16] :- Y-coordinate. –8,192 to +8,191, Sign Extended to 16 bits.
2	[HEIGHT | WIDTH]	The geometry of the bitmap. WIDTH: [13:0] :- Width of the bitmap. HEIGHT: [29:16] :- Height of the bitmap.
3	[NUMBER[13:0]]	The number of DWORDs in the bitmap. It should be *m* in this case. The HostData_Blt packet is limited to 16384 DWORDs by the COUNT field in the packet header. The maximum RASTER DWORDs for a HostData_Blt packet with a single BIGCHAR is 0x3F97. ***A value of zero is acceptable, but this is assumed to be the end of the packet.***
4	[RASTER_1]	The 1st DWORD of the mono bitmap data.
...	…	…
m+3	[RASTER_m]	The m-th DWORD of the mono bitmap data.

6.2.5 POLYLINE (Updated: 09-10-2002)

Functionality

Draw a polyline specified by a set of coordinates (x_0, y_0), (x_1, y_1), ..., (x_n, y_n), where coordinate (x_0, y_0) is the beginning of the polyline, and coordinate (x_n, y_n) is the end.

Pseudocode:

```
1. The CP calculates the number of draw packets that will be generated for the POLYLINE packet. A maximum of 127 line segments (128 Indices) per draw packet is allowed so as not to overflow the brush offset calculation. The number of draw packets is computed as follows:
    a. Number Draw Packets = COUNT ÷ 128
2. Set the Line (B12) Boolean. This applies to all draw packets that are submitted. Note that the CP does not change the state context between draw packets within a POLYLINE packet.
3. Write the 2D Booleans to the assigned context.
4. For each draw packet:
    a. Compose the DRAW_INITIATOR and write it to the *VGT_DRAW_INITIATOR* register with the primitive type set to 2D_LINE_STRIP.
    b. If not the last draw packet, the NUM_INDICES field in the DRAW_INITIATOR is set to 129 (0x81). Note that the N-1st indice is repeated at the subsequent draw packet. The first draw packet will have the 0th indice written twice.
    c. For the last draw packet, the NUM_INDICES field in the DRAW_INITIATOR is set to COUNT+2. This includes either the 0th indice replicated twice or the N-1st indice from the previous draw initiator from this packet.
    d. Calculate and send the VGT the Line's compound indices. See the "Specification of the CP's 2D Implementation in CRAYOLA" document for further details.
        i. DST_X:
            1. First line segment of 2nd and Subsequent Draw Packets within POLYLINE packet:
                a. $DST\_X_n = DST\_X_{n-1}$
            2. Otherwise:
                a. $DSX\_X_n = X_n$
        ii. DST_Y:
            1. First line segment of 2nd and Subsequent Draw Packets within POLYLINE packet:
                a. $DST\_Y_n = DST\_Y_{n-1}$
            2. Otherwise:
                a. $DSX\_Y_n = Y_n$
        iii. BRUSH_OFFSET = Accumulated offset into the line brush pattern.
            1. First line segment of first draw packet generated:
                a. BRUSH_OFFSET = Bits 20:16 in the BRUSH_Y_X of the GMC.
            2. First line segment of 2nd and Subsequent Draw Packets generated:
                a. $BRUSH\_OFFSET_n = BRUSH\_OFFSET_{n-1}$
            3. Second line segment of 2nd and Subsequent Draw Packets generated:
                a. $BRUSH\_OFFSET_n = BRUSH\_OFFSET_{n-1}$ & 0x3F + MAX(|ΔX|, |ΔY|)
            4. Otherwise:
                a. $BRUSH\_OFFSET_n = BRUSH\_OFFSET_{n-1}$ + MAX(|ΔX|, |ΔY|)
    e. Update DWORD Count: COUNT ← COUNT - 128
```

Note that the VGT generates the "Line Strips" for the POLYLINE packet. The CP just sends a single "compound index" per line coordinate.

Write Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[Y0 | X0]	The starting coordinate of the polyline. X0: [15:0] :- X-component of the coordinate. –8,192 to +8,191, Sign Extended to 16 bits. Y0: [31:16]:- Y-component. –8,192 to +8,191, Sign Extended to 16 bits.
2	[Y1 | X1]	The 2nd coordinate of the polyline. Definition of bits is the same as above.
...	…	…
n+1	[Yn | Xn]	The ending coordinate of the polyline. Definition of bits is the same as above.

6.2.6 POLYSCANLINES (Updated: 01-29-2003)

Functionality

Draw a number of scanlines. The number can be one.

All scanlines in the POLYSCANLINES packet are processed within the same context. Because the NUM_LINEs per SCAN_n cannot be determined by the HEADER of the packet, a DRAW_INITIATOR is issued for each of the SCAN_n subsections.

It is assumed that End_n ≥ Start_n in the microcode that processes this packet.

Pseudocode:

```
1. For Every SCAN_n:
    a. Send single DRAW_INITIATOR to the VGT_DRAW_INITIATOR register representing all of the scanlines in the SCAN_N sub-packet. The primitive is set as a 2D_FILL_RECT_LIST and the NUM_INDICES is computed as follows:
           NUM_INDICES = NUM_LINE * 3
    b. For each scan line within the SCAN_n:
        1. Calculate the UL, UR, LL compound indices and send these to the VGT. See the "Specification of the CP's 2D Implementation in CRAYOLA" document for details.
            1. HEIGHT is applied globally to each of the scanline primitives.
            2. WIDTH = End_n – Start_n.
```

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

`DATA_BLOCK`

Ordinal	Field Name	Description
1	[SCAN_COUNT]	The number of scan subpackets identified by SCAN_x, where x denotes the ordinal number of a SCAN subpacket. A number of zero is acceptable, but no scanlines should follow.
2	{ SCAN_1 }	The 1st scanline/polyscanline.
...	...	...
n+1	{ SCAN_n }	The n-th scanline/polyscanline.

- `DATA_BLOCK.SCAN_x`

Ordinal	Field Name	Description
1	[NUM_LINE[13:0]]	The number of line segments in a polyscanline. Minimum is 0x0001, Maximum is 0x3fff.
2	[HEIGHT | TOP]	TOP: [15:0] :- Y-coordinate of the polyscanline. –8,192 to +8,191, Sign Extended to 16 bits. HEIGHT: [29:16] :- The thickness of the line measured in pixels. 0 to 16,383 Pixels
3	[END_1 | START_1]	START_1: [15:0] :- The starting x-coordinate of the 1st line segment. –8,192 to +8,191, Sign Extended to 16 bits. END_1: [31:16]:- The ending x-coordinate of the 1st line segment. –8,192 to +8,191, Sign Extended to 16 bits.
...	...	...
n+2	[END_n |START_n]	START_n: [15:0] :- The starting x-coordinate of the n-th line segment. END_n: [31:16]:- The ending x-coordinate of the n-th line segment.

6.2.7 NEXTCHAR (Updated: 01-20-2003)

Functionality

Print a character at a given screen location using the default foreground and background colors.

Note: The NextChar packet relies on the Driver issuing one of the HostData_Blt packet variants immediately before hand to set-up the necessary 2D state. This is required because the NextChar packet does not execute the GMC microcode (i.e. the MSB of its opcode is not set).

Pseudocode:

1. Set the B4 (Source Fetch) and B9 (Embedded Source) Booleans. Write the 2D Booleans to the assigned context.
2. Write the BASE_ADDRESS field in the Texture T0 Constant (Source Texture Info) with the base address of the raster data in external memory where the BITMAP data will be written for the assigned context. Bits [31:19] is the base address and bits [18:16] is the current context.
3. Send an EVENT_INITIATOR with ID = SC_WAIT_WC to the Scan Converter to tell it to wait for the write confirmation before processing the subsequent draw initiator.
4. Write the raster data to external memory. The last DWORD is written with a write confirmation.
5. Write the DRAW_INITIATOR to the *VGT_DRAW_INITIATOR* register. The PRIM_TYPE is set to 2D_COPY_RECT_LIST_V0. Note that the raster order is arbitrary, as the source data will not overlap the destination for this primitive.
6. Calculate the UL, UR, LL compound indices and send them to the VGT. See the "Specification of the CP's 2D Implementation in CRAYOLA" document for details.

Format

Ordinal	Field Name
1	[HEADER]
2	{DATA_BLOCK}

`DATA_BLOCK`

Ordinal	Field Name	Description
1	[DST_Y | DST_X]	The coordinates of the top-left corner of the destination bitmap. DST_X: [15:0]:- x-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. DST_Y: [31:16]:- y-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
2	[DST_H | DST_W]	The width and height of the destination bitmap, expressed in unsigned integers. DST_W: [13:0]:- Width. DST_H [29:16]:- Height.
3	[BITMAP_DATA_1]	The 1st DWORD of the bitmap data.
...	…	…
N+2	[BITMAP_DATA_n]	The n-th DWORD of the bitmap data.

6.2.8 PLY_NEXTSCAN (Updated: 01-20-2003)

Functionality

Draw a number of scanlines or polyscanlines using the current settings.

Note: The Ply_NextScan packet relies on the Driver issuing a PolyScanLines packet immediately before hand to set-up the necessary 2D state. This is required because the Ply_NextScan packet does not execute the GMC microcode (i.e. the MSB of its opcode is not set).

Microcode Pseudopodia:

1. Send single DRAW_INITIATOR to the *VGT_DRAW_INITIATOR* register representing all of the scanlines in the packet.
    a. NUM_INDICES = COUNT * 3

    Where COUNT is the N-1 count of the number of DWORDs in the HEADER (Assumes no SETTINGS).
2. For every scan line:
    a. Calculate the UL, UR, LL compound indices and send these to the VGT. See the "Specification of the CP's 2D Implementation in CRAYOLA" document for details.
        1. HEIGHT is applied globally to each of the scanline primitives.
        2. WIDTH = End_n – Start_n.

Format

Ordinal	Field Name	Description
1	[HEADER]	The packet header
2	[HEIGHT | TOP]	TOP: [15:0] :- Y-coordinate of the scanline/polyscanline. –8,192 to +8,191, Sign Extended to 16 bits. HEIGHT: [29:16] :- The thickness of the line measured in pixels. 0 to 16,383 Pixels
3	[END_1 | START_1]	START_1: [15:0] :- The starting x-coordinate of the 1st dash. –8,192 to +8,191, Sign Extended to 16 bits. END_1: [31:16]:- The ending x-coordinate of the 1st dash. –8,192 to +8,191, Sign Extended to 16 bits.
…	…	…
n+2	[END_n |START_n]	START_n: [15:0] :- The starting x-coordinate of the 1st dash. END_n: [31:16]:- The ending x-coordinate of the 1st dash.

6.2.9 SET_SCISSORS (Updated: 10-17-2002)

Functionality

Set the scissors to the given parameters.

The PA performs the operations for scissoring.

This packet is considered a "2D State" packet. A new context is assigned only if this is the first state update packet processed after a prior 2D/3D "draw" packet.

Pseudocode:

1. Write the [TOP_LEFT] value to the *PA_SC_SCREEN_SCISSOR_TL* register for the assigned context.
2. Write the [BOTTOM_RIGHT] value to the *PA_SC_SCREEN_SCISSOR_BR* register for the assigned context.

Format

Ordinal	Field Name	Description
1	[HEADER]	The packet header
2	[TOP_LEFT]	See the PA register spec for the format of the PA_SC_SCREEN_SCISSOR_TL register.
3	[BOTTOM_RIGHT]	See the PA register spec for the format of the PA_SC_SCREEN_SCISSOR_BR register..

6.2.10 PAINT_MULTI (Updated: 07-10-2002)

Functionality

Paint a number of rectangles on the screen with one color. The color used is specified in field SETTINGS while the location and geometry of the rectangles are specified in field DATA_BLOCK.

The number of rectangles can be computed from the HEADER of the packet, so all of the paint rectangles are sent to the VGT with a single DRAW_INITIATOR using the same context.

Pseudocode:

1. Send single DRAW_INITIATOR to the *VGT_DRAW_INITIATOR* register representing all of the paint rectangles in the packet. The primitive is set as a 2D_FILL_RECT_LIST. The number of indices in the DRAW_INITIATOR is computed as follows:

   NUM_INDICES = ((COUNT + 1) ÷ 2) * 3

   Where COUNT is the N-1 DWORD count after the SETTINGS section is processed.

2. For every paint rectangle, the Micro Engine (ME) computes and outputs the Upper Left, Upper Right, and Lower Left compound indices to the VGT. See the "Specification of the CP's 2D Implementation in CRAYOLA" document for details.

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

`DATA_BLOCK`

Ordinal	Field Name	Description
1	[DST_X1 | DST_Y1]	The coordinates of the top-left corner of the 1st rectangle. DST_Y1: [15:0]:- y-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. DST_X1: [31:16]:- x-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
2	[DST_W1 | DST_H1]	The width and height of the 1st rectangle, expressed in unsigned integers. DST_H1: [13:0]:- Height. DST_W1: [29:16]:- Width.
...	...	...
2n-1	[DST_Xn | DST_Yn]	The coordinates of the top-left corner of the n-th rectangle. DST_Yn: [15:0]:- y-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. DST_Xn: [31:16]:- x-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
2n	[DST_Wn | DST_Hn]	The width and height of the n-th rectangle, expressed in unsigned integers. DST_Hn: [13:0]:- Height. DST_Wn: [29:16]:- Width.

6.2.11 BITBLT_MULTI (Updated: 04-14-2003)

Functionality

Copy a number of source rectangles to destination rectangles of the screen respectively. It is assumed that the geometry of the destination is identical to its source.

Each of the BLIT primitives can have a different rasterization order – because of the source-to-destination relationship. Therefore each of the blit primitives are sent with

Settings Restrictions:

A BitBlt_Multi with a mono opaque source, SRC_TYPE=0, or a mono transparent source, SRC_TYPE=1, and a ROP code set to source copy, 0xCC, is not supported.

Microcode Pseudopodia:

1. If ROP[7:4] == ROP[3:0], then set B4 and C2.x (Source Fetch Booleans).
2. For each Bit Blit in the packet the CP does the following:
    1. The Micro Engine (ME) computes and outputs the Upper Left, Upper Right, and Lower Left compound indices to the VGT. See the "Specification of the CP's 2D Implementation in CRAYOLA" document for details.
    2. Check for surface coherency overlap and issue flush request if needed.
    3. Compose the DRAW_INITIATOR settings and write this value to the *VGT_DRAW_INITIATOR* register for the assigned context. The primitive type is set as a 2D_COPY_RECT_LIST_V*. The primitive type represents the rasterization order that is determined by the CP. The NUM_INDICES field in the DRAW_INITIATOR = 3.

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[SRC_X1 | SRC_Y1]	The coordinates of the top-left corner of the 1st source bitmap. SRC_Y1: [15:0]:- y-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. SRC_X1: [31:16]:- x-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
2	[DST_X1 | DST_Y1]	The coordinates of the top-left corner of the 1st destination. The definition of bits is the same as SRC_X1 and SRC_Y1.
3	[SRC_W1| SRC_H1]	The width and height of the 1st source bitmap, expressed in unsigned integers. SRC_H1: [13:0]:- Height. SRC_W1: [29:16]:- Width.
…	…	…
3n-1	[SRC_Xn | SRC_Yn]	The coordinates of the top-left corner of the n-th source bitmap. SRC_Yn: [15:0]:- y-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. SRC_Xn: [31:16]:- x-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
3n-2	[DST_Xn | DST_Yn]	The coordinates of the top-left corner of the n-th destination. The definition of bits is the same as SRC_Xn and SRC_Yn.
3n	[SRC_Wn| SRC_Hn]	The width and height of the n-th source bitmap, expressed in unsigned integers. SRC_Hn: [13:0]:- Height. SRC_Wn: [29:16]:- Width.

6.2.12 TRANS_BITBLT (Updated 04-14-2003)

Functionality

Copy pixels from the source rectangle to the destination with transparency.

Settings Restrictions:

A Trans_BitBlt with a mono opaque source, SRC_TYPE=0, or a mono transparent source, SRC_TYPE=1, and a ROP code set to source copy, 0xCC, is not supported.

Pseudocode:

```
1. Write [CLR_FCN_CNTL] DWORD to the RB_CLRCMP_CONTROL register in RB.
2. CP will set the Source Fetch (B0) and Hilite (B8) Booleans if the CLRCMP_FCN_SEL field in the CLR_FCN_CNTL DWORD = 3.
3. Write [SRC_REF_CLR] DWORD to the RB_CLRCMP_SRC register in RB.
4. Write [DST_REF_CLR] DWORD to the RB_CLRCMP_DST_LO register in RB.
5. Jump to the BITBLT_MULTI microcode because ordinal 4-6 is the same format as the BITBLT packet.
```

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

`DATA_BLOCK`

Ordinal	Field Name	Description
1	[CLR_FCN_CNTL]	This field decides how the transparent blitting is done.
2	[SRC_REF_CLR]	Source reference color in the RGBQUAD format. This is the color to be stripped off from the source.
3	[DST_REF_CLR]	Destination reference color in the RGBQUAD format. This is the color to be preserved at the destination.
4	[SRC_X1 | SRC_Y1]	The coordinates of the top-left corner of the 1st source bitmap. SRC_Y1: [15:0]:- y-coordinate, ranging from -8192 to 8191. Bits 14 and 15 should be copies of bit 13. SRC_X1: [31:16]:- x-coordinate, ranging from -8192 to 8191. Bits 30 and 31 should be copies of bit 29.
5	[DST_X1 | DST_Y1]	The coordinates of the top-left corner of the 1st destination. The definition of bits is the same as SRC_X1 and SRC_Y1.
6	[SRC_W1| SRC_H1]	The width and height of the 1st source bitmap, expressed in unsigned integers. SRC_H1: [13:0]:- Height. SRC_W1: [29:16]:- Width.

- **DATA_BLOCK.CLR_FCN_CNTL**

This field controls how the source pixels are written to the destination, depending on the source and destination reference colors and comparison settings. The source pixels may be filtered against the source reference color, and the destination pixels with a specific color may be preserved according to field CLRCMP_FCN_DST.

Bit(s)	Bit-Field Name	Description
2:0	CLRCMP_FCN_SRC	Strip off the source reference color from the source pixels. 0 :- Do not strip off source pixels. All source pixels are written to the destination. 1 :- Block the blitting source. No source pixel is written to the destination. 2, 3 :- reserved. 4 :- The source pixels whose color is NOT equal to the reference color are written to the destination. (DRAW_ON_NEQ) 5 :- The source pixels whose color is equal to the reference color are written to the destination. (DRAW_ON_EQ) 6 :- Reserved. 7 :- The source pixels whose color is equal to the reference color will be XORed with the foreground color of a mono bitmap, and then written to the destination. That is, destPixel = srcPixel XOR foregrndColor if srcPixel is equal to the foreground color of a mono bitmap, specifically text. This is referred to as flipping sometimes. (FLIP_ON_EQ)
7:3	Reserved	Reserved
10:8	CLRCMP_FCN_DST	Preserve pixels at the destination. 0 :- Do not preserve the destination pixels. All pixels from the source are written to the destination. (FALSE) 1 :- Preserve all the destination pixels. No source pixel is written to the destination. (TRUE) 2, 3 :- Reserved. 4 :- The destination pixels whose color is equal to the reference color are preserved. No source pixel is written on top of the pixels. (DRAW_ON_NEQ) 5 :- The destination pixels whose color is NOT equal to the reference color are preserved. (DRAW_ON_EQ) 6, 7 :- Reserved.
23:11	Reserved	Reserved
25:24	CLRCMP_FCN_SEL	Source for Color Key 0 :- Destination: Use destination compare. 1 :- Source: Use source compare. 2 :- Source and Destination: Use AND of source and destination compare. 3 :- Hilite (CP sets Hilite Boolean (B8) if the CLRCMP_FCN_SEL = 3)
31:26	Reserved	Reserved

6.2.13 LOAD_PALETTE (Updated: 11-21-2002)

Functionality

Load a palette for future 2D operations.

This packet is considered a “2D State” packet. A new context is assigned only if this is the first state update packet processed after a prior 2D/3D “draw” packet.

Microcode Pseudopodia:

1. The CP skips the SCALE_DATATYPE information.
2. The CP writes an EVENT_INITIATOR with ID = SC_WAIT_WC to the Scan Converter. The SC will wait for the write confirmation before processing the next packet.
3. The CP writes the palette data to external memory. The last DWORD is written with a write confirmation.
4. The CP writes the external memory address to the BASE_ADDRESS field in the Texture T2 constant (LUT Texture Info Constant). This address is: BASE_ADDRESS[31:12] = Palette_Base[31:15] & Current_Context[14:12].

Format

Ordinal	Field Name	Description
1	[HEADER]	The packet header
2	[SCALE_DATATYPE]	1:- The palette has 16 entries (4 bpp palette) (*Not supported in CRAYOLA*) 2:- The palette has 256 entries (8 bpp palette).
3	[COLOR_1]	The 1st entry of the palette. Data is in destination format (i.e. ARGB8888, RGB565, RGB555,…) ***Note that 16bpp formats need to be packed to 2 entries per DWORD.***
4	[COLOR_2]	The 2nd entry of the palette. Bits are defined as above.
...	…	…
n+2	[COLOR_n]	The n-th entry of the palette.

6.3 2D PACKETS – NEW

6.3.1 HOSTDATA_BLT2 (Updated: 04-14-2003)

Functionality

This packet has similar functionality as the legacy HOSTDATA_BLT packet. This version has only one set of raster data (i.e. One Big Character).

The CP skips over the raster data and the graphics pipe is responsible for fetching the data from the command stream.

Driver cannot wrap the Ring Buffer within the packet.

Settings Restrictions:

If the ROP is set to source copy, 0xCC, then the "settings" portion of this packet should not include a brush.

Pseudopodia:

1. Unpack and Convert the FRGD_COLOR to floating point values and write them to the ALU Constant C0.
2. Unpack and Convert the BKGD_COLOR to floating point values and write them to the ALU Constant C1.
3. Set the B9 (Embedded Source) 2D Boolean.
4. If ROP[7:4] == ROP[3:0], then set B4 and C2.x (Source Fetch Booleans).
5. Write the BASE_ADDRESS field of the Texture T0 Constant with bits [31:12] of the RASTER_POINTER.
6. Write the DRAW_INITIATORs to the *VGT_DRAW_INITIATOR* register for every scan line **. The PRIM_TYPE is set to 2D_COPY_RECT_LIST_V0. Note that the raster order is arbitrary, as the source data will not overlap the destination for this primitive.
7. Calculate the UL, UR, LL compound indices and send them to the VGT. See the "Specification of the CP's 2D Implementation in CRAYOLA" document for details.
8. Skip over the raster data.

** Note for texture fetch performance, the CP generates a rectangle primitive for every scan line of the destination. This forces the texture fetch requests to be sequential for the 1D texture map.

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[FRGD_COLOR]	Foreground color in the RGBQUAD format. For mono-to color expansion only. The field is ineffective if field SRC_TYPE at SETTINGS.GUI_CONTROL is set to a type other than mono opaque or mono transparent (0 or 1).
2	[BKGD_COLOR]	Background color in the RGBQUAD format. For mono-to color expansion only. The field is ineffective if field SRC_TYPE at SETTINGS.GUI_CONTROL is set to a type other than mono opaque or mono transparent (0 or 1).
3	[BaseY | BaseX]	The coordinate of the top-left corner of the character's bitmap. BaseX: [15:0] :- X-coordinate. -8,192 to +8,191, Sign Extended to 16 bits. BaseY: [31:16] :- Y-coordinate. -8,192 to +8,191, Sign Extended to 16 bits.
4	[HEIGHT | WIDTH]	The geometry of the bitmap. WIDTH: [13:0] :- Width of the bitmap. HEIGHT: [29:16] :- Height of the bitmap.
5	[RASTER_POINTER]	Address [31:2] in the command buffer where the raster data is located.
6 to End	[RASTER_DATA]	Embedded bitmap data.

6.3.2 HOSTDATA_BLT_PNTR (Updated: 04-14-2003)

Functionality

This packet has the same functionality as the HOSTDATA_BLT2 packet.

This packet however does not include the RASTER_DATA. The RASTER_DATA is in a separate buffer than the command buffer. In addition, only one BIGCHAR is supported.

Settings Restrictions:

If the ROP is set to source copy, 0xCC, then the "settings" portion of this packet should not include a brush.

Pseudopodia:

1. Unpack and Convert the FRGD_COLOR to floating point values and write them to the ALU Constant C0.
2. Unpack and Convert the BKGD_COLOR to floating point values and write them to the ALU Constant C1.
3. Set the B9 (Embedded Source) 2D Boolean.
4. If ROP[7:4] == ROP[3:0], then set B4 and C2.x (Source Fetch Booleans).
5. Write the BASE_ADDRESS field of the Texture T0 Constant with bits [31:12] of the RASTER_POINTER.
6. Write the DRAW_INITIATORs to the *VGT_DRAW_INITIATOR* register for each scan line **. The PRIM_TYPE is set to 2D_COPY_RECT_LIST_V0. Note that the raster order is arbitrary, as the source data will not overlap the destination for this primitive.
7. Calculate the UL, UR, LL compound indices and send them to the VGT. See the "Specification of the CP's 2D Implementation in CRAYOLA" document for details.

** Note for texture fetch performance, the CP generates a rectangle primitive for every scan line of the destination. This forces the texture fetch requests to be sequential for the 1D texture map.

Format

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

`DATA_BLOCK`

Ordinal	Field Name	Description
1	[FRGD_COLOR]	Foreground color in the RGBQUAD format. For mono-to color expansion only. The field is ineffective if field SRC_TYPE at SETTINGS.GUI_CONTROL is set to a type other than mono opaque or mono transparent (0 or 1).
2	[BKGD_COLOR]	Background color in the RGBQUAD format. For mono-to color expansion only. The field is ineffective if field SRC_TYPE at SETTINGS.GUI_CONTROL is set to a type other than mono opaque or mono transparent (0 or 1).
3	[BaseY | BaseX]	The coordinate of the top-left corner of the character's bitmap. BaseX: [15:0] :- X-coordinate. –8,192 to +8,191, Sign Extended to 16 bits. BaseY: [31:16] :- Y-coordinate. –8,192 to +8,191, Sign Extended to 16 bits.
4	[HEIGHT | WIDTH]	The geometry of the bitmap. WIDTH: [13:0] :- Width of the bitmap. HEIGHT: [29:16] :- Height of the bitmap.
5	[RASTER_PNTR]	Address [31:2] of the 1st DWORD of the mono bitmap data.

6.3.3 GRADFILL (Updated: 05-20-2002)

Functionality

Draw gradient filled triangle or rectangle using the vertex colors supplied. The triangle or rectangle is defined by three or four vertices, each with a format: {x, y, and color}. No source surface is used/defined.

The primitive type is set to a triangle if there are only three vertices supplied. If there are four vertices supplied the primitive type is set to be a rectangle list.

For 16bpp mode (i.e. 5:6:5:0), the Driver expands the colors to fit into the 32bpp format (i.e. 8:8:8:8). For this mode, each color cannot exceed its ordinal resolution. – 5 bits for Red, 6 bits for Green, and 5 bits for Blue. The Driver is responsible for ensuring that this is the case.

Also, for non-32bpp destination types, the CP sets RB_BLENDCONTROL.COLOR_DITHER_MODE to DITHER_LUT. For 32bpp destination types, the GMC sets this field to DITHER_TRUNC.

The ROP code used for this packet should be 0xCC (Source Copy).

Microcode Pseudopodia:

1. Set 2D Booleans B0, B13, and C2.X
2. Compose and Send the Draw Initiator:
   a. The Primitive Type is set depending on the number of indices supplied:
      i. 3 (Triangle): DI_PT_2D_TRI_STRIP
      ii. 4 (Rectangle): DI_PT_2D_COPY_RECT_LIST. Note Color term is sent as the [SRC_X | SRC_Y] term. The shader program interprets the color value correctly.
   b. The Num_Indices = 3. Only 3 indices are sent for either primitive type. If it is a rectangle, the CP discards the last indice.
3. For each vertex, the CP writes the compound index to the VGT, where the compound index consists of the DST_X_Y and COLOR DWORDs.

Format:

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

Ordinal	Field Name	Description
1	[DST_X$_1$ | DST_Y$_1$]	The coordinates of the 1st vertex of the triangle strip. DST_Y1: [15:0]:- X-coordinate, ranging from -8192 to 8191. DST_X1: [31:16]:- Y-coordinate, ranging from -8192 to 8191.
2	[COLOR$_1$]	32bpp aRGB 8888
3	[DST_X$_2$ | DST_Y$_2$]	The coordinates of the 2nd vertex of the triangle strip. DST_Y2: [15:0]:- X-coordinate, ranging from -8192 to 8191. DST_X2: [31:16]:- Y-coordinate, ranging from -8192 to 8191.
4	[COLOR$_2$]	32bpp aRGB 8888
5	[DST_X$_3$ | DST_Y$_3$]	The coordinates of the 3rd vertex of the triangle strip. DST_Y3: [15:0]:- X-coordinate, ranging from -8192 to 8191. DST_X3: [31:16]:- Y-coordinate, ranging from -8192 to 8191.
6	[COLOR$_3$]	32bpp aRGB 8888
7	[DST_X$_4$ | DST_Y$_4$]	The coordinates of the Nth vertex of the triangle strip. DST_Y4: [15:0]:- X-coordinate, ranging from -8192 to 8191. DST_X4: [31:16]:- Y-coordinate, ranging from -8192 to 8191. ***Supplied for Rectangle Primitive Only.***
8	[COLOR$_4$]	32bpp aRGB 8888 ***Supplied for Rectangle Primitive Only.***

6.3.4 ALPHABLEND (Updated: 02-11-2003)

This packet has different functionality depending on the OP field in the first ordinal.

For OP == 0 (STRETCH_BLT):

This packet is similar to BITBLT packet with the addition of a [DST_W | DST_H] entry. The CP will use this to generate the [DST_X | DST_Y] register writes to the VGT. Booleans are set the same as in BITBLT (B4 and C2.x).

For OP == 1 (ALPHA_BLEND_CONSTANT):

The source bitmap has no per-pixel alpha value and the blending operation is applied to each of the pixels' color channels based on a constant source alpha value specified in Src_Const_Alpha.

```
Dst.Red = Round(((Src.Red * SourceConstantAlpha) +  ((255 - SourceConstantAlpha) * Dst.Red)) / 255);

Dst.Green = Round(((Src.Green * SourceConstantAlpha) + ((255 - SourceConstantAlpha) * Dst.Green)) / 255);

Dst.Blue = Round(((Src.Blue * SourceConstantAlpha) + ((255 - SourceConstantAlpha) * Dst.Blue)) / 255);

/* Do the following calculation only if the destination bitmap has an alpha channel. */

Dst.Alpha = Round(((Src.Alpha * SourceConstantAlpha) + ((255 - SourceConstantAlpha) * Dst.Alpha)) / 255);
```

Similar to a STRETCH_BLT packet, but an additional 32 bit IEEE float is included that represents a constant alpha value. This value is written to C2.w as well as RB_BLEND_ALPHA. Additionally RB_ BLENDCONTROL is written with: 0x6E016E01. Booleans B0,B14,B15,and B19 should be set.

Note that RB_BLENDCONTROL is set back to its default value** after 'rolling' the context by the GMC processing.

For OP == 2 (ALPHA_BLEND_SOURCE):

Each pixel in the source bitmap has an alpha value and the Src_Const_Alpha value is not used for the source (set to 255).

```
Dst.Red = Src.Red + Round(((255 - Src.Alpha) * Dst.Red) / 255);

Dst.Green = Src.Green + Round(((255 - Src.Alpha) * Dst.Green) / 255);

Dst.Blue = Src.Blue + Round(((255 - Src.Alpha) * Dst.Blue) / 255);

/* Do the next computation only if the destination bitmap has an alpha channel. */

Dst.Alpha = Src.Alpha + Round(((255 - Src.Alpha) * Dst.Alpha) / 255);
```

***** Note: The request to do an ALPHA_BLEND_SOURCE operation on a source that does not have an alpha term (i.e. RGB565) is invalid and will generate erroneous results.***

Similar to a STRETCH_BLT packet. Additionally RB_BLEND_ALPHA is written with a floating point 255.0 (0x437F0000) and RB_ BLENDCONTROL is written with: 0x670D6701. Booleans B0,B14,B16,B19 should be set.

Note that RB_BLENDCONTROL is set back to its default value** after 'rolling' the context by the GMC processing.

For OP == 3 (ALPHA_BLEND_BOTH):

Each pixel in the source bitmap has an alpha value and the Src_Const_Alpha value is used (a non 255 value).

```
Temp.Red = Round((Src.Red * SourceConstantAlpha) / 255);

Temp.Green = Round((Src.Green * SourceConstantAlpha) / 255);

Temp.Blue = Round((Src.Blue * SourceConstantAlpha) / 255);

/* The next calculation must be done even if the destination bitmap does not have an alpha channel. */

Temp.Alpha = Round((Src.Alpha * SourceConstantAlpha) / 255);

/* Note that the following equations use the just-computed Temp.Alpha value: */

Dst.Red = Temp.Red + Round(((255 - Temp.Alpha) * Dst.Red) / 255);

Dst.Green = Temp.Green + Round(((255 - Temp.Alpha) * Dst.Green) / 255);

Dst.Blue = Temp.Blue + Round(((255 - Temp.Alpha) * Dst.Blue) / 255);

/* Do the next computation only if the destination bitmap has an alpha channel.  */

Dst.Alpha = Temp.Alpha + Round(((255 - Temp.Alpha) * Dst.Alpha) / 255);

/* The Round(x) function rounds to the nearest integer, calculated as Trunc(x + 0.5). */
```

Similar to an ALPHA_BLEND_CONSTANT packet. The constant alpha is written to C2.w. Additionally RB_BLEND_ALPHA is written with a floating point 255.0 (0x437F0000) and RB_BLENDCONTROL is written with: 0x670D6701. Booleans B0,B14,B15,B16,B19 should be set.

Note that RB_BLENDCONTROL is set back to its default value** after 'rolling' the context by the GMC processing.

** Note: The RB_BlendControl register is defaulted to: 0x00010001 by the GMC processing in the microcode:

```
PL_RegFieldSetS(RB, RB_BLENDCONTROL, COLOR_SRCBLEND,    GL_ONE);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, COLOR_COMB_FCN,    COMB_DST_PLUS_SRC);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, COLOR_DESTBLEND,   GL_ZERO);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, COLOR_BLEND_ENABLE,DISABLE);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, COLOR_DITHER_MODE, DITHER_TRUNC);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, ALPHA_SRCBLEND,    GL_ONE);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, ALPHA_COMB_FCN,    COMB_DST_PLUS_SRC);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, ALPHA_DESTBLEND,   GL_ZERO);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, ALPHA_BLEND_ENABLE,DISABLE);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, ALPHA_DITHER_MODE, DITHER_TRUNC);
```

Format:

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

1	[OP | SRC_DATA_FORMAT]	Blending selection and source constant alpha. OP: [31:30]:- Blending operation: 0: No blending (STRETCH_BLT) 1: Constant Alpha Blending (ALPHA_BLEND_CONSTANT) 2: Per-pixel alpha blending (ALPHA_BLEND_SOURCE) 3: Constant and per-pixel alpha blending (ALPHA_BLEND_BOTH) SRC_DATA_FORMAT[5:0]: Source Data Format. The CP writes this to the Data_Format field of the T0 Texture constant. ***See the TP spec for the encoding.***
2	[SRC_CONST_ALPHA]	Source Constant Alpha in IEEE floating point format. Range is between 0.0 to 1.0, where 0.0 is transparent and 1.0 is opaque. ***This ordinal is only present for blending operations:*** ***ALPHA_BLEND_CONSTANT*** ***ALPHA_BLEND_BOTH***
3	[SRC_X | SRC_Y]	The coordinates of the top-left corner of the source bitmap. SRC_Y: [15:0]:- Y-coordinate, ranging from -8192 to 8191. SRC_X: [31:16]:- X-coordinate, ranging from -8192 to 8191.
4	[DST_X | DST_Y]	The coordinates of the top-left corner of the destination bitmap. The definition of bits is the same as SRC_X and SRC_Y.
5	[SRC_W | SRC_H]	The width and height of the source bitmap, expressed in unsigned integers. SRC_H: [13:0]:- Height. SRC_W: [29:16]:- Width.
6	[DST_W | DST_H]	The width and height of the destination bitmap, expressed in unsigned integers. The definition of bits is the same as SRC_W and SRC_H. ***This term must be supplied even if the SRC and DST terms are equal.***

6.3.5 AAFONT (Updated: 01-23-2003)

Functionality

Draw an AA font string with a source alpha-only surface to a destination surface. The source surface holds the blending factor between the foreground color and the background color for each pixel in the glyph. The following is the GDI reference rendering of the AA font:

```
if ($c_f \geq c_b$):
        $c = c_b + \alpha_k \text{ \textasciicircum\ } (1/\gamma) * (c_f - c_b)$
if ($c_f \leq c_b$):
        $c = c_b + (1 - (1 - \alpha_k) \text{ \textasciicircum\ } (1/\gamma)) * (c_f - c_b)$
where,
      c : blended color
      $c_b$ : background color
      $c_f$ : foreground color
      $\alpha_k$ : k'th blending fraction. $\alpha_k = (k == 0\ ?\ 0 : (k+1)/16)$
      $\gamma$ : 2.33
```

Before submitting this packet, the driver will cache the glyph into a device surface with the following 16bpp format:

High 8 bits: $(1 - (1 - \alpha_k) \text{ \textasciicircum\ } (1/\gamma))$, [0, 255] = (0.0, 1.0) Floating Point.

Low 8 bits: $\alpha_k \text{ \textasciicircum\ } (1/\gamma)$, [0, 255] = (0.0, 1.0) Floating Point.

So the source surface becomes an alpha surface based on the glyph. When a character is rendered, the shader would be able to select the corresponding alpha value in the high 8 bits or in the low 8 bit based on the comparison result between the two values per color component (R, G, B) in the foreground and the background colors.

Note that there could be considered two different cases. In the first case, the background color is constant; in the second case, the background color is a bitmap. The rendering could be faster for the first case because there is no need to look-up the background color in the frame buffer.

Microcode Pseudopodia:

AA_FONT_BGND: - Similar to a BITBLT packet, except that the source is assumed to be a 16 bit value made up of 2 8-bit 'alpha' components. The T0 source constant DATA_FORMAT field should be set to Fmt_8_8 (YX) (0xA). Booleans B0, B6, B17 and B19 should be set. Note that the 'font surface' must respect all surface alignment requirements.

AA_FONT_DST - Similar to a BITBLT packet, except the source is assumed to be a 16 bit value made up of 2 8-bit 'alpha' components. The T0 source constant DATA_FORMAT field should be set to _8_8 (YX) (0xA). Booleans B0, B6, B17, B18 and B19 should be set. This packet also requires that the T3 texture constant be written by the CP with the Destination Surface parameters (i.e. set this up the same way as T0 for a normal BITBLT, using the DST_PITCH_OFFSET information rather than SRC_PITCH_OFFSET info).

RB_BlendControl is set to Dither_Round instead of Dither_Truncate for the AA_Font packet. Note that the GMC processing sets the RB_BlendControl back to the following default value: 0x00010001.

```
PL_RegFieldSetS(RB, RB_BLENDCONTROL, COLOR_SRCBLEND,    GL_ONE);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, COLOR_COMB_FCN,    COMB_DST_PLUS_SRC);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, COLOR_DESTBLEND,   GL_ZERO);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, COLOR_BLEND_ENABLE,DISABLE);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, COLOR_DITHER_MODE, DITHER_TRUNC);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, ALPHA_SRCBLEND,    GL_ONE);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, ALPHA_COMB_FCN,    COMB_DST_PLUS_SRC);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, ALPHA_DESTBLEND,   GL_ZERO);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, ALPHA_BLEND_ENABLE,DISABLE);
PL_RegFieldSetS(RB, RB_BLENDCONTROL, ALPHA_DITHER_MODE, DITHER_TRUNC);
```

Format:

Ordinal	Field Name
1	[HEADER]
2	{SETTINGS}
3	{DATA_BLOCK}

DATA_BLOCK

1	[HAS_BACKCOLOR | FOREGROUND_COLOR]	HAS_BACKCOLOR: [31]:- Constant back-ground color: 0: No constant background color, the blending is between the foreground color and a background bitmap (AA_FONT_DST) 1: Has background color* (AA_FONT_BGND) FOREGROUND_COLOR: For 16bpp aRGB1555 15 - Alpha 14:10 - Red 09:05 - Green 04:00 - Blue For 16bpp RGB565 15:11 - Red 10:05 - Green 04:00 - Blue For 32bpp ARGB8888 31:24 – N/A : Alpha Not Used for AAFont Packets 23:16 - Red 15:08 - Green 07:00 - Blue
2	[BACKGROUND_COLOR]	BACKGROUND_COLOR, *optional*, depends on HAS_BACKCOLOR. This has the same format as the FOREGROUND_COLOR.
3	[SRC_X1 | SRC_Y1]	The coordinates of the top-left corner of the 1st glyph. SRC_Y1: [15:0]:- Y-coordinate, ranging from -8192 to 8191. SRC_X1: [31:16]:- X-coordinate, ranging from -8192 to 8191.
4	[DST_X1 | DST_Y1]	The coordinates of the top-left corner of the destination bitmap. The definition of bits is the same as SRC_X1 and SRC_Y1.
5	[SRC_W1| SRC_H1]	The width and height of the 1st glyph, expressed in unsigned integers. SRC_H1: [13:0]:- Height. SRC_W1: [29:16]:- width.
…	…	…
(3n-2)+2	[SRC_Xn | SRC_Yn]	The coordinates of the top-left corner of the n-th glyph. SRC_Yn: [15:0]:- Y-coordinate, ranging from -8192 to 8191. SRC_Xn: [31:16]:- X-coordinate, ranging from -8192 to 8191.
(3n-1)+2	[DST_Xn | DST_Yn]	The coordinates of the top-left corner of the destination bitmap. The definition of bits is the same as SRC_Xn and SRC_Yn.
3n+2	[SRC_Wn| SRC_Hn]	The width and height of the n-th glyph, expressed in unsigned integers. SRC_Hn: [13:0]:- Height. SRC_Wn: [29:16]:- Width.

* If there is a background color, assume an opaque rectangle is already painted. The supplied constant background color is only for the alpha selection calculation.

6.3.6 2D_ENDIAN_MODE (Updated: 01-22-2003)

Functionality

Update the source and destination endian mode controls for 2D operations.

The DP_SRC_ENDIAN information is used by the CP to fill in the Texture T0 constant that is composed by the microcode while processing 2D primitives.

The DP_DST_ENDIAN information is written to the RB_Color0_Info.colo0_endian field when the GMC for a 2D packet is processed.

This packet should not be used in a real-time stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	The packet header.
2	[DP_SRC_ENDIAN]	DP_SRC_ENDIAN[1:0] 00 – ENDIAN_NONE. No Endian Swapping. 01 – ENDIAN_8IN16: 0xAABBCCDD → 0xBBAADDCC 10 – ENDIAN_8IN32: 0xAABBCCDD → 0xDDCCBBAA 11 – ENDIAN_16IN32: 0xAABBCCDD → 0xCCDDAABB This format is defined in the CRAYOLA Texture Constant spreadsheet. Updates to the Texture Constant format for this field will supercede this specification.
3	[DP_DST_ENDIAN]	DP_DST_ENDIAN[1:0] – Same format as the DP_SRC_ENDIAN.

6.3.7 DRAW_2D_COHER_RECT (Updated: 05-12-2003)

Functionality

Draws the extents of the 2D coherency rectangle that is maintained by the CP for 2D source coherency. Polylines are used to draw the top and left extents. No line stippling is supported and a foreground color needs to be specified in the GMC.

This packet should not be used in a real-time stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	The packet header.
2	[DUMMY]	Dummy Data. Any value can be provided.

6.4 3D PACKETS

This section documents the formats of the 3D draw packets after the June 15th milestone is met.

The only change to the 3D draw packets is that the INDEX_OFFSET DWORD has been removed.

6.4.1 DRAW_INDX

Functionality

Draws a set of primitives using a vertex buffer(s) pointed to indices fetched or auto-generated by the VGT.

The CP tests Viz Query status and deterministically issues DMA request of index data to the VGT. The corresponding DRAW_INITIATOR is processed through the Micro Engine as a Type-0 packet.

The VGT can auto-generate indices. The SOURCE_SELECT field in the DRAW_INITIATOR indicates that auto-generation is desired. See the CRAYOLA VGT specification for details. The INDEX_BASE and INDEX_SIZE in the DRAW_INDX packet must not be supplied when auto-generation of indices is desired. The COUNT field in the HEADER of the packet will reflect the decreased size of the DRAW_INDX packet and the CP will stop accordingly.

This packet cannot appear in a Real-Time Stream.

Microcode:

1. If the VIZQ_USE flag is set, the CP will test Visibility results referenced by the VIZQ_ID.
2. If the Viz_Query test passes (i.e. Viz Query "DISCARD" = '0'):
    a. The PFP issues a DMA request of the index data to the VGT's Index DMA Engine. This is accomplished by writing the INDEX_BASE to the VGT_DMA_BASE** register and the INDEX_SIZE to the VGT_DMA_SIZE** register. These write requests bypass the Micro Engine (ME) and go through an independent path in the RBBM to the VGT via the Global Register Bus. The VGT is responsible for associating the index data with the DRAW_INITIATOR write.
    b. The Pre-Fetch Parser (PFP) discards everything but the packet's Header and DRAW_INITIATOR. It sends these to the ME. The DWORD count in the Header is set to zero by the PFP. The ME subsequently writes the DRAW_INITIATOR value to the *VGT_DRAW_INITIATOR* register at the current context.
3. If the Viz_Query test fails (i.e. Viz Query "DISCARD" = '1'), then the PFP discards the entire DRAW_INDX packet.

**** Note: The CP always writes the VGT_DMA_BASE and VGT_DMA_SIZE to context zero. The VGT gets the associated context from the associated draw initiator, which is written to the correct context.**

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[VIZQ_USE | VIZQ_ID]	Viz Query Information: Bit 8 -- VIZQ_USE: Flag to indicate whether or not to use the visibility results for the packet. Bits 5:0 --VIZQ_ID: ID[5:0] of which of the 64 queries to test.
3	[DRAW_INITIATOR]	Draw Initiator Register in the CRAYOLA. *Written to the VGT_DRAW_INITIATOR register for the assigned context.*
4	[INDEX_BASE]	Base Address [31:1] of Index Buffer (Word-Aligned). *Written to the VGT_DMA_BASE register (No Context Supplied).* (Must Not be Supplied if Indices are Auto-Generation)
5	[SWAP | INDEX_SIZE]	SWAP[31:30] – Swap code used when fetching the index buffer by the VGT. The CP just passes the entire DWORD to the VGT. The VGT separates-out the SWAP field. INDEX_SIZE = Size [23:0] of index buffer in WORDs. *Written to the VGT_DMA_SIZE register (No Context Supplied).* (Must Not be Supplied if Indices are Auto-Generation)

6.4.2 3D_DRAW_INDX_2

Functionality

Draws a set of primitives using a vertex buffer(s) pointed to indices in the packet or indices auto-generated by the VGT.

This packet is used for draw packets with a "small number" of indices. It is faster for the Driver to just put the indices into the command buffer instead of copying them to a separate index buffer.

This packet cannot appear in a Real-Time Stream.

Microcode Pseudopodia:

1. Write the DRAW_INITIATOR DWORD to the *VGT_DRAW_INITIATOR* register.
2. Write the rest of the data to the Immediate Data FIFO in the VGT (*VGT_IMMED_DATA* register). Note that this step is optional if the indices are to be auto-generated (Indicated in the DRAW_INITIATOR DWORD).

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[DRAW_INITIATOR]	Primitive type and other control. *Written to the VGT_DRAW_INITIATOR register for the assigned context.*
3 to End	[indx16 #1 | indx16 #0] or [indx32 #0]	Index Data. *Written to the VGT_IMMED_DATA register for the assigned context.* Up to or 32K 16-bit indices or 16K 32-bit indices to vertex data pointed to by state registers. The INDEX_SIZE field in the DRAW_INITIATOR indicates whether the indices are 16-bit or 32-bit. (Must not be supplied if indices are auto-generated as indicated in the "Draw Initiator")

6.5 STATE MANAGEMENT PACKETS

6.5.1 SET_STATE

Functionality

This packet sets the state for rendering operations. The parameters are the pointers to each state sub-block and instruction code.

For sub-blocks, this command checks to see if any of the submitted sub-block pointers are already resident (i.e. Pointer and Size match previous submittal). If not, the sub-blocks that miss are fetched and loaded into the CP. The Micro Engine (ME) then waits for an available context and loads the fetched state data into the chip. If the sub-block pointers match, then the sub-block is not fetched.

The address in the memory-mapped register space where the sub-block data will be written is computed as follows:

Memory-Mapped Register Byte Address [16:2] = "01"// Context[2:0] // SUBBLK_n_GFX_OFFSET [9:0]

where 'n' is the sub-block number {0,1,2,3,4,5,6,7}.

For instruction memory updates (Vertex and Pixel Shaders), the base address and code size are included in the Set_State command packet. The CP remembers the last instance of these values. If the base address or size of the shader is different from the last occurrence, the CP will fetch the instruction code and write these instructions to the memory-mapped Instruction Memory. The memory-mapped address is computed as follows:

Vertex Shaders:

Addr = SQ_INSTRUCTION0_ALU_0 + VERTEX_SHADER_BASE + vsRelativeDwordAddr

Pixel Shaders:

Addr = SQ_INSTRUCTION0_ALU_0 + PIXEL_SHADER_BASE + psRelativeDwordAddr

The CP's Micro Engine maintains the shader base and the relative start address for both the Vertex and Pixel instruction code. The relative start addresses wrap within the Instruction Memory. The CP writes the Vertex Shader relative start address and size to the SQ_VS_PROGRAM register and the Pixel Shader relative start address and size to the SQ_PS_PROGRAM register in the Sequencer. These values are the start address in instructions and the size in instructions. An instruction consists of 3 DWORDs.

The instruction memory is partitioned as a dual ring. The Vertex and Pixel shaders are independently fetched.

For instruction memory updates, the ME waits for enough available instruction space before writing the instruction code. This may result in the ME waiting for multiple contexts to free.

The Set_State packet is processed in two places in the CP – the Pre-Fetch Parser (PFP) and the Micro Engine (ME). The PFP remembers the last pointer and size for each of the sub-blocks and last address/size for instruction code. The PFP issues the fetch requests for the sub-blocks and instruction code if they are not already resident in the chip (i.e. Invalid or Mismatch Occurs). The ME keeps track of the usage of the contexts and will stall the command stream until the desired context (from the PFP) is available.

There is a "disable" control per item in the Set_State packet to allow sub-block and shader code updates to be skipped.

This packet can be used within a Real-Time stream to update the sub-block data, but no matching occurs. All the sub-blocks are unconditionally loaded if not disabled. Vertex and Pixel shader code cannot be loaded with this packet. The Vertex and Pixel shader pointers are ignored.. The Im_Load and Im_Load_Immediate packets are provided for updating the real-time shader code.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[Sub_Blk_0 | Disable | Size_0]	Sub_Blk_0 – Sub-Block Pointer[31:5]. Address in Memory of Sub-Block Data. Disable [4] – Disables processing of the sub-block (i.e. CP Skips). Size_0 – Sub-Block Size[3:0]. Size in Multiple of 8 DWORDs: A value of 0x0 specifies that 8 DWORDs will be fetched. A value of 0x1 specifies that 16 DWORDs will be fetched. : : : : : : : : A value of 0xE specifies that 120 DWORDs will be fetched. A value of 0xF specifies that 128 DWORDs will be fetched.
3	[Sub_Blk_1 | Disable | Size_1]	Sub-Block #1 Pointer and Size. Same format as Ordinal 2.
4	[Sub_Blk_2 | Disable | Size_2]	Sub-Block #2 Pointer and Size. Same format as Ordinal 2.
5	[Sub_Blk_3 | Disable | Size_3]	Sub-Block #3 Pointer and Size. Same format as Ordinal 2.
6	[Sub_Blk_4 | Disable | Size_4]	Sub-Block #4 Pointer and Size. Same format as Ordinal 2.
7	[Sub_Blk_5 | Disable | Size_5]	Sub-Block #5 Pointer and Size. Same format as Ordinal 2.
8	[Sub_Blk_6 | Disable | Size_6]	Sub-Block #6 Pointer and Size. Same format as Ordinal 2.
9	[Sub_Blk_7 | Disable | Size_7]	Sub-Block #7 Pointer and Size. Same format as Ordinal 2.
10	[VS_INSTR_BASE | Disable]	Vertex Shader Instruction Code Base Address – [31:5] Disable [4] – Disables processing of the VS Instruction Data (i.e. CP Skips). Reserved [3:0]
11	[VS_INSTR_SIZE]	Vertex Shader Size in DWORDs of the Vertex Shader Code – [13:0] Note that the size must also be a multiple of 3 DWORDs.
12	[PS_INSTR_BASE | Disable]	Pixel Instruction Code Base Address – [31:5] Disable [4] – Disables processing of the PS Instruction Data (i.e. CP Skips). Reserved [3:0]
13	[PS_INSTR_SIZE]	Pixel Shader Size in DWORDs of the Pixel Shader Code – [13:0] Note that the size must also be a multiple of 3 DWORDs.

6.5.2 SET_CONSTANT

Functionality

This packet loads constant data into the chip.

The CONST_ID field identifies which constant is being updated. The CP chooses the starting memory-mapped register address based on the CONST_ID. The CONST_OFFSET field is a DWORD-offset from the starting address. All the constant data in the packet is written to consecutive register address beginning at the starting address.

Note that the CONST_OFFSET must be a multiple of 16 for ALU constants and a multiple of 6 for Texture Constants.

The equations below show the computation of the starting addresses:

ALU: Start_Address[16:2] = SQ_CONSTANT_0[16:2] + CONST_OFFSET.

Texture: Start_Address[16:2] = SQ_FETCH_0[16:2] + CONST_OFFSET.

Boolean: Start_Address[16:2] = SQ_CF_BOOLEANS[16:2] + CONST_OFFSET.

Loop: Start_Address[16:2] = SQ_CF_LOOP[16:2] + CONST_OFFSET.

Register: Start_Address[16:2] = A[16:2] = 0x2000** + CONST_OFFSET + Current_Context.

The SET_CONSTANT packet can appear in a Real-Time stream. For real-time streams, the starting addresses are computed as follows:

ALU: Start_Address[16:2] = SQ_CONSTANT_RT_0[16:2] + CONST_OFFSET.

Texture: Start_Address[16:2] = SQ_FETCH_RT_0[16:2] + CONST_OFFSET.

Boolean: Start_Address[16:2] = SQ_CF_RT_BOOLEANS[16:2] + CONST_OFFSET.

Loop: Start_Address[16:2] = SQ_CF_RT_LOOP[16:2] + CONST_OFFSET.

Register: Start_Address[16:2] = A[16:2] = 0x2000** + CONST_OFFSET.

** Note: 0x2000 is the start of the GFX decode space (Byte Address = 0x8000).

For Non-Real-Time Constants and Incremental Register updates, the CP will write the data to external memory if the corresponding constant write enable is set. This allows an entire constant context to be later loaded into the chip with the LOAD_CONSTANT_CONTEXT (LCC) packet. The LOAD_CONSTANT_CONTEXT packet controls the write enables. See this packet for details.

The starting external memory address that the constant data is written to is computed as follows:

ALU_Mem_Start_Address[31:2] = CONST_BASE_ALU[31:13] & CONST_OFFSET[10:0]
Texture_Mem_Start_Address[31:2] = CONST_BASE_TEX[31:13] & CONST_OFFSET[10:0]
Loop_Mem_Start_Address[31:2] = CONST_BASE_LOOP[31:13] & CONST_OFFSET[10:0]
Bool_Mem_Start_Address[31:2] = CONST_BASE_LOOP[31:13] & CONST_OFFSET[10:0]
Register_Mem_Start_Address[31:2] = Register_Base[31:13] & CONST_OFFSET[10:0]

If the data is written to memory, the last DWORD is written with a write confirmation. The PFP uses the confirmation to throttle the processing of the LOAD_CONSTANT_CONTEXT packet.

If the packet is processed when the associated write enable flag is cleared, the corresponding "base valid" flag is reset to force the constant load on the next LOAD_CONSTANT_CONTEXT packet. There is an individual "base valid flag" for each CONST_ID type. Note that the "base valid flags" can be reset by the INVALIDATE_STATE packet, which is defined elsewhere in this specification.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CONST_CONTROL | CONST_ID | CONST_OFFSET]	CONST_CONTROL[31:24] : ***Reserved*** CONST_ID [23:16] :. 0x0 – ALU Constant (Vertex and Pixel) 0x1 – Texture Constant 0x2 – Boolean Constant 0x3 – Loop Constant 0x4 – Incremental Register Update 0x5 to 0xFF – ***Reserved*** CONST_OFFSET[10:0] – Offset in DWORDs from the base address for the constant type as implied by the CONST_ID field. ***This field must be a multiple of 16 for ALU constants and a multiple of 6 for Texture constants.***
3 to N	[Constant Data]	DWORD Data for Constants. ***Note that the number of DWORDs must be a multiple of 16 for ALU constants and a multiple of 6 for Texture constants.***

6.5.3 LOAD_CONSTANT_CONTEXT

Functionality

This packet provides the ability to have the CP fetch an existing constant context into the chip instead of using individual SET_CONSTANT packets.

The CP stores base addresses and valid flags for each CONST_ID type.

For Non-Real-Time Stream Constant Updates: If the "valid flag" is false, or the "base address" does not match the BASE_ADDR field in the packet, or the FORCE_MISMATCH is set, the CP will fetch "NUM_DWORDS" of DWORDs from external memory. ***Note that if the NUM_DWORDS field is zero, no constant data will be fetched.***

To preserve the coherency between writes and reads, the CP first waits until all prior data has been written to memory from prior SET_CONSTANT packets before issuing read request(s) for the constant data.

The CP computes the DWORD-aligned external memory address as follows:

Mem_Addr[31:2] = BASE_ADDR[31:13] & CONST_OFFSET[10:0]

The CP writes the constant data to consecutive register addresses starting at the computed starting address. The starting address is computed as shown below.

For Non-Real-Time Streams:

ALU: Start_Address[16:2] = SQ_CONSTANT_0[16:2] + CONST_OFFSET.

Texture: Start_Address[16:2] = SQ_FETCH_0[16:2] + CONST_OFFSET.

Loop: Start_Address[16:2] = SQ_CF_LOOP[16:2] + CONST_OFFSET.

Bool: Start_Address[16:2] = SQ_CF_BOOLEANS[16:2] + CONST_OFFSET.

Register: Start_Address[16:2] = 0x2000**[16:2] + CONST_OFFSET + Current_Context.

For Real-Time Streams:

ALU: Start_Address[16:2] = SQ_CONSTANT_RT_0[16:2] + CONST_OFFSET.

Texture: Start_Address[16:2] = SQ_FETCH_RT_0[16:2] + CONST_OFFSET.

Loop: Start_Address[16:2] = SQ_CF_RT_LOOP[16:2] + CONST_OFFSET.

Bool: Start_Address[16:2] = SQ_CF_RT_BOOLEANS[16:2] + CONST_OFFSET.

Register: Start_Address[16:2] = 0x2000**[16:2] + CONST_OFFSET.

** Note: 0x2000 is the start of the GFX decode space (Byte Address = 0x8000).

This packet is also used to enable the writing of the data back to memory for SET_CONSTANT packets. If the CONST_CONTROL field is 0x01, then the write enable is set for the CONST_ID type.

Note that the ME_INIT packet clears the constant write enables (ALU, Texture, Loop, Bool, and, Reg) unconditionally when processed by the Pre-Fetch Parser. See the ME_INIT packet for details.

The respective base addresses will be remembered from the BASE_ADDR in preparation for the next occurrence of the LOAD_CONSTANT_CONTEXT packet according to the state of the write enables as follows:

Previous Write Enable	New Write Enable	Save Base Address
0	0	No (Valid Flag is ***Preserved***)
0	1	Yes (Valid Flag is ***Set***)
1	0	No (Valid Flag is ***Preserved***)
1	1	Yes (Valid Flag is ***Set***)

Note that the setting/clearing of the write enable is done even if the packet matches and is discarded.

The "base valid" flags can also be reset by the INVALIDATE_STATE packet defined elsewhere in this specification.

For Real-Time streams, no matching of the prior base address occurs. The constant data is unconditionally fetched.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[BASE_ADDR | FORCE_MISMATCH]	BASE_ADDR [31:13] – Base address for the block in Memory from where the CP will fetch the constants. For CONST_ID = 0x04 (Incremental Register Update), the BASE_ADDR is the base address for subsequent register writes when a Set_Constant is issued with Const_ID = 0x04 and writes are enabled. FORCE_MISMATCH[0] – Set to force the mismatch regardless of the comparison between the base address pointers.
3	[CONST_CONTROL | CONST_ID | CONST_OFFSET]	CONST_CONTROL[31:24] : 0x00 – None 0x01 –Write_Enable - Sets Write Enable Type Identified by the CONST_ID field. 0x02 to 0xFF – ***Reserved*** CONST_ID [23:16] :. 0x00 – ALU Constant (Vertex and Pixel) 0x01 – Texture Constant 0x02 – Boolean Constant 0x03 – Loop Constant 0x04 – Incremental Register Update. 0x05 to 0xFF – ***Reserved*** Bits 15:11 -- ***Reserved*** CONST_OFFSET[10:0] – Offset in DWORDs from the base address for the constant type as implied by the CONST_ID field.
4	[NUM_DWORDS]	NUM_DWORDS[11:0] -- Number of DWORDS that the CP will fetch and write to the constant memory in the chip. A value of zero will cause no constants to be loaded. The base and valid pointers will be updated in the CP's matching logic and the constant write enable will be updated never the less. This must be a multiple of 16 for ALU constants and a multiple of 6 for Texture constants.
N ** Up to N=257	[CONST_OFFSET]	Bits 31:24– ***Reserved*** CONST_ID[23:16] – Needs to be set to the same value as ordinal #3. Bits 15:12– ***Reserved*** CONST_OFFSET[10:0] – Same Definition as Above.
N+1 ** Up to N+1=258	[NUM_DWORDS]	Bits 31:12 – ***Reserved*** NUM_DWORDS[11:0] – Same Definition as Above.

***** Note: Ordinals #3 and #4 can be replicated in order to generate multiple requests for the same type of constant. The limit is 127 replications (N=257, N+1=258). This allows the individual loading of all 128 16-DWORD constants that are exposed in the register map.***

6.5.4 IM_LOAD

Functionality

This command causes a load of the Vertex, Pixel, Real-Time, or Shared instruction code. The intended use of this packet is for incremental state updates of the program memory (i.e. Instead of using the Set_State packet).

This and the Im_Load_Immediate are the only packets that can be used to update Real-Time and Shared Instruction code.

The CP compares the address and size in this packet with a "valid" address and size for the stream that is being loaded. If the address and size are valid in the CP and they match, then no code is loaded.

See the Set_State packet for details of where the CP writes the instruction code for Pixel and Vertex shaders.

For Real-Time and Shared code, the relative instruction starting address is included in the packet. The CP uses the following equation to determine the memory-mapped DWORD address.

Addr[16:2] = SQ_INSTRUCTION_ALU_0 + INSTR_START[27:16] * 3.

Note that the INSTR_START is ignored for Vertex and Pixel shader updates. The INSTR_START is required however for real-time and shared code updates.

The IM_LOAD packet can be used within a Real-Time Stream. The Real-Time/Shared code is unconditionally loaded and written to the INSTR_START address in the instruction memory.

For Real-Time instruction updates, the CP writes the SQ_PS_PROGRAM value with the size set to 0x800 and the base set to INSTR_START.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[INSTR_BASE_ADDR | CODE_ID]	Instruction Code Base Address – [31:5] Double Octword Aligned 4:2 -- Unused 1:0 : Instruction Code Identifier 00 – Vertex 01 – Pixel 1X – Real-Time / Shared Code
3	[INSTR_START | INSTR_SIZE]	INSTR_START [27:16] –Relative instruction address where the CP will start writing the shader code for the real-time/shared code. INSTR_SIZE [13:0] -- Instruction Code Size in DWORDs. Note that the INSTR_SIZE should also be a multiple of 3 DWORDs

6.5.5 IM_LOAD_IMMEDIATE

Functionality

This command has the same functionality as the IM_LOAD packet, except that the instruction code is embedded in the packet.

The reason for this packet is primarily to minimize the number of latencies encountered when loading 2D default state. If the IM_LOAD were used the 3D-to-2D transition would encounter 2 latencies – the first to get the 2D Indirect Buffer and the second to get the instruction code.

The IM_LOAD_IMMEDIATE packet can be used within a Real-Time Stream. The real-time code will be loaded starting at the INSTR_START address.

For Real-Time instruction updates, the CP writes the SQ_PS_PROGRAM value with the size set to 0x800 and the base set to INSTR_START.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CODE_ID]	31:2 -- Unused 1:0 : Instruction Code Identifier 00 – Vertex 01 – Pixel 1X – Real-Time / Shared Code
3	[INSTR_START | INSTR_SIZE]	INSTR_START [27:16] –Relative instruction address where the CP will start writing the shader code for the real-time/shared code. INSTR_SIZE [13:0] -- Instruction Code Size in DWORDs. Note that the INSTR_SIZE should also be a multiple of 3 DWORDs
4 to N	[INSTR_CODE_n]	Instruction Code.

6.5.6 INVALIDATE_STATE

Functionality

This command provides for selective invalidation of the Sub-Block, Vertex Shader, Pixel Shader, Constant, and Incremental Register "last-load" pointers.

This packet can be used as an "Invalidate All State" when all the invalidate control bits are set.

If the Driver needs confirmation of the state invalidation, the MEM_WRITE packet could be used. The Driver would insert a MEM_WRITE after the INVALIDATE_STATE packet. The consequential semaphore write would indicate that the state pointers have been invalidated.

The INVALIDATE_STATE packet is not allowed within a Real-Time stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[INVALIDATE_CNTL]	Bits 31:13 – Reserved Bit 14 – Set to Invalidate Loop Constant Base. Bit 13 – Set to Invalidate Boolean Constant Base. Bit 12 – Set to Invalidate Incremental Register Update Base. Bit 11 – Set to Invalidate Texture Constant Base. Bit 10 – Set to Invalidate ALU Constant Base. Bit 9 – Set to Invalidate Pixel Instruction Code Address and Size. Bit 8 – Set to Invalidate Vertex Instruction Code Address and Size. Bit 7 – Set to Invalidate Sub-Block Pointer 7. Bit 6 – Set to Invalidate Sub-Block Pointer 6. Bit 5 – Set to Invalidate Sub-Block Pointer 5. Bit 4 – Set to Invalidate Sub-Block Pointer 4. Bit 3 – Set to Invalidate Sub-Block Pointer 3. Bit 2 – Set to Invalidate Sub-Block Pointer 2. Bit 1 – Set to Invalidate Sub-Block Pointer 1. Bit 0 – Set to Invalidate Sub-Block Pointer 0.

6.6 WAIT-ON / SYNCHRONIZATION PACKETS

6.6.1 REG_RMW

Functionality

The CP reads a register performs the logic operation on its value and writes the modified value to the register.

Reg[Adrs] ← [Reg[Adrs] & [AND_MASK | Reg[AND_Adrs]] | [OR_MASK | Reg[OR_Adrs]]

The address is limited to byte address 0x7FFC (DWORD address 0x1FFF). The context management associated with accessing registers above this address is not supported.

Note that ordinals #3 and #4 can be register addresses instead of immediate values. If an addresses is supplied, the micro engine will read the register and use its value as the appropriate mask.

This packet can be executed within either a non-Real-Time or Real-Time stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[AND_MASK_SRC | OR_MASK_SRC | MOD_ADRS]	AND_MASK_SRC[31] – 0 → Immediate, 1 → Reg[And_Adrs] OR_MASK_SRC[30] – 0 → Immediate, 1 → Reg[Or_Adrs] ADDRESS[12:0] – Register DWORD address. Maximum DWORD address is 0x1FFF.
3	[AND_MASK] or [AND_ADRS]	AND_Mask [31:0] – Value logically and'd with register contents. AND_ADRS[12:0] – Register DWORD address for And_Mask. Max is 0x1FFF.
4	[OR_MASK] or [OR_ADRS]	OR_Mask[31:0] – Value logically or'd with AND Mask result. OR_ADRS[12:0] – Register DWORD address for Or_Mask. Max is 0x1FFF.

6.6.2 WAIT_REG_MEM

Functionality

The CP polls either a memory or register location (indicated by MEM_SPACE) and tests the polled value with the reference value given in the command packet. The test is qualified by both the specified function and mask.

If the test passes, the parsing continues. If it fails, the CP waits for the Wait_Interval * 16 Clocks, then tests the Poll_Address again.

This packet can be executed within either a non-Real-Time or Real-Time stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[MEM_SPACE | FUNCTION]	31:5: Reserved MEM_SPACE [4]: 0=>Register, 1=>Memory FUNCTION [2:0] 000 – Always (Compare Passes). Still does read operation. 001 – Less Than (<) the Reference Value. 010 – Less Than or Equal (<=) to the Reference Value. 011 – Equal (=) to the Reference Value. 100 – Not Equal (!=) to the Reference Value. 101 – Greater Than or Equal (>=) to the Reference Value. 110 – Greater Than (>) the Reference Value. 111 – Always (Compare Passes). Still does read operation.
3	[POLL_ADDRESS]	Address to poll. If the address is a memory location then bits [31:2] specify the address and [1:0] indicate the swap code to be used. If the address is a memory-mapped register, then bits [14:0] is the DWORD memory-mapped register address that the CP will read.
4	[REFERENCE]	Reference Value [31:0].
5	[MASK]	Mask for Comparison [31:0]
6	[WAIT_INTERVAL]	Wait_Interval[15:0]: Interval to wait between unsuccessful polling operations. The ME will poll every 16*Wait_Interval clocks.

6.6.3 WAIT_REG_EQ

Functionality

The CP polls a register location and tests the polled value for equality with the reference value, qualified by the mask value, given in the command packet.

If the test passes, the parsing continues. If it fails, the CP waits for the Wait_Interval * 16 Clocks, then tests the Poll_Address again.

This packet can be executed within either a non-Real-Time or Real-Time stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
3	[POLL_ADDRESS]	Address to poll: bits [14:0] contain the DWORD memory-mapped register address that the CP will read.
4	[REFERENCE]	Reference Value [31:0].
5	[MASK]	Mask for Comparison [31:0]
6	[WAIT_INTERVAL]	Wait_Interval[15:0]: Interval to wait between unsuccessful polling operations. The ME will poll every 16*Wait_Interval clocks.

6.6.4 WAIT_REG_GTE

Functionality

The CP polls a register location and tests the polled value to be greater than or equal with the reference value, qualified by the mask value, given in the command packet.

If the test passes, the parsing continues. If it fails, the CP waits for the Wait_Interval * 16 Clocks, then tests the Poll_Address again.

This packet can be executed within either a non-Real-Time or Real-Time stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
3	[POLL_ADDRESS]	Address to poll: bits [14:0] contain the DWORD memory-mapped register address that the CP will read.
4	[REFERENCE]	Reference Value [31:0].
5	[MASK]	Mask for Comparison [31:0]
6	[WAIT_INTERVAL]	Wait_Interval[15:0]: Interval to wait between unsuccessful polling operations. The ME will poll every 16*Wait_Interval clocks.

6.6.5 MEM_WRITE

Functionality

This command packet provides the opportunity to write a single DWORD to a memory location. One use could be as a semaphore write to indicate to the CPU that an operation has completed. The data is embedded in the command packet.

This packet can be executed within either a non-Real-Time or Real-Time stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[ADDRESS]	DWORD-Aligned Address [31:2] Bits [1:0] – Swap function used for data write.
3	[DATA]	Data [31:0]

6.6.6 MEM_WRITE_CNTR

Functionality

This command packet provides the opportunity to write the current value in CP_PROG_COUNTER to a memory location. The data in the counter is incremented every 1 or16 core clocks, depending on the setting in bit 31 of CP_ME_CNTL.Prog_Cnt_Size.

This packet can be executed within either a non-Real-Time or Real-Time stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[ADDRESS]	DWORD-Aligned Address [31:2] Bits [1:0] – Swap function used for data write.

6.6.7 COND_WRITE

Functionality

The CP polls either a memory or register location (indicated by POLL_SPACE) and tests the polled value with the reference value provided in the command packet. The test is qualified by both the specified function and mask.

If the test passes, the write occurs to either a register or memory depending on WRITE_SPACE.

If the test fails, the CP skips the write. In either case, the CP then continues parsing the command stream.

This packet can be executed within either a non-Real-Time or Real-Time stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[WRITE_SPACE | POLL_SPACE | FUNCTION]	31:9: Reserved WRITE_SPACE 8: 0=>Register, 1=>Memory 7:5 : Reserved POLL_SPACE 4: 0=>Register, 1=>Memory FUNCTION [2:0] 000 – Always (Compare Passes). Still does read operation. 001 – Less Than (<) the Reference Value. 010 – Less Than or Equal (<=) to the Reference Value. 011 – Equal (=) to the Reference Value. 100 – Not Equal (!=) to the Reference Value. 101 – Greater Than or Equal (>=) to the Reference Value. 110 – Greater Than (>) the Reference Value. 111 – Always (Compare Passes). Still does read operation.
3	[POLL_ADDRESS]	Address to Poll If the address is a memory location then bits [31:2] is the address and [1:0] is the swap code to be used. If the address is a memory-mapped register, then bits [14:0] is the DWORD memory-mapped register address that the CP will read.
4	[REFERENCE]	Reference Value [31:0].
5	[MASK]	Mask for Comparison [31:0]
6	[WRITE_ADDRESS]	If WRITE_SPACE = Register: WRITE_ADDRESS[14:0] -- DWORD memory-mapped register address that the will be written. If WRITE_SPACE = Memory: WRITE_ADDRESS[31:2] -- DWORD-Aligned Address of destination memory location. WRITE_ADDRESS[1:0] -- SWAP Used for Memory Write.
7	[WRITE_DATA]	Write Data[31:0] that will be conditionally written to the ADDRESS.

6.6.8 *EVENT_WRITE*

Functionality

The CP will generate the event given in the packet by writing to the VGT_EVENT_INITIATOR register.

For the "time stamp" (_TS) events, the CP will also set up a write to memory (timestamp) that will occur at the completion of the specified event. The Address & Data fields are therefore required for the time stamp events.

There's a counter select (CNTR_SEL) bit in ordinal 2 that applies to Cache_Flush_Ts, Cache_Flush_and_Inv_Ts, Vs_Done_Ts & Ps_Done_Ts events only. When this bit is set it will select the current value in CP_PROG_COUNTER rather than the embedded packet data.

There's also a software management select (SWM_SEL) bit that applies only to the shader done events. When this bit is set, it will write to the software managed event address & data registers rather than the hardware managed registers.

Note for the Screen Extent Report event that the Address field is required, but the Data field should not be present.

Note also that only the zpass_done event effects the current context. The other events have no effect on context.

This packet is not supported within a Real-Time stream.

Pseudopodia

The CP decodes the EVENT_TYPE of the EVENT_INITIATOR DWORD and performs the actions listed below:

Event Code	Event Description	Action Performed
VS_DONE_TS	Vertex Shader Done (VSD)	Write ADDRESS to CP_ME_VS_EVENT_ADDR FIFO Write DATA to CP_ME_VS_EVENT _DATA FIFO Write the EVENT_INITIATOR to the VGT_EVENT_INITIATOR register. **A VsDone Event from the SQ will then generate a DWORD write of the DATA to the ADDRESS.**
PS_DONE_TS	Pixel Shader Done (PSD)	Write ADDRESS to CP_ME_PS_EVENT_ADDR FIFO Write DATA to CP_ME_PS_EVENT _DATA FIFO Write the EVENT_INITIATOR to the VGT_EVENT_INITIATOR register. **A PsDone Event from the SQ will then generate a DWORD write of the DATA to the ADDRESS.**
CACHE_FLUSH_TS CACHE_FLUSH_AND_INVALIDATE_TS	Cache Flush Done (CFD)	Write ADDRESS to CP_ME_CF_EVENT_ADDR FIFO Write DATA to CP_ME_CF_EVENT_DATA FIFO Write the EVENT_INITIATOR to the VGT_EVENT_INITIATOR register. **A pulse on the RC_CP_cache_flush signal will then generate a DWORD write of the DATA to the ADDRESS.**
SCREEN_EXT_RPT	Screen Extent Report	Sends Report signal to HZ, which responds be sending 3 pieces of Data that are written to memory at Address, Address+1 & Address+2.
ZPASS_DONE	Z-Pass Done	Clears the context valid flag, which in turn will cause the context to be rolled on the next update to state.
Others	Others	CP Writes EVENT_INITIATOR to the VGT_EVENT_INITIATOR register.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CNTR_SEL | SWM_SEL | EVENT_INITIATOR]	CNTR_SEL[31] – Applies only to the Cache_Flush_Ts, Cache_Flush_and_Inv_Ts, VS_Done_Ts & PS_Done_Ts events. 0 - Send DATA; 1 - Send current CP_PROG_COUNTER value. SWM_SEL[30] – Applies only to the shader done events. 0 – CP_ME_VS|PS_EVENT_ADDR|DATA; 1 – CP_ME_VS|PS_EVENT_ADDR|DATA_SWM EVENT_INITIATOR[5:0] – Defined in the Crayola VGT Specification. The CP writes this value to the VGT_EVENT_INITIATOR register for the assigned context.
3	[ADDRESS]	ADDRESS[31:2] – DWORD address in memory. ADDRESS[1:0] – SWAP Used for Memory Write. ***Driver should only supply this ordinal for time stamp & screen extent report events.***
4	[DATA]	Data [31:0] value that will be written to memory when event occurs. ***Driver should only supply this ordinal for time stamp events.***

6.6.9 EVENT_WRITE_SHD – Shader Done

Functionality

The CP will generate the event given in the packet by writing to the VGT_EVENT_INITIATOR register.

It will also set up a write to memory (timestamp) that will occur at the completion of the specified event.

When the Source bit in ordinal 2 is set, it will write to the software managed event address & data registers rather than the hardware managed registers.

When the counter select (CNTR_SEL) bit is set in ordinal 2, it will select the current value in CP_PROG_COUNTER rather than the embedded packet data.

When the software management select (SWM_SEL) bit is set, it will write to the software managed event address & data registers rather than the hardware managed registers.

This packet is not supported within a Real-Time stream.

Pseudopodia

The CP decodes the EVENT_TYPE of the EVENT_INITIATOR DWORD and performs the actions listed below:

Event Code	Event Description	Action Performed
VS_DONE_TS	Vertex Shader Done (VSD)	Write ADDRESS to CP_ME_VS_EVENT_ADDR FIFO Write DATA to CP_ME_VS_EVENT _DATA FIFO Write the EVENT_INITIATOR to the VGT_EVENT_INITIATOR register. **A VsDone Event from the SQ will then generate a DWORD write of the DATA to the ADDRESS.**
PS_DONE_TS	Pixel Shader Done (PSD)	Write ADDRESS to CP_ME_PS_EVENT_ADDR FIFO Write DATA to CP_ME_PS_EVENT _DATA FIFO Write the EVENT_INITIATOR to the VGT_EVENT_INITIATOR register. **A PsDone Event from the SQ will then generate a DWORD write of the DATA to the ADDRESS.**

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CNTR_SEL | SWM_SEL | EVENT_INITIATOR]	CNTR_SEL[31] – 0 - Send DATA; 1 - Send current CP_PROG_COUNTER value. SWM_SEL[30] – 0 – CP_ME_VS|PS_EVENT_ADDR|DATA; 1 – CP_ME_VS|PS_EVENT_ADDR|DATA_SWM EVENT_INITIATOR[5:0] – Defined in the Crayola VGT Specification. The CP writes this value to the VGT_EVENT_INITIATOR register for the assigned context.
3	[ADDRESS]	ADDRESS[31:2] – DWORD address in memory. ADDRESS[1:0] – SWAP Used for Memory Write.
4	[DATA]	Data [31:0] value that will be written to memory when event occurs.

6.6.10 EVENT_WRITE_CFL – Cache Flush

Functionality

The CP will generate the event given in the packet by writing to the VGT_EVENT_INITIATOR register. It will also set up a write to memory (timestamp) that will occur at the completion of the specified event.

When the counter select (CNTR_SEL) bit in ordinal 2 is set, it will select the current value in CP_PROG_COUNTER rather than the embedded packet data.

This packet is not supported within a Real-Time stream.

Pseudopodia

The CP decodes the EVENT_TYPE of the EVENT_INITIATOR DWORD and performs the actions listed below:

Event Code	Event Description	Action Performed
CACHE_FLUSH_TS CACHE_FLUSH_AND_INVALIDATE_TS	Cache Flush Done (CFD)	Write ADDRESS to CP_ME_CF_EVENT_ADDR FIFO Write DATA to CP_ME_CF_EVENT_DATA FIFO Write the EVENT_INITIATOR to the VGT_EVENT_INITIATOR register. **A pulse on the RC_CP_cache_flush signal will then generate a DWORD write of the DATA to the ADDRESS.**

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CNTR_SEL | EVENT_INITIATOR]	CNTR_SEL[31] – Selects between supplied data and current programmable counter value to be written out to the supplied address. 0 - Send DATA; 1 - Send current CP_PROG_COUNTER value. EVENT_INITIATOR[5:0] – Defined in the Crayola VGT Specification. The CP writes this value to the VGT_EVENT_INITIATOR register for the assigned context.
3	[ADDRESS]	ADDRESS[31:2] – DWORD address in memory. ADDRESS[1:0] – SWAP Used for Memory Write.
4	[DATA]	Data [31:0] value that will be written to memory when event occurs.

6.6.11 EVENT_WRITE_SER – Screen Extent Report

Functionality

The CP will generate the SCREEN_EXT_RPT event by writing to the VGT_EVENT_INITIATOR register.

This packet is not supported within a Real-Time stream.

Pseudopodia

The CP decodes the EVENT_TYPE of the EVENT_INITIATOR DWORD and performs the actions listed below:

Event Code	Event Description	Action Performed
SCREEN_EXT_RPT	Screen Extent Report	Sends Report signal to HZ, which responds be sending 3 pieces of Data that are written to memory at Address, Address+1 & Address+2.

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[EVENT_INITIATOR]	EVENT_INITIATOR[5:0] – Defined in the Crayola VGT Specification. The CP writes this value to the VGT_EVENT_INITIATOR register for the assigned context.
3	[ADDRESS]	ADDRESS[31:2] – DWORD address in memory. ADDRESS[1:0] – SWAP Used for Memory Write.

6.6.12 EVENT_WRITE_ZPD – Z-Pass Done

Functionality

The CP will generate the event given in the packet by writing to the VGT_EVENT_INITIATOR register.

Note that the zpass_done event effects the current context.

This packet is not supported within a Real-Time stream.

Pseudopodia

The CP decodes the EVENT_TYPE of the EVENT_INITIATOR DWORD and performs the actions listed below:

Event Code	Event Description	Action Performed
ZPASS_DONE	Z-Pass Done	Clears the context valid flag, which in turn will cause the context to be rolled on the next update to state.

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[EVENT_INITIATOR]	EVENT_INITIATOR[5:0] – Defined in the Crayola VGT Specification. The CP writes this value to the VGT_EVENT_INITIATOR register for the assigned context.

6.6.13 REG_TO_MEM

Functionality

This command packet provides the opportunity to write a memory location with data that is read from a memory-mapped register. The command packet specifies the source register address and the destination memory address.

Note: As an alternate or for multiple register-to-memory transfers, the DMA engine in the CP can be used.

This packet can be executed within a Real-Time stream. The read operations will pass through the Real-Time path of the RBBM.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[REG_ADDR |]	Bits 31:15 – Reserved REG_ADDR[14:0] – Memory-Mapped DWORD Address of register.
3	[ADDRESS]	ADDRESS[31:2] -- DWORD-Aligned Address of destination memory location. ADDRESS[1:0] -- SWAP Code Used for Memory Write.

6.6.14 WAIT_FOR_IDLE

Functionality

Wait for the non-Real Time portion of the graphics pipe to be idle.

The WAIT_FOR_IDLE packet cannot be used within a Real-Time stream.

Microcode Pseudopodia:

The microcode writes a value of 0x00000001 to the NQWAIT_UNTIL register to force the RBBM to wait for graphics pipe to be idle and the Command FIFO in the RBBM to be empty before transferring any more data to the register backbone. Command packets placed after this one will stall until the wait condition is satisfied.

Note that this is different than using a WAIT_UNTIL write because the data stall point is before the Command FIFO and Event Engine in the RBBM. See the RBBM Specification for more details on the NQWAIT_UNTIL function.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header field of the packet.
2	[Dummy Field]	Dummy field. Any value can be placed in this field.

6.6.15 CP_INTERRUPT

Functionality

Sets the interrupt status corresponding to the stream where the CP_INTERRUPT packet was parsed as indicated by the INT_ID field in ordinal 2.

This provides the ability for the Driver to identify the stream that generates the interrupt. Because the Driver puts this packet in the command stream, it can also use the interrupt as an "almost-finished-parsing" flag from the CP if it is placed near the end of a command stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header field of the packet.
2	[INT_ID]	The Driver must set the appropriate interrupt bit. For Non-Real-Time streams: Bit 31 - Interrupt for Ring Buffer Bit 30 - Interrupt for Indirect Buffer #1 Bit 29 - Interrupt for Indirect Buffer #2 Bit 28:0 - Reserved For Real-Time Steams: Bits 31:16 – Reserved Bit 15 – Interrupt for RTS#15 Bit 14 – Interrupt for RTS#14 Bit 13 – Interrupt for RTS#13 Bit 12 – Interrupt for RTS#12 Bit 11 – Interrupt for RTS#11 Bit 10 – Interrupt for RTS#10 Bit 9 – Interrupt for RTS#9 Bit 8 – Interrupt for RTS#8 Bit 7 – Interrupt for RTS#7 Bit 6 – Interrupt for RTS#6 Bit 5 – Interrupt for RTS#5 Bit 4 – Interrupt for RTS#4 Bit 3 – Interrupt for RTS#3 Bit 2 – Interrupt for RTS#2 Bit 1 – Interrupt for RTS#1 Bit 0 – Interrupt for RTS#0

6.6.16 VIZ_QUERY

Functionality

This packet contains Viz Query information indicating the beginning and ending of the Viz Query extent (bounding box) processing. The CP uses this packet to set its internal Viz Query status.

The CP maintains the following set of status for each of the 64 Viz Queries

1. A "DISCARD" bit – Default to "0" on reset.
2. A "End_Received" (END_RCVD) bit – Default to "0" on reset.

The CP does the following for a Viz_Query packet for "Begin" (i.e. VIZQ_END = 0):

1. If the "END_RCVD" bit is already set for the ID (i.e. CP is expecting status from the Scan Converter) then the CP waits. Note: The SC *must* return a visibility result to the CP for every "Viz_Query End" event otherwise the chip may hang.
2. Otherwise:
    a. The "DISCARD" bit is cleared (KEEP is assumed)
    b. The CP issues a "Viz Query Begin" event. This is a write to the *VGT_EVENT_INITIATOR* register with the EVENT_ID set to "Viz Query Begin" by the Micro Engine (ME).

Note that the Scan Converter resets its visibility result when it receives the "Viz Query Begin" event.

Note that the Driver writes to the PA_SC_VIZ_QUERY register to set the ID of the Viz Query for the Scan Converter.

The CP does the following for a Viz_Query packet with the "VIZQ_END" flag set.

1. Set the corresponding "END_RCVD" bit by the Pre-Fetch Parser (PFP). This will stall the next Viz_Query "BEGIN" packet until the status is sent back from the SC.
2. Create a "Viz Query End" event. This is a write to the *VGT_EVENT_INITIATOR* register with the EVENT_ID set to "Viz Query End" by the ME.

The visibility results are sent back to the CP via a dedicated interface from the Scan Converter (SC). When the CP receives a transfer from the SC, it does the following:

1. Clears the corresponding "END_RCVD" bit for the Viz Query.
2. Sets the "DISCARD" bit to the value provided by the SC.

Note that the PFP forwards the entire packet to the ME. The ME generates the event initiators based on the whether it is a "begin" or "end" packet.

The context is marked "dirty" when this packet is processed because the Viz Query events use state when processed.

This packet cannot be included in a Real-Time Stream.

Packet Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[VIZQ_END | VIZQ_ID]	Viz Query Information: Bit 8 -- VIZQ_END: Flag indicating END of a Viz Query Extent Processing. 0 => Begin 1 => End Bits 5:0 --VIZQ_ID: Event Code – 0 to 63 – of visibility bits to test.

6.7 MPEG PACKETS

6.7.1 MPEG_INDEX

Functionality

Packed register writes for MPEG and Generation of Indices.

The packet format has changed since R300: Mask and "Dummy" DWORDs have been removed and the Draw Initiator is a new format.

This packet is not allowed within a Real-Time stream. This restriction is not however because of the CP or RBBM.

Microcode Pseudopodia:

1. Write the DRAW_INITIATOR to the *VGT_DRAW_INITIATOR* register for the assigned context.
2. For each "first index", generate other two indices for the rectangle list and output all indices to the VGT_IMMED_DATA register.
3. Discard any values after the indices in the command packet.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header field of the packet.
2	[DRAW_INITIATOR]	Written Unconditional to *VGT_DRAW_INITIATOR* register The Prim_Type should be a DI_PT_RECTLIST. See the VGT Specification for details.
3	[NUM_INDICES]	Number of Index Base Values (0x0001 to 0x3FFF)
4 to 4+(NUM_INDICES-1)	[32-Bit INDEX]	First Index of Quad. (0x00000000 to 0xFFFFFFFD) For each "First Index", CP will generate the other 2 indices and output: FIRST_INDEX FIRST_INDEX+1 FIRST_INDEX+2 All indices are written to the VGT_IMMED_DATA register.

6.8 MISCELLANEOUS PACKETS

6.8.1 NOP

Functionality

Skip a number of DWORDs to get to the next packet.

Pseudocode:

The Pre-Fetch Parser (PFP) just discards the data in the packet. No data is sent out of the Command Processor (CP).

Format

Ordinal	Field Name
1	[HEADER]
2	{DATA_BLOCK}

DATA_BLOCK

This field may consist of a number of DWORDs, and the content may be anything.

6.8.2 INDIRECT_BUFFER

Functionality

This packet is used for dispatching Indirect Buffers. This more easily allows the Pre-Fetch Parser (PFP) to recognize indirect buffer dispatches.

Whether an Indirect Buffer #1 or Indirect Buffer #2 is fetched is dependant on the context of this packet. If this packet is in the Primary (Ring) Buffer, the Indirect Buffer #1 is fetched. If the packet is in the Indirect Buffer #1, the Indirect Buffer #2 is fetched. The difference is only in the parsing priority of the data stream and the storage location of the pre-fetched data in the CP. Indirect Buffer #2 commands have a higher priority than Indirect Buffer #1 commands.

Included in this packet is the Multi-Pass enable control. If this bit is set, the PFP will stop parsing consecutive commands and instead wait for the Continue/Loop signal to be asserted by the Scan Converter (See the CP Unit Specification for the signal interface). If the Loop signal is asserted, the PFP will re-fetch the indirect buffer and it will be re-processed by the CP. If the Scan Converter tells the CP to continue, the PFP will resume pre-parsing of the command stream.

Note that the CP still supports the dispatching of indirect buffers via Type-0 packets for legacy support. See the Packet Restrictions section of this document for more details.

This packet is not allowed within a Real-Time Stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[IB_BASE]	Indirect Buffer Base Address [31:5] – Double-Octword-Aligned
3	[MULTIPASS | IB_SIZE]	MULTIPASS [31] – Set to tell the CP that this Indirect Buffer is a Multi-Pass operation. Indirect Buffer Size [19:0] – Size of the Indirect Buffer in DWORDs.

6.8.3 INDIRECT_BUFFER_PFD

Functionality

This packet has the same functionality as the INDIRECT_BUFFER packet, except that the initiation of the indirect buffer fetch is pipelined out of the CP, through the RBBM, and then back to the CP via the Global Register Bus. The fetching of the indirect buffer will be not start until the CP receives the initiation (Base and Size) from the Global Register Bus.

This allows the fetching of the indirect buffer to be stalled by WAIT_UNTIL conditions in the RBBM. One usage of this packet is as follows:

1. HostData_Blt packet to move an indirect buffer from AGP to Local Memory.
2. Type-0 packet to WAIT_UNTIL register in RBBM with the Wait_IdleClean bit set.
3. Indirect_Buffer_PFD packet to initiate the fetching and processing of the indirect buffer packet moved in step #1.

The usage of the Indirect_Buffer_PFD packet in this case along with the Wait_IdleClean causes the CP to not fetch the indirect buffer until after it is moved to Local Memory.

This packet is not allowed within a Real-Time Stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[IB_BASE]	Indirect Buffer Base Address [31:5] – Double-Octword-Aligned
3	[MULTIPASS | IB_SIZE]	MULTIPASS [31] – Set to tell the CP that this Indirect Buffer is a Multi-Pass operation. Indirect Buffer Size [19:0] – Size of the Indirect Buffer in DWORDs.

6.8.4 IB_PREAMBLE

Functionality

This packet should be placed at the start of Indirect Buffers if the IB_START_CHECK_ENABLE bit is set in the CP_ME_CNTL register. The Micro Engine (ME) in the CP will check for this packet following an IB_PREFETCH_START packet and assert an error (IB_ERROR) interrupt if this is not present.

This provides some level of protection against parsing of bad data representing an indirect buffer. See the Error Checking section of the CP Unit Specification for details.

If the micro engine encounters this packet when the IB_START_CHECK_ENABLE bit is not set, the packet will be discarded.

The IB_PREAMBLE packet is not allowed within a Real-Time Stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CHECKSUM0]	Needs to be 0x12B9B0A1 (π * 100,000,000)
3	[CHECKSUM1]	Needs to be 0x1033C4D6 (e * 100,000,000)

6.8.5 LOAD_EXECUTE – Not Currently Supported

Functionality

***** Experimental Use Only for CRAYOLA *****

Load CP executable code into the Micro Engine's instruction RAM and begins executing the code.

The CP's Pre-Fetch Parser (PFP) issues the fetch request for the code to the CP's command stream fetcher and inserts a MICRO_PREFETCH instruction into the command stream. The MICRO_PREFETCH instruction tells the Micro Engine (ME) to load the code from the state buffer into the ME's instruction RAM and then jump to the code.

The current instruction pointer (IP or IP_RT) is pushed onto the return stack before jumping to the packet's code. On a return from the executable, the CP will restore this value to the ME's Instruction Pointer from the return stack. The last micro instruction therefore must be: "RTN". See the CP Unit specification for details.

This packet is allowed within a Real-Time stream. Separate instruction RAM is available for both normal and Real-Time operations.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CODE_ADDRESS]	Double-Octword-Aligned Address [31:5] of location of executable code in memory.
3	[CODE_SIZE]	Size [5:0] of executable in DWORDs. The CP supports up to 64 micro instructions. The loading of the micro-instructions will unconditionally write code that was previously fetched.

6.8.6 ME_INIT

Functionality

The ME_INIT packet should be sent to the CP immediately after loading the microcode and enabling the Micro Engine (ME). The packet should be sent for both a non-RT and the RT micro engines. This Type-3 packet is used by the ME to initialize internal state information that is used by other packets.

When processed in a non-Real-Time stream, the packet has the side affect of invalidating the matching pointers in the Pre-Fetch Parser and clearing the constant and incremental register write enables.

Note: If the ME_INIT packet changes the MAX_CONTEXT value then it needs to be proceeded with an INVALIDATE_STATE packet with a full mask to invalidate the matching pointers for sub-block, instruction, and constant reuse.

The packet is used to:

1. Enable / Disable Real-Time streams (Non-RT ME Only):

    a. If the current non-real-time context is '0', the CP will wait for context '1' to be idle, then copy from '0' to '1' before allowing real-time processing. The real-time enable flag is visible in the CP_STATE_CNTL register.

    b. If context '0' is not the current context, but is being used as a non-real-time context, then the CP will wait for context '0' to be idle.

2. Enabled/Disable 2D and 3D Implicit Synchronization (Non-RT ME Only):

3. Sub-Block Offset Values -- SUBBLK_0_GFX_OFFSET,..., SUBBLK_7_GFX_OFFSET (Unique to each Non-RT and RT ME).

4. Set-up the Instruction Memory partitioning parameters (Unique to each Non-RT and RT ME). ***Note that the pixel shader start address must be greater than the vertex shader start address.***

    a. Write Vertex_Shader_Base to ME's DMA engine.
    b. Write Pixel_Shader_Base-1 to ME's DMA engine (Vertex_Shader_End).
    c. Write Pixel_Shader_Base to ME's DMA engine.
    d. Write the Pixel_Shader_End with the value: 0x1FFFC (End of Memory Map)

5. Maximum Context {0...7} in the Chip. Writing the MAX_CONTEXT also has the following side affects:

    a. Set the Context_Dirty flag.
    b. Sets the Current Context to:
        i. Zero if Real-Time is Disabled
        ii. One if Real-Time is Enabled
    c. Resets the Dirty Rectangle for 2D Coherency Control (See the 2D Implementation Spec for Details):
        i. Dirt_Left = Dirt_Top = -8,192
        ii. Dirt_Right = Dirt_Bottom = +8,191

6. Write Confirm Interval – Number of DWORD writes CP should send to MC between issuing a Write Confirmation. ** ***Experimental for CRAYOLA. Microcode currently discards this value as of 02-23-2003. Microcode may be written in the future to take advantage of this parameter.*** **

7. NQ Select and External Memory SWAP control

    a. NQ Select – Setting for sending transactions through the Queued/Non-Queued Path through the RBBM.
    b. External Memory Swap – Swap Code that ME will use 2D Brush, Palette, and Immediate Data as well as Constant and Header Dump writes.

8. Error Detection Control Booleans (Unique to each Non-RT and RT ME). ***Note: This feature degrades performance and is therefore used for debug only. To use, the microcode must be recompiled with UCODE_DEBUG defined.***

    a. Reserved_Check_Enable
    b. IB_Start_Check_Enable
    c. Protected_Mode_Enable
    d. Protected Mode Address
    e. Check for Type-0 Packets in Indirect Buffers.

9. Header Dump Parameters – CP will write the headers of the packets to external memory for debug. ***Note: This feature degrades performance and is therefore used for debug only. To use, the microcode must be recompiled with UCODE_DEBUG defined.***

    a. Header_Dump_Enable
    b. Header_Dump_Base[31:12] – Base Address in External Memory where to write the packet headers.
    c. Header_Dump_Swap[1:0] – Swap Control for Header Dump.
    d. Header_Dump_Size[29:0] – Size of Allocated External Memory in DWORDs

10. Microcode Reset Control

    a. Set to clear areas in micro engines scratch memory. This should only be set on the first ME_INIT packet submitted.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[MASK]	DWORD Mask. Bits 31:10 – Reserved Bit 9 – Microcode Default Reset Control (No Ordinals are Associated with this Bit) (Non-RT Only) Bit 8 – Header Dump Present Bit 7 – Error Detection Control Booleans Present Bit 6 – Non-Queued Flag and External Memory Swap Present Bit 5 – Wait Confirm Interval Present. Bit 4 – Maximum Contexts Present (Non-RT Only) Bit 3 – Instruction Memory Thresholds Present Bit 2 – Sub-Block Offsets Present – RT can have separate offsets than Non-RT Bit 1 – 2D/3D Implicit Synchronization Present (Non-RT Only) Bit 0 – Real-Time Enable/Disable Present (Non-RT Only)
3	[RT_ENABLE]	Real-Time Enable / Disable Bits 31:1 – Reserved Bit 0 – Real-Time Enable. Set to enable, clear to disable real-time stream processing.
4	[ISYNC_2D_to_3D | ISYNC_3D_to_2D]	Bits 31:2 – Reserved Bit 1 – ISYNC_2D_to_3D: Any 3D processing stalls if 2D operation is in-progress. Bit 0 – ISYNC_3D_to_2D: Any 2D processing stalls if 3D operation is in-progress.
5	[SUBBLK_0_GFX_OFFSET]	Bits 31:10 – Reserved Bits 9:0 – DWORD Offset in GFX Decode Space for the Sub-Block.
6	[SUBBLK_1_GFX_OFFSET]	Bits 31:10 – Reserved Bits 9:0 – DWORD Offset in GFX Decode Space for the Sub-Block.
7	[SUBBLK_2_GFX_OFFSET]	Bits 31:10 – Reserved Bits 9:0 – DWORD Offset in GFX Decode Space for the Sub-Block.
8	[SUBBLK_3_GFX_OFFSET]	Bits 31:10 – Reserved Bits 9:0 – DWORD Offset in GFX Decode Space for the Sub-Block.
9	[SUBBLK_4_GFX_OFFSET]	Bits 31:10 – Reserved Bits 9:0 – DWORD Offset in GFX Decode Space for the Sub-Block.
10	[SUBBLK_5_GFX_OFFSET]	Bits 31:10 – Reserved Bits 9:0 – DWORD Offset in GFX Decode Space for the Sub-Block.
11	[SUBBLK_6_GFX_OFFSET]	Bits 31:10 – Reserved Bits 9:0 – DWORD Offset in GFX Decode Space for the Sub-Block.
12	[SUBBLK_7_GFX_OFFSET]	Bits 31:10 – Reserved Bits 9:0 – DWORD Offset in GFX Decode Space for the Sub-Block.
13	[VERTEX_SHADER_BASE | PIXEL_SHADER_BASE]	Instruction Memory Thresholds Bits 31:28 – Reserved Bits 27:16 – Vertex Shader Start Address in Instructions. Bit 15:12 – Reserved Bits 11:0 – Pixel Shader Start Address in Instructions (***Must be greater than the Vertex Shader Start***). ***These values must be programmed for Real-Time for correct operation of the Real-Time DMA Engine.***
14	[MAX_CONTEXT]	Bits 31:3 – Reserved Bits 2:0 – Maximum Context in Chip. Values are 0 to 7 Note that if real-time is enabled, it takes Context 0 away from non-RT processing.
15	[WC_INTERVAL]	Bits 31:8 – Reserved Bits 7:0 – Write Confirm Interval
16	[NQ_DEFAULT | EXTERNAL_MEM_SWAP]	Bit 31 – Non-Queued RBBM Path. Defaults to "Queued" at reset in hardware. 0 => All transactions from the Micro Engine will go through the RBBM's Queued path. 1 => All transactions from the Micro Engine will go through the RBBM's Non-Queued Path. Bits 30:2 – Reserved Bits 1:0 – Swap Code Used for the following transactions: • 2D Brush, Palette, and Immediate Data Writes • Constant Data Writes
17	[Reserved_Check_Enable | IB_Start_Check_Enable | Protected_Mode_Enable | T01_in_IB_Interrupt_Enable | T01_in_IB_Stop_Enable | Protected_Mode_Address]	Bit 31 – Reserved_Check_Enable : Enable reserved bit checking (Degrades Performance). Bit 30 – IB_Start_Check_Enable : Enable checking for pre-amble at start of IB. Bit 29 – Protected_Mode_Enable : Enable protected mode. Bit 28 – Enable Interrupting if Type-0 / Type-1 found in Indirect Buffer (Degrades Performance). Bit 27 – Enable Stopping if Type-0 / Type-1 Packet found in Indirect Buffer(Degrades Performance). Bits 26:15 – Reserved Bits 14:0 -- Protected_Mode_Address : DWORD address. All addresses below this are protected.
18	[Header_Dump_Base | Header_Dump_Swap]	Header_Dump_Base[31:12] – Base Address of external memory location where CP will dump PM4 Headers. Bits 11:2 – Reserved: Should be set to zero. Header_Dump_Swap[1:0] – Swap Code Used When Writing Headers to Memory.
19	[Header_Dump_Enable | Header_Dump_Size]	Header_Dump_Enable[31] – Enable Writing PM4 Headers to Memory for Debug (Degrades Performance). Bit 30 – Reserved. Header_Dump_Size[29:0] – Size in DWORDs for the Header Dump Ring in External Memory.

6.8.7 FIX2FLT_REG

Converts an integer value (0 to 255) to IEEE floating point and writes to the specified register.

The DWORD register address is only valid up to address 0x1FFF (Byte Address 0x7FFC). The packet cannot write into the context-based registers, constants, or instruction memory of the CRAYOLA.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[REGISTER_ADDR]	Bits 31:13 -- Reserved REGISTER_ADDR[12:0] – DWORD address of the destination register in the address map.
3	[FIXED_VALUE]	Bits 31:8 – Reserved FIXED_VALUE[7:0] – 8-bit Integer Value (0 to 255).

6.8.8 SET_BIN_MASK

Used with SET_BIN_SELECT packet to setup the predication test.

The SET_BIN_MASK packet sets two consecutive 32-bit registers CP_BIN_MASK_LO and CP_BIN_MASK_HI. The combined 64-bit value specifies the current driver defined bin category.

The CP's Prefetch Parser compares the CP_BIN_MASK and CP_BIN_SELECT registers to determine whether subsequent predicated packets are processed. The comparison tests consists of a bitwise AND operation followed by an OR reduce operation on the result to detect if any of the bits are set.

If any bits are set, the predicated packet is processed; otherwise the predicated packet is skipped.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[BIN_MASK_LO]	Set BIN_MASK_LO[31:0] register to value supplied.
3	[BIN_MASK_HI]	Set BIN_MASK_HI[31:0] register to value supplied.

6.8.9 SET_BIN_SELECT

Used with SET_BIN_MASK packet to setup the predication test.

The SET_BIN_SELECT packet sets two consecutive 32-bit registers CP_BIN_SELECT_LO and CP_BIN_SELECT_HI. The combined 64-bit value specifies the bin category of the subsequent command stream data.

The CP's Prefetch Parser compares the CP_BIN_MASK and CP_BIN_SELECT registers to determine whether subsequent predicated packets are processed. The comparison test consists of a bitwise AND operation followed by an OR reduce operation on the result to detect if any of the bits are set.

If any bits are set, the predicated packet is processed; otherwise the predicated packet is skipped.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[BIN_SELECT_LO]	Set BIN_SELECT_LO[31:0] register to value supplied.
3	[BIN_SELECT_HI]	Set BIN_SELECT_HI[31:0] register to value supplied.

6.9 INTERNAL CP PACKETS

The Pre-Fetch Parser (PFP) generates Type-3 packets, which are not accessible by the Driver. These packets are only used internal to the CP for the PFP to communicate to the Micro Engine (ME).

6.9.1 IB_PREFETCH_START

The IB_PREFETCH_START packet is generated by the PFP when it detects an indirect buffer initiation in either the Ring or IB1 command streams. The packet tells the Micro Engine to start processing packets from the next-most priority queue.

If processing from Ring, start processing from IB1.

If processing from IB1, start processing from IB2.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[DUMMY]	The micro engine ignores this field. The Pre-Fetch Parser puts 0xdeadbeef into this field.

6.9.2 IB_PREFETCH_END

The IB_PREFETCH_END packet is generated by the PFP when it detects the end of a pre-fetched indirect buffer. The PFP inserts this at the end of the fetched indirect buffer to tell the ME to go back to process the initiating stream (See IB_PREFETCH_START).

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[DUMMY]	The micro engine ignores this field. The Pre-Fetch Parser puts 0xdeadbeef into this field.

6.9.3 SUB-BLOCK_PREFETCH

The SUB-BLOCK_PREFETCH packet is generated by the PFP when it is processing a SET_STATE packet and the sub-block needs to be fetched. The packet identifies which sub-block is to be updated (SUBBLK_ID) and the DWORD count. The ordinal corresponding to each sub-block id is sent only when a mismatch occurs, thus the number of ordinals sent varies with each Set_State packet. The last ordinal sent will have the END_PACKET bit set. The fetched data is returned to the Ring/IB state data queue. Up to 128 DWORDs can be fetched per sub-block as indicated in the Set_State packet.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2-to-N	[SUBBLK_ID | COUNT]	Bit 31:27 – Reserved SUBBLK_ID[26:24] – Sub-Block Id 0-7 Bit 23:8 – Reserved COUNT[7:0] – Number of DWORDs to take from the State Queue.
N+1	[END_PACKET]	Bit 31 – END_PACKET Bit 30:0 – Reserved

6.9.4 CONST_PREFETCH

The CONST_PREFETCH packet is generated by the PFP when it is processing a LOAD_CONSTANT_CONTEXT packet and the constant data needs to be fetched. The packet identifies which constant is to be updated and the DWORD count. The LCC packet can repeat the set of CONST_OFFSET & NUM_DWORDS ordinals (the other fields remain constant for the duration of the packet). When this occurs, they will be repeated also in the CONST_PREFETCH packet. The last ordinal sent must have the END_PACKET bit set. The fetched data is returned to the state data queue.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[BASE_ADDR]	BASE_ADDR [31:13] – Base address for the block in Memory from where the CP will fetch the constants.
3-to-N	[CONST_ID | CONST_OFFSET]	Bits 31:24 – Reserved CONST_ID[23:16] 0x00 – ALU Constant (Vertex and Pixel) 0x01 – Texture Constant 0x02 to 0x03 – Reserved 0x04 – Incremental Register Update. 0x05 to 0xFF – Reserved Bit 15:12 – Reserved CONST_OFFSET[10:0] – Offset in DWORDs.
4-to-N+1	[NUM_DWORDS]	Bits 31:12 – Reserved NUM_DWORDS[11:0] – Number of DWORDs to take from the state data queue.
N+2	[END_PACKET]	Bits 31 – END_PACKET 30:24 – Reserved

6.9.5 INSTR_PREFETCH

The Instruction_Prefetch (INSTR_PREFETCH) packet is generated by the Pre-Fetch Parser (PFP) when it is processing a Set_State or Im_Load packet and the instruction code needs to be fetched.

The packet tells the Micro Engine (ME) to get "INSTR_SIZE" DWORDs of instruction code from the Ring/IB1 state data queue. It also identifies the code type (VS, PS, or Real-Time/Shared).

The ME maintains the relative start pointers and available counts for the Instruction Memory. The ME will throttle the instruction update until a context is available and there is enough room in the Instruction Memory for the instruction code.

For Real-Time/Shared Code it also includes the INSTR_START value. This value represents the absolute memory-mapped register address – bits 16:2 of the global register bus address. The start value is written to the SQ_PS_PROGRAM register at context #0 for real-time and the instruction size is set to 0x800. The size is not used by the sequencer for real-time so a value of 0x800 is a "don't care".

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CODE_ID]	Bits 31:2 -- Reserved CODE_ID[1:0] 00 => Vertex Shader 01 => Pixel Shader 1X => Real-Time / Shared Code
3	[INSTR_START | INSTR_SIZE]	Bit 31:28 -- Reserved INSTR_START[27:16] -- Start Address for Real-Time / Shared Code in Instruction Memory. Bits 15:14 -- Reserved INSTR_SIZE [13:0] – Number of DWORDs for the ME to read from the Ring/IB State Data Queue.

6.9.6 INSTR_MATCH

The Instruction Match (INSTR_MATCH) packet is generated by the Pre-Fetch Parser (PFP) when it is processing a Set_State or Im_Load packet and the instruction code pointer matches the prior pointers (i.e. does not need to be fetched).

The Micro Engine uses this packet to manage the Instruction Memory pointers.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CODE_ID]	Bits 31:2 -- Reserved CODE_ID[1:0] 00 => Vertex Shader 01 => Pixel Shader 1X => N/A

6.9.7 MICRO_PREFETCH – Not Currently Supported

** Experimental Use Only for CRAYOLA **

The MICRO_PREFETCH instruction tells the Micro Engine (ME) to load the code from the state buffer into the ME's instruction RAM and then jump to the code.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CODE_SIZE]	Bits 31:6 -- Reserved CODE_SIZE[5:0] – Number of DWORDs to fetch from State Data Queue and Load Into Micro Instruction RAM.

6.9.8 INCR_UPDT_STATE

The Incremental Update to State packet is created by the Prefetch Parser when it encounters a Type-0 packet that writes to the GFX_DECODE space (0x8000-0xFFFF). Its functionality is identical to a Type-0 packet, except that the microengine writes the data to the register address corresponding to the current context. If needed, the microengine will also update the context prior writing out the data.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[ONE_REG_WR | BASE_INDEX]	Bits 31:16 – Reserved Bit 15 -- 0:- Write the data to N consecutive registers. 1:- Write all the data to the same register. Bits 14:0 – The BASE_INDEX[14:0] correspond to byte address bits [16:2] and is the DWORD Memory-mapped address.
4	REG_DATA_1	Data
…	…	…
N + 1	REG_DATA_N	Data

6.9.9 INCR_UPDT_CONST

The Incremental Update to Constants packet is created by the Prefetch Parser when it encounters a Type-0 packet that writes to the Constant Store register space (0x10000-0x13FFF). Its functionality is identical to a Type-0 packet, except that the microengine, if needed, will update the context prior writing out the data.

Format

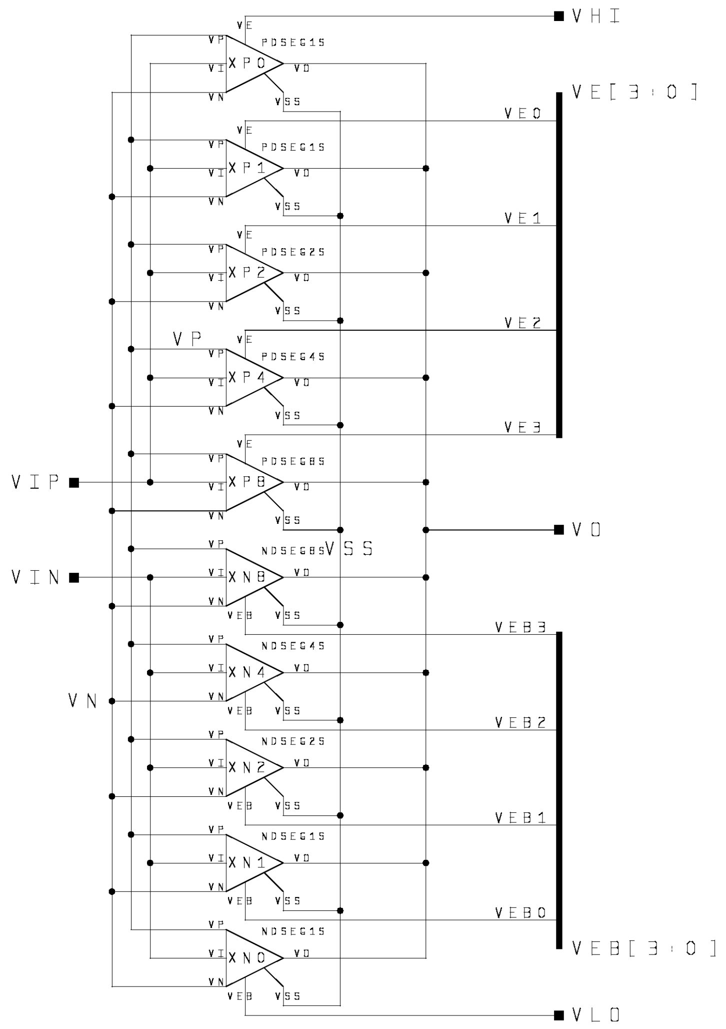
Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[ONE_REG_WR | BASE_INDEX]	Bits 31:16 – Reserved Bit 15 -- 0:- Write the data to N consecutive registers. 1:- Write all the data to the same register. Bits 14:0 – The BASE_INDEX[14:0] correspond to byte address bits [16:2] and is the DWORD Memory-mapped address.
4	REG_DATA_1	Data
…	…	…
N + 1	REG_DATA_N	Data

6.9.10 INCR_UPDT_INSTR

The Incremental Update to Instructions packet is created by the Prefetch Parser when it encounters a Type-0 packet that writes to the Instruction Store register space (0x14000-0x1ffff). Its functionality is identical to a Type-0 packet, except that the microengine, if needed, will update the context prior writing out the data.

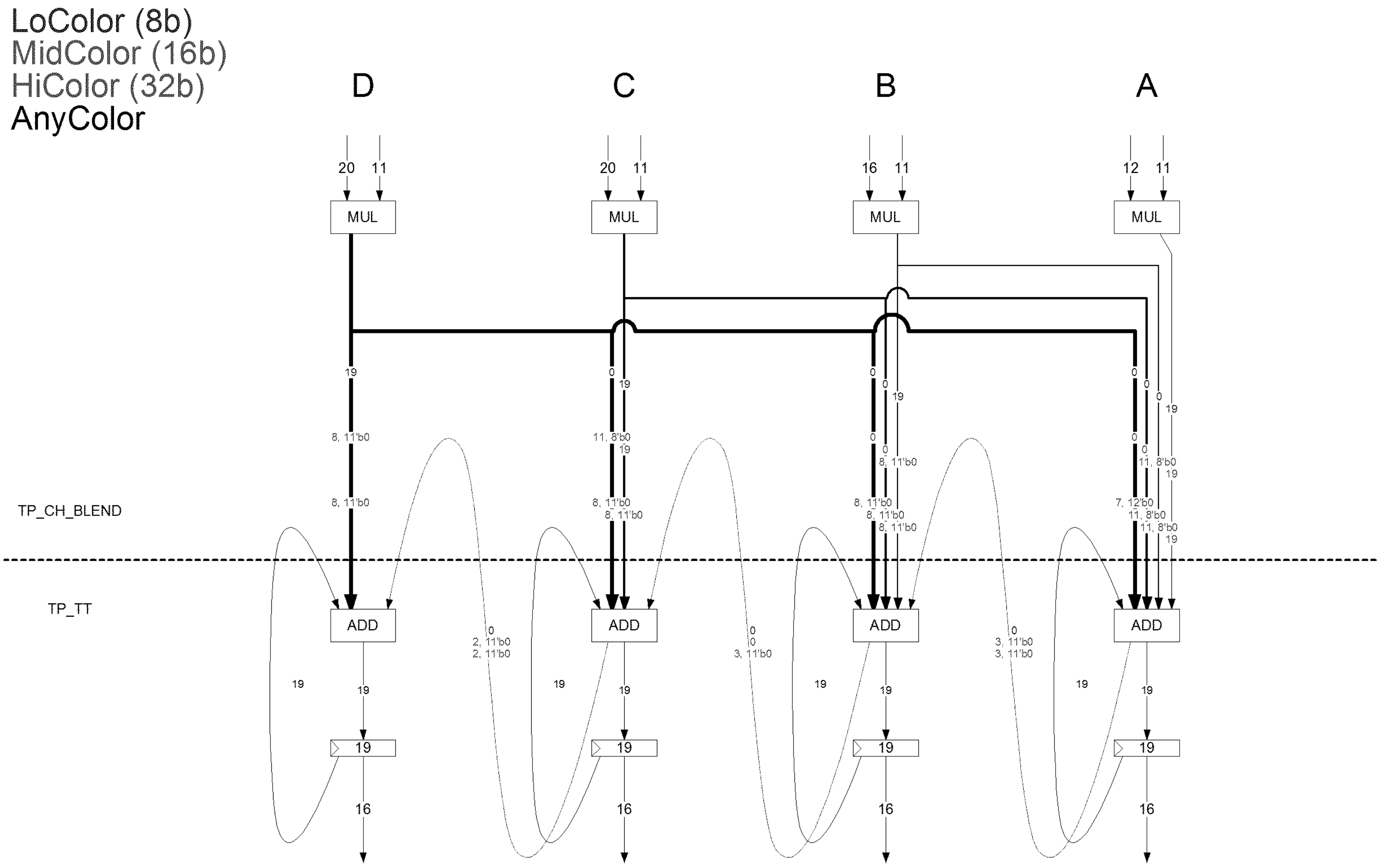
Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[ONE_REG_WR | BASE_INDEX]	Bits 31:16 – Reserved Bit 15 -- 0:- Write the data to N consecutive registers. 1:- Write all the data to the same register. Bits 14:0 – The BASE_INDEX[14:0] correspond to byte address bits [16:2] and is the DWORD Memory-mapped address.
4	REG_DATA_1	Data
…	…	…
N + 1	REG_DATA_N	Data

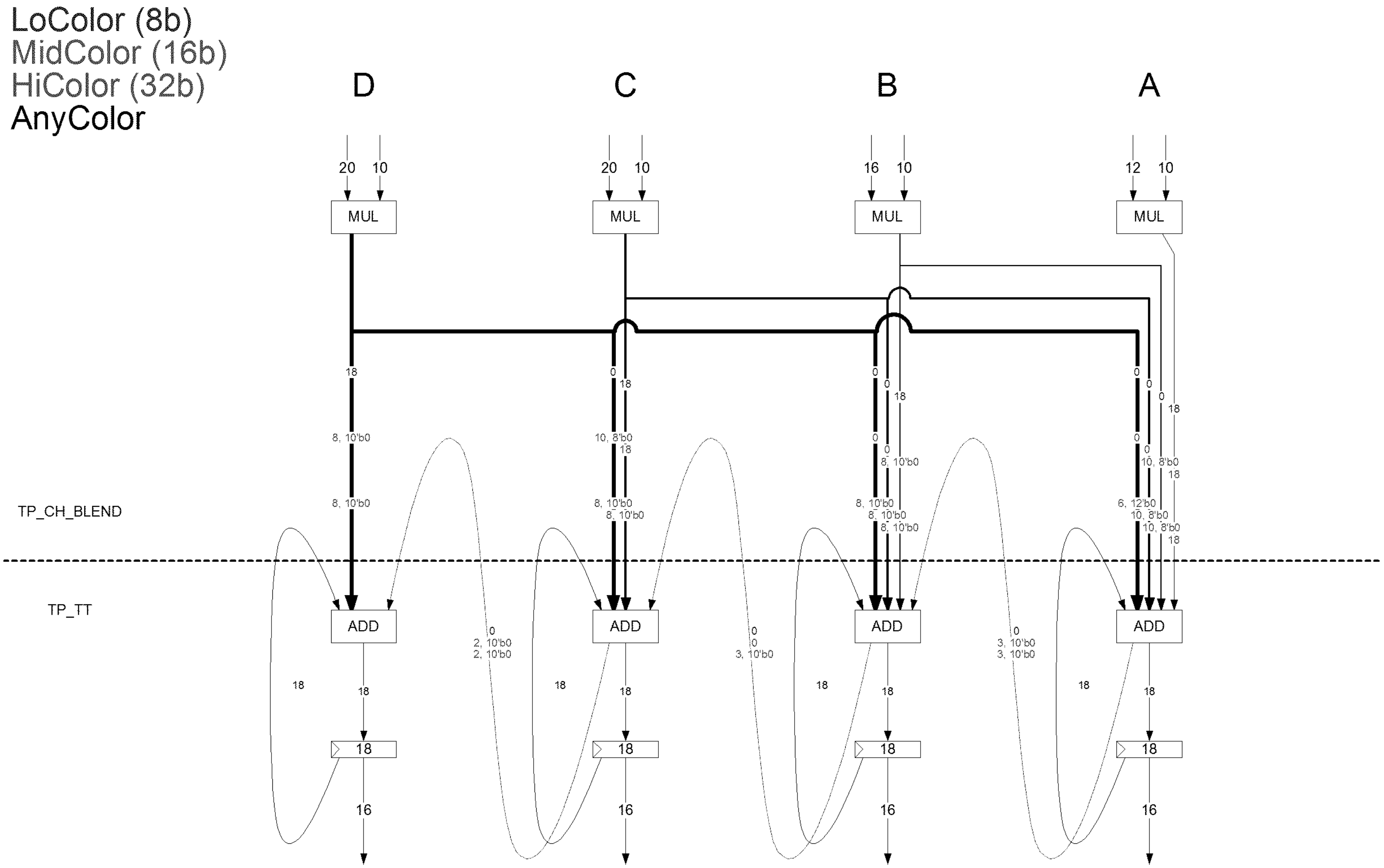


Copyright © 2002

Schematic	Rev	Project	For	Date	Engineer	
OUTDRIVESLOW	REV 1.0	0.13UM TSMC I/O Cells	ATI Technologies Inc.	1-9-2003_20:54	Joe Nolan	bASICs Engineering

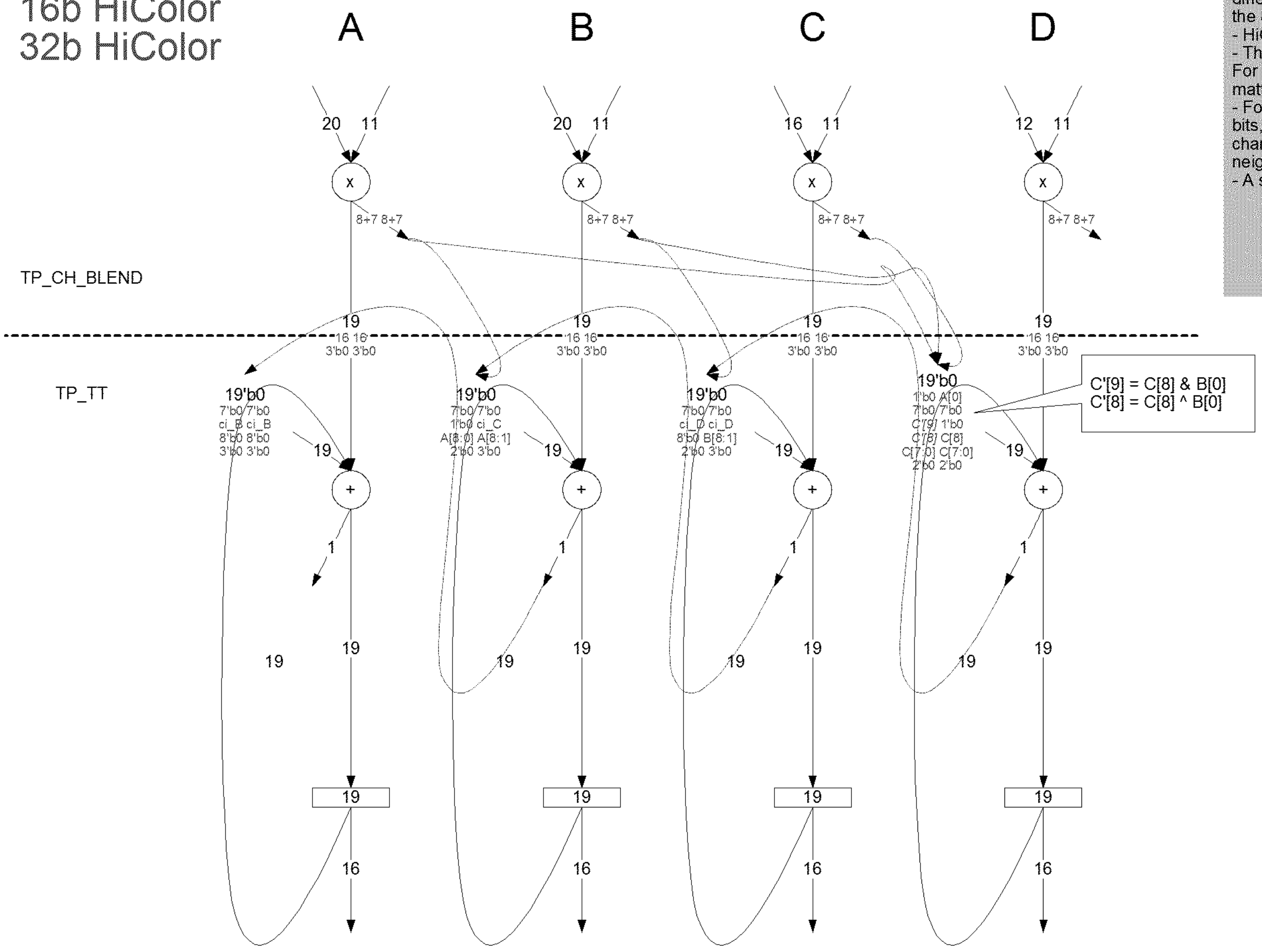


Important notes:
- Wh and Wv are lerp weights of 6b each.
- Ws is 12b (e1m11), which is either mip(6)+ani(5) or mip(5)+z(6). Doing anisotropic filtering on volume mipmaps still loses 1b of mip, and potentially gives artifacts due to improper accumulation.
- For 8888 case, we need 19b of accumulation register in order to properly accumulate a monochromatic map (single color blend). This is 8b of source (lerps don't introduce fraction), with 11b of Ws precision. Blending different colors together introduces some error, which shows up at the LSB in the 8b lerps. This is fine since the result is still gonna be between the two endpoints.
- The multiplier is 20x11 worst case, but lower resolution where it's less needed.
- The adder is 5-terms (!), which requires a 3b carry.
- The output is only 16b. The LSBs disappear when accumulated properly.
- LowColor offer up to 8b of additionnal fractional precision (8.8). Mid and HiColor only offer 3 (16.3 or 32.3). MidColor is 16.3 because we require 19b of accumulation for LoColor. In MidColor, those 19b are split into 8+11, hence the 3 additionnal bits, kept in the second accumulator. HiColor simply uses the same number of additionnal bits in order to align the same number of hardcoded 0s in the last accumulator add.
- Channels can be OR'd together in order to construct Mid and HiColor.

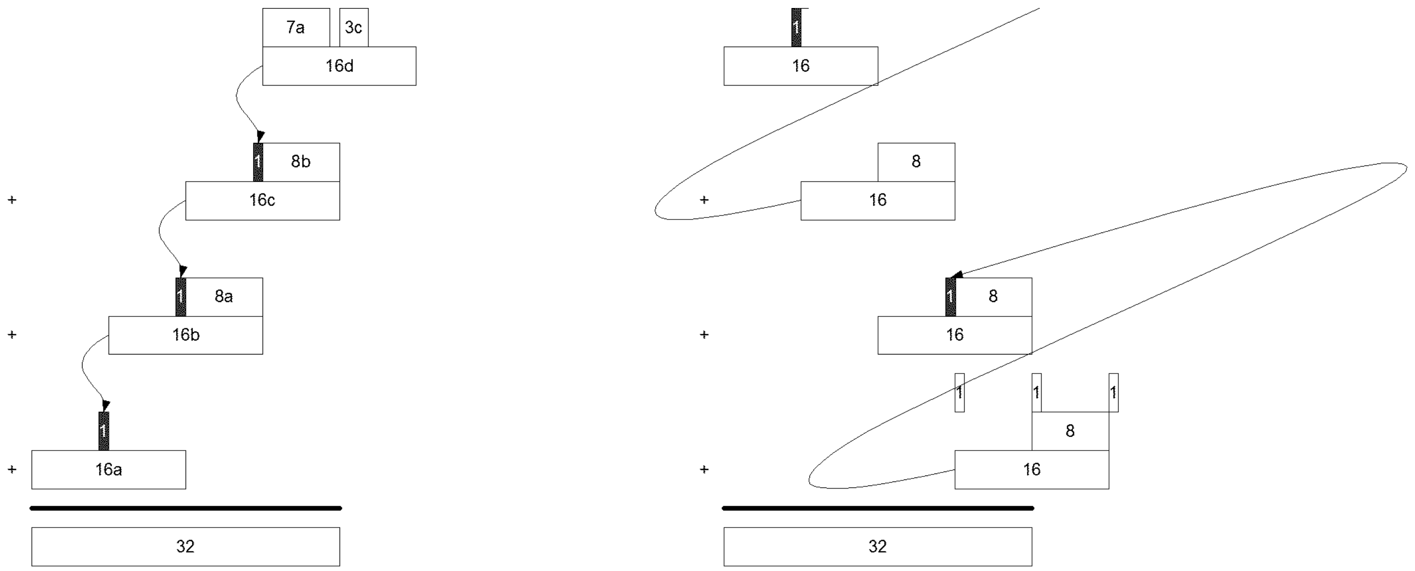


Important notes:
- Wh and Wv are lerp weights of 6b each.
- Ws is 11b (e1m10), which is either mip(5)+ani(5) or mip(4)+z(6). Doing anisotropic filtering on volume mipmaps still loses 1b of mip, and potentially gives artifacts due to improper accumulation.
- For 8888 case, we need 18b of accumulation register in order to properly accumulate a monochromatic map (single color blend). This is 8b of source (lerps don't introduce fraction), with 10b of Ws precision. Blending different colors together introduces some error, which shows up at the LSB in the 8b lerps. This is fine since the result is still gonna be between the two endpoints.
- The multiplier is 20x10 worst case, but lower resolution where it's less needed.
- The adder is 5-terms (!), which requires a 3b carry.
- The output is only 16b. The LSBs disappear when accumulated properly.
- LowColor offer up to 8b of additionnal fractional precision (8.8). Mid and HiColor only offer 2 (16.2 or 32.2). MidColor is 16.2 because we need 18b of accumulation for LoColor. In MidColor, those 18b are split into 8+10, hence the 2 additionnal bits, kept in the bottom accumulator. HiColor simply uses the same number of additionnal bits in order to align the same number of hardcoded 0s in the last accumulator add.
- Channels can be OR'd together in order to construct Mid and HiColor.

16b HiColor
32b HiColor

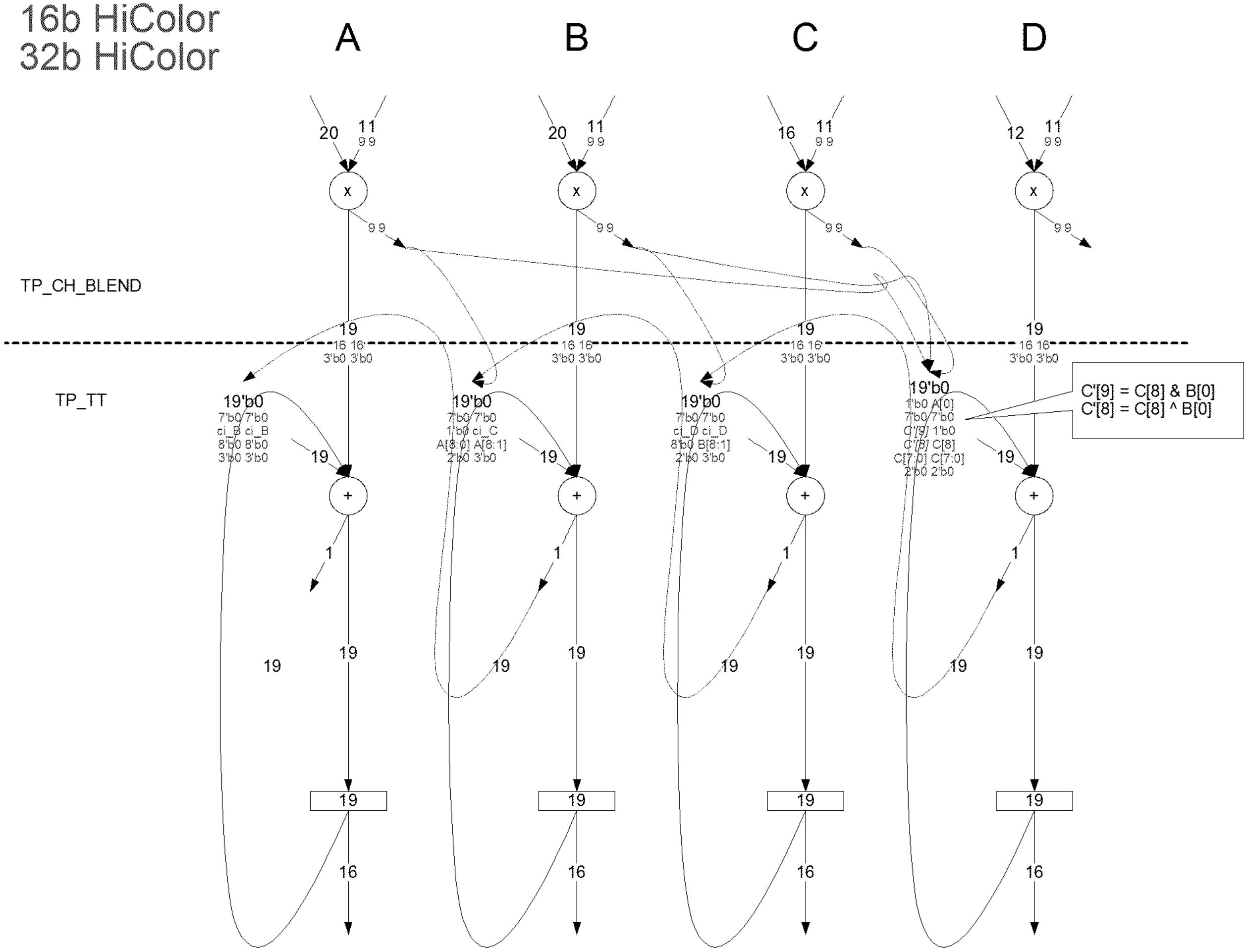


Important notes:
- Wh and Wv are lerp weights of 6b each.
- Ws is 12b (e1m11), and contains a combination of two
aniso(5b), mip(5b). We require 11b for standard 8888 fo
volumes.
- For 8888 case, we need 19b of accumulation register ir
properly accumulate a monochromatic map (single color
source (lerps don't introduce fraction), with 11b of Ws pr
different colors together introduces some error, which sh
the 8b lerps.
- HiColor could drop 1b of mip when doing volume map,
- The multiply spits 27b (16+11), but we need fewer bits
For HiColor, the multiply could be 20x10, but since we ne
matter?
- For HiColor, the multiply spits 16+9=25b. The 10b are
bits, either mip(4) + z(6), or mip(4) + ani(4). Those 10b a
channels to the right. These are split in 8+1, where the 8
neighbor, but the 1 ends up in the rightmost unit.
- A special 2b adder is used for the last unit in order to av


7a
3c
16d
1
8b
+
16c
1
8a
+
16b
1
+
16a
32
1
16
8
+
16
1
8
+
16
1
1
1
8
+
16
32

either mip(5) + 2(4), or mip(5) + ani(4). Those 9b are the spread over the channels to the right. These are split in 8+1, where the 8b goes to the neighbor, but the 1 ends up in the rightmost unit.
- A special 2b adder is used for the last unit in order to avoid 4-term add.

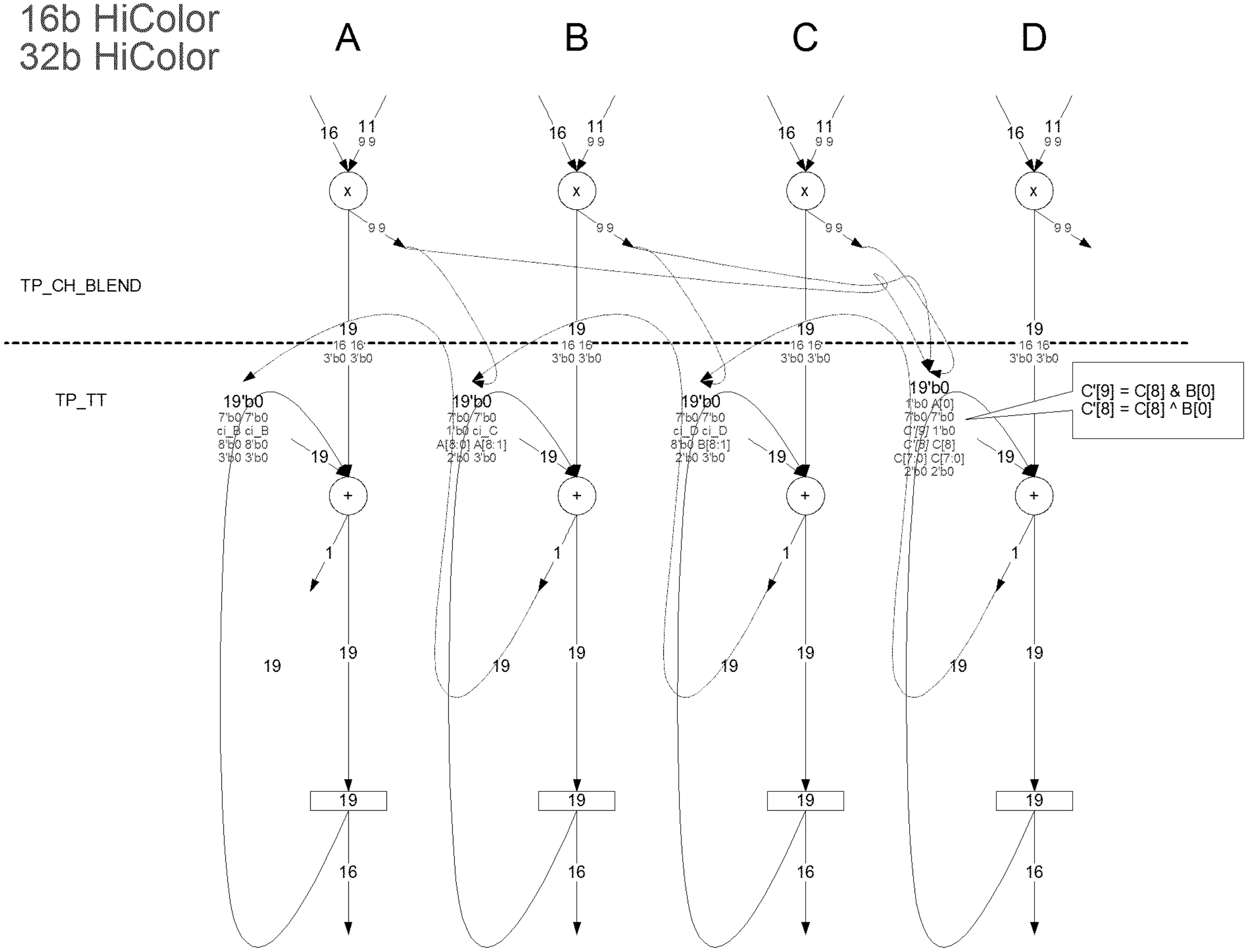
16b HiColor
32b HiColor

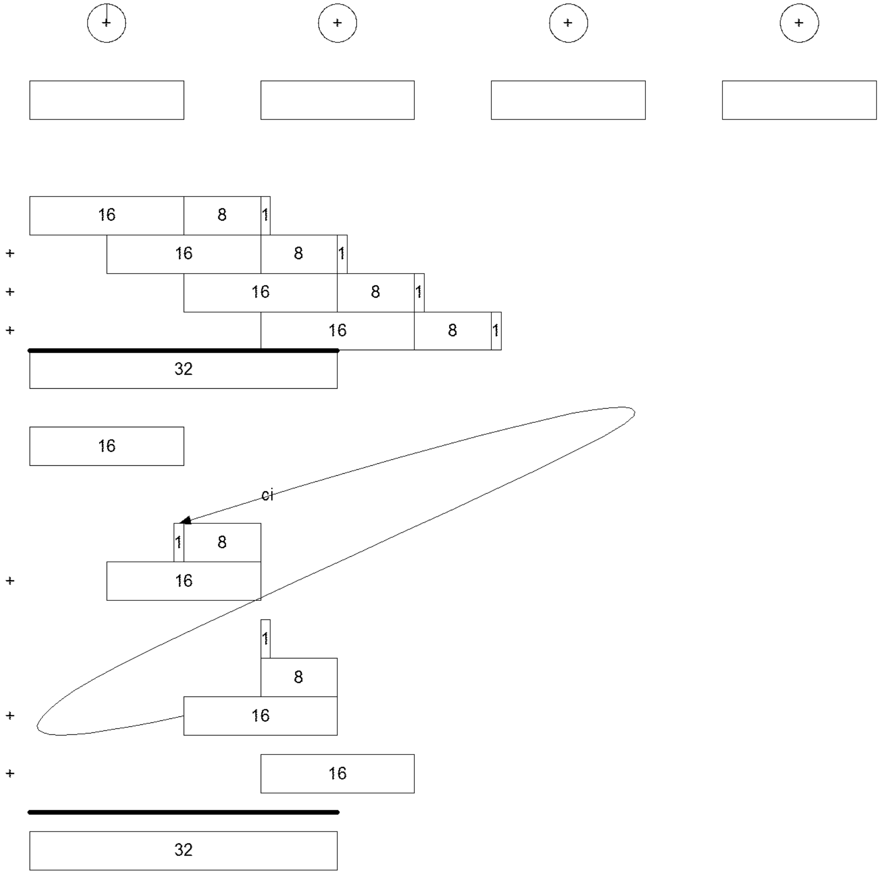


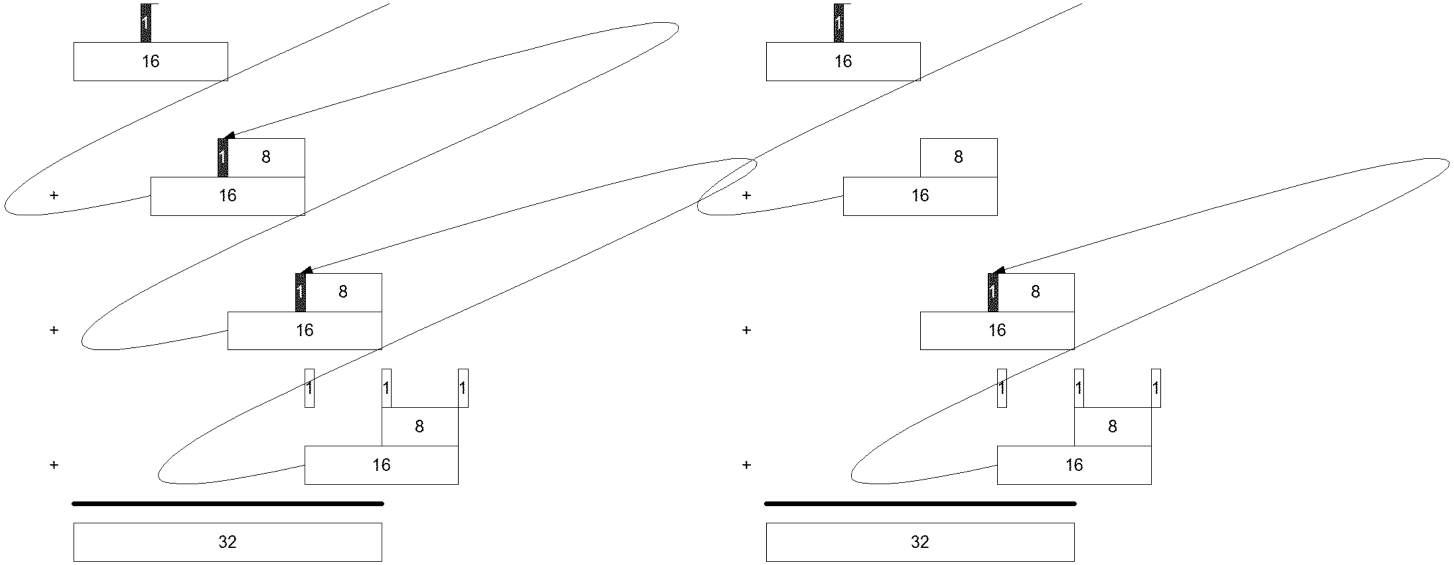
C'[9] = C[8] & B[0]
C'[8] = C[8] ^ B[0]

either mip(5) + 2(4), or mip(5) + ani(4). Those 9b are the spread over the channels to the right. These are split in 8+1, where the 8b goes to the neighbor, but the 1 ends up in the rightmost unit.
- A special 2b adder is used for the last unit in order to avoid 4-term add.

16b HiColor
32b HiColor




+
+
+
+
16
8
1
+
16
8
1
+
16
8
1
+
16
8
1
32
16
ci
1
8
+
16
1
8
+
16
+
16
32


1
16
1
8
+
16
1
8
+
16
1
1
1
8
+
16
32
1
16
8
+
16
1
8
+
16
1
1
1
8
+
16
32

Tom Frisinger
11/13/03
v.0.99q

R400 Floating Point Numerics

Red = open questions/issues, answers/opinions appreciated. Thank you in advance ☺

32-bit Floating Point Format (32-bit FP)

Same as single precision IEEE 754 floating point specification:

```
bit  31      = sign (1-bit)
bits 30..23  = exponent (8-bits, biased 127)
bits 22..0   = mantissa (23-bits)
```

```
normalized 32-bit floating point    = (-1)^sign * 1.mantissa * 2^(exponent + (-bias))
denormalized 32-bit floating point  = (-1)^sign * 0.mantissa * 2^-126
```

NaNs (exponent = 0xFF, mantissa != 0x0) (32-bit FP)

NaNs are supported and comply with IEEE 754

```
QNaNs (quiet)      = bit 22 = 1, bits 21..0 == any
SNaNs (signaling)  = bit 22 = 0, bits 21..0 != 0x0
```

SNaNs and QNaN are referred to collectively as NaNs

NaNs propagate unmodified (except for NaN * 0 = 0 and 0 * NaN = 0).
SNaNs are not converted to QNaNs during arithmetic operations.

Source modifiers (abs/neg) do affect NaNs sign (see note in LoadVector4() routine below).
Clamping/saturation does not affect NaNs.

Clever software is free to do what they choose with the mantissa bits.

The following NaN codes are currently supported:
R400_FP_NAN = 0xFFC00000 -- indefinite floating point operation (fyi, this is a QNaN)

ZEROs (exponent = 0x0, mantissa = 0x0) (32-bit FP)

+ZERO and -ZERO are supported and comply with IEEE 754

```
+ZERO  = 0x00000000 (bit 31 = 0) (aka 0.0f, ZERO)
-ZERO  = 0x80000000 (bit 31 = 1)
```

Non-ZERO Denorm Support (exponent=0x0, mantissa != 0x0) (32-bit FP)

32-bit floating point denorms are not supported on R400 for arithmetic and logical operations.

R400 will use a flush to zero policy for 32-bit floating point denorms. Sign will be preserved:
-denorm → -ZERO
+denorm → +ZERO

IEEE requires denorms to be supported.

Supporting denorms guarantees that for all real numbers the following property (of real numbers) is preserved:

X = Y ↔ X + (-Y) = 0

Flushing denorms to zero does not preserve this relationship. It's important to understand and appreciate this because unexpected results could be produced. For example, two non-equal normalized floating point numbers may be subtracted producing a result that it too small to be represented in the normalized range. Without denorms, such a result will be flushed to zero, which may introduce a zero where one is not expected potentially affecting downstream computation.

The design decision of not supporting 32-bit floating point denorms was based on several facts. First, adding denorm support is not with out cost in both area and schedule. Second, it's believed that normal graphics operations would not take advantage of the 32-bit floating point denorm range.

32-bit floating point denorms are however preserved during move operations.

ONEs (for reference, not special) (32-bit FP)

```
+ONE  = 0x3F800000 (bit 31 = 0) (aka 1.0f, ONE)
-ONE  = 0xBF800000 (bit 31 = 1)
```

MAX/MIN (for reference, not special) (32-bit FP)

```
+MAX_FLOAT  = 0x7F7FFFFF (bit 31 = 0) (aka MAX_FLOAT)
-MAX_FLOAT  = 0xFF7FFFFF (bit 31 = 1)
```

INFINITYs (exponent = 0xFF, mantissa = 0x0) (32-bit FP)

+INFINITY and -INFINITY are supported and comply with IEEE 754

```
+INFINITY  = 0x7F800000 (bit 31 = 0) (aka INFINITY, +INF, INF)
-INFINITY  = 0xFF800000 (bit 31 = 1) (aka -INF)
```

Rounding (32-bit FP)

Rounding is implemented using a round to zero policy only. This is equivalent to truncation. Note this is an IEEE rounding mode, but round to nearest even is the default rounding mode for IEEE.

Overflow (32-bit FP)

Follows round toward zero convention.

> +MAX_FLOAT → +MAX_FLOAT
< -MAX_FLOAT → -MAX_FLOAT

Underflow (32-bit FP)

Follows flush to zero convention.

Smaller than $+2^{126}$ → +ZERO
Smaller than -2^{126} → -ZERO

Notation (32-bit FP)

```
<x>     = all 32-bit floating point numbers including +/-ZERO, +/-INFINITY and NaNs
<xnn>   = all 32-bit floating point numbers including +/-INFINITY and +/-ZERO but excluding NaNs
<xnnz>  = all floating point numbers including +/-INFINITY but excluding NaNs and +/-ZERO

<f>     = all finite 32-bit floating point number including +/-ZERO but excluding +/-INFINITY and NaNs
<fnz>   = all finite 32-bit floating point numbers but excluding +/-INFINITY, NaNs and +/-ZERO
<+f>    = all positive <f> including +ZERO, excluding -ZERO
<+fnz>  = all positive <f> including +ZERO, excluding -ZERO
<-f>    = all negative <f> excluding +/-ZERO
<-fnz>  = all negative <f> excluding +/-ZERO

A is any element in set <xnn>
B is any element in set <xnn>
Af is any element in set <f>
```

Arithmetic Operations (32-bit FP)

Denorms are treated as and obey rules of appropriate ZERO.

```
-(+ZERO) = -ZERO     -- Applies to all rules below.
-(-ZERO) = +ZERO     -- Applies to all rules below.
```

Addition (32-bit FP)

```
NaN   + <xnn>  = NaN
<xnn> + NaN    = NaN
NaN1  + Nan2   = NaN1

(A + B) == (B + A)   -- Applies to all addition rules below.
(A - B) == (A + -B)  -- Applies to all addition rules below.

Af + -Af = +ZERO     -- Applies to all addition rules below.

+INFINITY + <f>        = +INFINITY
-INFINITY + <f>        = -INFINITY
+INFINITY + +INFINITY  = +INFINITY
-INFINITY + -INFINITY  = -INFINITY
+INFINITY + -INFINITY  = R400_FP_NAN

-ZERO + <f>    = <f>
+ZERO + <fnz>  = <fnz>
+ZERO + +ZERO  = +ZERO
```

Multiplication (32-bit FP)

```
NaN    * <xnnz>  = NaN
<xnnz> * NaN     = NaN
NaN1   * Nan2    = NaN1

(A * B) == (B * A)   -- Applies to all multiplication rules below.
```

+/-ZERO * <x> = +ZERO -- This is not IEEE 754 compliant. IEEE rules states +/-ZERO * NaN = NaN, +/-ZERO * +/-INFINITY = NaN and all other multiplication by +/-ZERO obey sign rules for multiplication.

Breaking this IEEE rule was not taken lightly. Legacy 1.x shaders require this behavior so not supporting it in some fashion was not an option. Also, in fixed point texture filtering and alpha blending a weight of zero specifies that the multiplicand should be ignored. This no longer holds true if you follow IEEE. Even worse, following IEEE rules when filtering and alpha blending limits important optimizations because source and destination reads are always required. If fact, we aren't even sure it's ever desirable to follow IEEE when filtering or alpha blending.

While the R400 will still do all texture filtering in fixed point, the R400 has the ability to alpha blend 16-bit floating point surfaces.

Given the above and the desire to make all floating point operations in the R400 consistent, led to us adopting the rule +/-ZERO * <x> = +ZERO. For what it's worth, the R200 and R300 follow this rule too. Had more time been available we might have considered supporting both multiplication rules. We do not anticipate any compatibility issues with this decision. One should perhaps only be cautious of the fact that +/-ZERO * NaN = +ZERO, which will terminate NaN propagation.

```
+INFINITY * <+fnz>      = +INFINITY
+INFINITY * <-fnz>      = -INFINITY
-INFINITY * <+fnz>      = -INFINITY
-INFINITY * <-fnz>      = +INFINITY
+INFINITY * +INFINITY   = +INFINITY
-INFINITY * -INFINITY   = +INFINITY
+INFINITY * -INFINITY   = -INFINITY
```

+ONE * <x> = <x> (except when <x> is -ZERO which always produces +ZERO)

Logical Operations (32-bit FP)

Denorms are treated as and obey rules of appropriate ZERO.

```
A  ==, >=, <=  A is TRUE
A  !=, >, <    A is FALSE

(A == B)  ==  (B == A) is TRUE
(A != B)  ==  (B != A) is TRUE

NaN    !=                    NaN is TRUE
NaN    ==, <, <=, >, >=      NaN is FALSE
NaN    !=                  <xnn> is TRUE
NaN    ==, <, <=, >, >=    <xnn> is FALSE
<xnn>  !=                    NaN is TRUE
<xnn>  ==, <, <=, >, >=      NaN is FALSE

+ZERO  ==, >=, <=  -ZERO is TRUE
+ZERO  !=, >, <    -ZERO is FALSE
-ZERO  ==, >=, <=  +ZERO is TRUE
-ZERO  !=, >, <    +ZERO is FALSE

-INFINITY  ==, >=, <=  -INFINITY is TRUE
-INFINITY  !=, >, <    -INFINITY is FALSE

+INFINITY  ==, >=, <=  +INFINITY is TRUE
+INFINITY  !=, >, <    +INFINITY is FALSE

-INFINITY  <, <=, !=  -MAX_FLOAT is TRUE
-INFINITY  ==, >, >=  -MAX_FLOAT is FALSE

+INFINITY  >, >=, !=  +MAX_FLOAT is TRUE
+INFINITY  ==, <, <=  +MAX_FLOAT is FALSE
```

16-bit Floating Point Format (16-bit FP)

```
bit  15      = sign (1-bit)
bits 14..10  = exponent (5-bits, biased 15)
bits 9..0    = mantissa (10-bits)

normalized floating point    = (-1)^sign * 1.mantissa * 2^(exponent + (-bias))
denormalized floating point  = (-1)^sign * 0.mantissa * 2^-14
```

NaNs (N/A) (16-bit FP)

NaNs do not exist in 16-bit floating point format.

An exponent of 31 encodes numbers a power of 2 larger in magnitude than numbers with an exponent of 30.

ZEROs (exponent = 0x0, mantissa = 0x0) (16-bit FP)

+ZERO_16 and -ZERO_16 are supported.

```
+ZERO_16  = 0x0000 (bit 15 = 0) (aka 0.0f, ZERO_16)
-ZERO_16  = 0x8000 (bit 15 = 1)
```

Non-ZERO Denorm Support (exponent=0x0, mantissa != 0x0) (16-bit FP)

16-bit floating point denorms are supported on R400 for arithmetic and logical operations.

See section on Non-ZERO denorm support in 32-bit floating point section above for notes and implications.

ONEs (for reference, not special) (16-bit FP)

```
+ONE_16  = 0x3C00 (bit 15 = 0) (ONE_16)
-ONE_16  = 0xBC00 (bit 15 = 1)
```

MAX/MIN (for reference, not special) (16-bit FP)

```
+MAX_FLOAT_16  = 0x7FFF (bit 15 = 0) (aka MAX_FLOAT_16)
-MAX_FLOAT_16  = 0xFFFF (bit 15 = 1)
```

INFINITYs (N/A) (16-bit FP)

INFINITYs do not exist in 16-bit floating point format.

An exponent of 31 encodes numbers a power of 2 larger in magnitude than numbers with an exponent of 30.

Rounding (16-bit FP)

Rounding is implemented using a round to nearest (round towards plus infinity) policy only.

Notation (16-bit FP)

```
<f>     = all 16-bit floating point number including +/-ZERO_16
<fnz>   = all 16-bit floating point numbers but excluding +/-ZERO_16
<+f>    = all positive <f> including +ZERO, excluding -ZERO_16
<+fnz>  = all positive <f> including +ZERO, excluding -ZERO_16
<-f>    = all negative <f> excluding +/-ZERO_16
<-fnz>  = all negative <f> excluding +/-ZERO_16
```

A is any element in set <f>
B is any element in set <f>

Arithmetic Operations (16-bit FP)

```
-(+ZERO_16) = -ZERO_16  -- Applies to all rules below.
-(-ZERO_16) = +ZERO_16  -- Applies to all rules below.
```

Addition (16-bit FP)

```
(A + B) == (B + A)      -- Applies to all addition rules below.
(A - B) == (A + -B)     -- Applies to all addition rules below.

A + -A = +ZERO_16       -- Applies to all addition rules below.

-ZERO_16 + <f>       = <f>
+ZERO_16 + <fnz>     = <fnz>
+ZERO_16 + +ZERO_16  = +ZERO_16
```

Multiplication (16-bit FP)

```
(A * B) == (B * A)      -- Applies to all multiplication rules below.

+/-ZERO_16 * <f> = +ZERO_16  -- See 32-bit FP section for justification.

+ONE_16 * <f> = <f> (except when <x> is -ZERO_16 which always produces +ZERO_16)
```

Logical Operations (16-bit FP)

```
A  ==, >=, <=  A is TRUE
A  !=, >, <    A is FALSE

(A == B)  ==  (B == A) is TRUE
(A != B)  ==  (B != A) is TRUE

+ZERO_16  ==, >=, <=  -ZERO_16 is TRUE
+ZERO_16  !=, >, <    -ZERO_16 is FALSE
```

-ZERO_16 ==, >=, <= +ZERO_16 is TRUE
-ZERO_16 !=, >, < +ZERO_16 is FALSE

R400 Conversion Numerics

16-bit / 32-bit Floating Point To Fixed Point Conversions

I believe IEEE specifies that converting to fixed point must obey round mode for finite floating point numbers. Behavior is undefined for +/-INFINITY, NaNs and out of range values under IEEE rules.

When converting to fixed point (any time fixed point hardware has floating point sources, e.g. rasterization and texture filtering) R400 will use a round to nearest policy (basically: (int)(floor(f + 0.5f)) when converting to integers).

In cooperation with Microsoft, R400 will use the following rules when converting 16-bit and 32-bit floating point values to fixed point:

MAX_FIXED = Largest value for given fixed point format
MIN_FIXED = Smallest value for given fixed point format

+INFINITY → MAX_FIXED
-INFINITY → MIN_FIXED

+NaN → MAX_FIXED
-NaN → MIN_FIXED

> MAX_FIXED → MAX_FIXED
< MIN_FIXED → MIN_FIXED

16-bit Floating Point To / From 32-bit Floating Point Conversions

The following table represents the relationship between the 32-bit and 16-bit floating point ranges:

32-bit FP exponent	unbiased exponent		16-bit FP exponent	16-bit FP fraction	Notes
255					
254	127				
...					Values start clamping to +/-MAX_FLOAT_16
127+16	16	Max exponent	31	1.xxxxxxxxxx	
127+15	15		30	1.xxxxxxxxxx	
127	0		15	1.xxxxxxxxxx	
113	-14	Min exponent	1	1.xxxxxxxxxx	
112	-15	Denormalized	0	0.1xxxxxxxxx	
111	-16	Denormalized	0	0.01xxxxxxxx	
110	-17	Denormalized	0	0.001xxxxxxx	
109	-18	Denormalized	0	0.0001xxxxxx	
108	-19	Denormalized	0	0.00001xxxxx	
107	-20	Denormalized	0	0.000001xxxx	
106	-21	Denormalized	0	0.0000001xxx	
105	-22	Denormalized	0	0.00000001xx	
104	-23	Denormalized	0	0.000000001x	
103	-24	Denormalized	0	0.0000000001	
102	-25		0	0.0	Values start clamping to +/-ZERO_16
...					
0			0	0.0	

+ZERO ↔ +ZERO_16
-ZERO ↔ -ZERO_16

When converting from 32-bit floating point to 16-bit floating point the following rules also apply:

Rounding is done using round to nearest.

+INFINITY → +MAX_FLOAT_16
-INFINITY → -MAX_FLOAT_16

+NaN → +MAX_FLOAT_16
-NaN → -MAX_FLOAT_16

> +MAX_FLOAT_16 → +MAX_FLOAT_16
< -MAX_FLOAT_16 → -MAX_FLOAT_16

R400 PA Numerics

The R400 has the ability to kill primitives that have positions containing NANs and +/-INFINITY. Three bits control this functionality. One bit kills if any XYs are NANs or +/-INFINITY, a second bit controls killing if any Zs are NANs or +/-INFINITY and the third bit controls killing the primitive if any Ws are NANs or +/-INFINITY.

The R400 should always be programmed to kill primitives that have any XYZW positions containing NaNs and +/-INFINITY. One should not be tempted to do things like disabling killing on Z when not Z buffering because clip testing and the like may lead to unexpected behavior.

Clipping is done using 32-bit floating point. The clipping HW may not follow the conventions in this document when presented with positions containing NaNs and +/-INFINITY. Clipping of normal finite floating point values do however. For this reason, it is strongly encourage that the chip be always programmed to kill positions containing NaNs and +/-INFINITY.

Clay to provide more detail on what happens when clipping and rasterizing with NaNs and +/-INFINITY. Also what happens with point size?

R400 Interpolator Numerics

All computations are done in 32-bit floating point and follow rules described above.

R400 will do barycentric interpolation. Interpolation equation for a parameter will be: P(i,j,k) = A + (B - A) * j + (C - A) * i where i,j,k represents the barycentric position and i + j + k = 1.0f. Note, the equation used on R400 is derived from P(i,j,k) = (A * k) + (B * j) + (C * i).

R400 Texture Pipe Numerics

Texture filtering computations are done in fixed point.
Texture gradient computations are done in 16-bit floating point.
All conversions follow the given rules.

The texture pipe cannot filter floating point textures and only allows point-sampled fetches into 16-bit (DATA_FORMATs *16_FLOAT) and 32-bit (DATA_FORMATs *32_FLOAT) floating point textures (and vertex arrays).

Fetched 32-bit floating point data is presented to the GPRs unmodified.

Fetched 16-bit floating point data is converted to 32-bit floating point using the above conversion rules before being stored in the GPRs.

Textures with DATA_FORMATs of *16_EXPAND allow 16-bit floating point textures to be filtered by first converting the 16-bit floating point data to a 16.16 signed fixed point (2's complement) representation. The conversion takes place in the texture cache and follows the rules above for converting floating point to fixed point. Note that this fixed point representation has a range of +/-32K while 16-bit floating point has a range of +/-64K so some dynamic range will be lost along with the obvious precision loss.

Gradients set through SetGradientsH and SetGradientsV are converted to and stored as 16-bit floating point.

Values are stored in the texture LOD register as 5.5 signed fixed point (2's complement). The texture LOD register is set through SetTexLOD at which point the 32-bit float pointing to fixed point conversion takes place.

R400 RB Numerics

All conversions from 32-bit floating point coming from shader pipe to the internal format follow conversion rules.

All internal computations (alpha blending) are done in 16-bit floating point as described above. (In fact, the actual format may have addition mantissa bits. Details to come. However, for the purposes of this document it can be thought of as 16-bit floating point.)

Alpha blending is supported for 16-bit floating point render targets.

Given our rules, one issue does arise when alpha blending 16-bit floats. Given the following alpha blending equation:

(+ONE_16 * SRC) + (+ZERO_16 * DST) = OUT_COLOR

One might assume that SRC and OUT_COLOR will have the same bit pattern (also assuming SRC can be represented with out clamping in 16-bit FP). This WAS how we were going to effectively disable alpha blending. However, consider when SRC=-ZERO_16. In this case +ONE_16 * -ZERO_16 = +ZERO_16 and +ZERO_16 * DST always equals +ZERO_16. Resulting in:
+ZERO_16 + +ZERO_16 = +ZERO_16 = OUT_COLOR.

What this means is that the bit pattern of -ZERO_16 cannot be outputted from the RB when alpha blending. Bummer.

To work around this issue, an alpha blend disable bit is provided. When alpha blending is disabled, 16-bit and 32-bit floating point values pass through the alpha blending logic unmodified.

Alpha blending is not supported for 32-bit floating point render targets as is required to be disabled.

Conversion to the render target format follows conversion rules for that format. Additionally, the R400 can be programmed to round to nearest, truncate, or dither when doing the conversion.

R400 Shader Pipe Numerics

All computations are done in 32-bit floating point as described above.

Miscellaneous Notes
WZYX = ABGR respectively. (W and A always in high channel)
In the pseudo code below, all subtractions are done as negated additions.

Execution Machine

```
#define FALSE   0
#define TRUE    1
#define SKIP    0
#define EXECUTE 1

//
// Think of these as globals
//
float   PreviousScalar;
boolean KillPixel;        // 1-bit stored in SQ
boolean PredicateReg;     // 1-bit stored in SQ
int     AddressReg;       // 9-bit signed integer stored in SQ

/* ExecuteShader */
void ExecuteShader(Instructions instructions, unsigned int count)
{
  Vector4 SRC_A, SRC_B, SRC_C;
  Vector4 scalarResult, vectorResult;
  unsigned short swizzlePatternA, swizzlePatternB, swizzlePatternC;
  boolean doAbsA, doNegA, doAbsB;
  boolean doNegB, doAbsC, doNegC;
  unsigned int vectorOpcode, scalarOpcode;
  unsigned byte vectorWriteMask, scalarWriteMask;
  boolean doVectorClamp, doScalarClamp;

  PreviousScalar = UNDEFINED;
  KillPixel      = FALSE;
  PredicateReg   = UNDEFINED;
  AddressReg     = UNDEFINED;

  while(count > 0) {

    currentInst = get_next_instruction(instructions);

    //
    // SRC_A
    //
    SRC_A              = extract_and_load_SRC_A(currentInst);
    SwizzlePatternA    = extract_swizzle_pattern_SRC_A(currentInst);
    doAbsA             = extract_abs_SRC_A(currentInst);
    doNegA             = extract_neg_SRC_A(currentInst);
    SRC_A              = LoadVector4(SRC_A, swizzlePattern, doAbsA, doNegA);

    //
    // SRC_B
    //
    SRC_B              = extract_and_load_SRC_B(currentInst);
    SwizzlePatternB    = extract_swizzle_pattern_SRC_B(currentInst);
    doAbsB             = extract_abs_SRC_B(currentInst);
    doNegB             = extract_neg_SRC_B(currentInst);
    SRC_B              = LoadVectorB(SRC_B, swizzlePattern, doAbsB, doNegB);

    //
    // SRC_C
    //
    SRC_C              = extract_and_load_SRC_C(currentInst);
    SwizzlePatternC    = extract_swizzle_pattern_SRC_C(currentInst);
    doAbsC             = extract_abs_SRC_C(currentInst);
    doNegC             = extract_neg_SRC_C(currentInst);
    SRC_C              = LoadVector4(SRC_C, swizzlePattern, doAbsC, doNegC);
    //
    //  Software should note that there is only one abs modifier bit for constants
    //  and that its state applies to all constants in the instruction.
    //

    //
    // Vector
    //
    vectorOpcode      = extract_vector_opcode(currentInst);
    doVectorClamp     = extract_vector_clamp(currentInst);
    vectorWriteMask   = extract_vector_write_mask(currentInst);

    //
    // Scalar
    //
    scalarOpcode      = extract_scalar_opcode(currentInst);
    doScalarClamp     = extract_scalar_clamp(currentInst);
    scalarWriteMask   = extract_scalar_write_mask(currentInst);

    //
    // Execute Instruction
    //
    ExecuteInstruction(vectorOpcode, scalarOpcode,
                       doVectorClamp, doScalarClamp,
                       SRC_A, SRC_B, SRC_C);

    --count;
  }

  // Done
  return;
}

/* ExecuteInstruction */
void ExecuteInstruction(unsigned short vectorOpcode, unsigned short scalarOpcode,
                        boolean doVectorClamp, boolean doScalarClamp,
                        byte vectorWriteMask, byte scalarWriteMask,
                        Vector4 SRC_A, Vector4 SRC_B, Vector4 SRC_C)
{
    Vector4 vectorResult, scalarResult;
    Vector4 vectorDST, scalarDST;

    //
    // In HW, SQ probably handles most of this
    //

    // Vector instruction

    switch(vectorOpcode) {

    case ADD: // 0x0
      vectorResult = ADD_V(SRC_A, SRC_B);
      break;
    case MUL: // 0x1
      vectorResult = MUL_V(SRC_A, SRC_B);
      break;
    case MAX: // 0x2
      vectorResult = MAX_V(SRC_A, SRC_B);
      break;
    case MIN: // 0x3
      vectorResult = MIN_V(SRC_A, SRC_B);
      break;
    case SETE: // 0x4
      vectorResult = SETE_V(SRC_A, SRC_B);
      break;
    case SETGT: // 0x5
      vectorResult = SETGT_V(SRC_A, SRC_B);
      break;
    case SETGE: // 0x6
      vectorResult = SETGE_V(SRC_A, SRC_B);
      break;
    case SETNE: // 0x7
      vectorResult = SETNE_V(SRC_A, SRC_B);
      break;
```

```
case FRACT: // 0x8
  vectorResult = FRACT_V(SRC_A);
  break;
case TRUNC: // 0x9
  vectorResult = TRUNC_V(SRC_A);
  break;
case FLOOR: // 0xA
  vectorResult = FLOOR_V(SRC_A);
  break;
case MULADD: // 0xB
  // Note 3 source
  vectorResult = MULADD_V(SRC_A, SRC_B, SRC_C);
  break;
case CNDE: // 0xC
  // Note 3 source
  vectorResult = CNDE_V(SRC_A, SRC_B, SRC_C);
  break;
case CNDGE: // 0xD
  // Note 3 source
  vectorResult = CNDGE_V(SRC_A, SRC_B, SRC_C);
  break;
case CNDGT: // 0xE
  // Note 3 source
  vectorResult = CNDGT_V(SRC_A, SRC_B, SRC_C);
  break;
case DOT4: // 0xF
  vectorResult = DOT4_V(SRC_A, SRC_B);
  break;
case DOT3: // 0x10
  vectorResult = DOT3_V(SRC_A, SRC_B);
  break;
case DOT2ADD: // 0x11
  // Note 3 source
  vectorResult = DOT2ADD_V(SRC_A, SRC_B, SRC_C);
  break;
case CUBE: // 0x12
  vectorResult = CUBE_V(SRC_A, SRC_B);
  break;
case MAX4: // 0x13
  vectorResult = MAX4_V(SRC_A);
  break;
case PRED_SETE_PUSH: // 0x14
  vectorResult = PRED_SETE_PUSH_V(SRC_A, SRC_B);
  break;
case PRED_SETNE_PUSH: // 0x15
  vectorResult = PRED_SETNE_PUSH_V(SRC_A, SRC_B);
  break;
case PRED_SETGT_PUSH: // 0x16
  vectorResult = PRED_SETGT_PUSH_V(SRC_A, SRC_B);
  break;
case PRED_SETGE_PUSH: // 0x17
  vectorResult = PRED_SETGE_PUSH_V(SRC_A, SRC_B);
  break;
case KILLE: // 0x18
  vectorResult = KILLE_V(SRC_A, SRC_B);
  break;
case KILLGT: // 0x19
  vectorResult = KILLGT_V(SRC_A, SRC_B);
  break;
case KILLGE: // 0x1A
  vectorResult = KILLGE_V(SRC_A, SRC_B);
  break;
case KILLNE: // 0x1B
  vectorResult = KILLNE_V(SRC_A, SRC_B);
  break;
case DST: // 0x1C
  vectorResult = DST_V(SRC_A, SRC_B);
  break;
case MOVA: // 0x1D
  vectorResult = MOVA_V(SRC_A, SRC_B);
  break;
default:
  break;
}  // switch(vectorOpcode)
```

```
// Scalar instruction
//
// SW should note scalar instruction is always coissued.  This means when using a 3 source
// vector instruction SRC_C is shared.  Typically this won't be useful and SW will want
// to mask scalar pipe in this case.  More details below.

switch(scalarOpcode) {

case ADD: // 0x0
  scalarResult = ADD_S(SRC_C);
  break;
case ADD_PREV: // 0x1
  scalarResult = ADD_PREV_S(SRC_C);
  break;
case MUL: // 0x2
  scalarResult = MUL_S(SRC_C);
  break;
case MUL_PREV: // 0x3
  scalarResult = MUL_PREV_S(SRC_C);
  break;
case MUL_PREV2: // 0x4
  scalarResult = MUL_PREV2_S(SRC_C);
  break;
case MAX: // 0x5
  scalarResult = MAX_S(SRC_C);
  break;
case MIN: // 0x6
  scalarResult = MIN_S(SRC_C);
  break;
case SETE: // 0x7
  scalarResult = SETE_S(SRC_C);
  break;
case SETGE: // 0x8
  scalarResult = SETGE_S(SRC_C);
  break;
case SETGT: // 0x9
  scalarResult = SETGT_S(SRC_C);
  break;
case SETNE: // 0xA
  scalarResult = SETNE_S(SRC_C);
  break;
case FRACT: // 0xB
  scalarResult = FRACT_S(SRC_C);
  break;
case TRUNC: // 0xC
  scalarResult = TRUNC_S(SRC_C);
  break;
case FLOOR: // 0xD
  scalarResult = FLOOR_S(SRC_C);
  break;
case EXP_IEEE: // 0xE
  scalarResult = EXP_IEEE_S(SRC_C);
  break;
case LOG_CLAMPED: // 0xF
  scalarResult = LOG_CLAMPED_S(SRC_C);
  break;
case LOG_IEEE: // 0x10
  scalarResult = LOG_IEEE_S(SRC_C);
  break;
case RECIP_CLAMPED: // 0x11
  scalarResult = RECIP_CLAMPED_S(SRC_C);
  break;
case RECIP_FF: // 0x12
  scalarResult = RECIP_FF_S(SRC_C);
  break;
case RECIP_IEEE: // 0x13
  scalarResult = RECIP_IEEE_S(SRC_C);
  break;
case RECIPSQRT_CLAMPED: // 0x14
  scalarResult = RECIPSQRT_CLAMPED_S(SRC_C);
  break;
case RECIPSQRT_FF: // 0x15
  scalarResult = RECIPSQRT_FF_S(SRC_C);
  break;
case RECIPSQRT_IEEE: // 0x16
  scalarResult = RECIPSQRT_IEEE_S(SRC_C);
  break;
case MOVA: // 0x17
  scalarResult = MOVA_S(SRC_C);
  break;
case MOVA_FLOOR: // 0x18
  scalarResult = MOVA_FLOOR_S(SRC_C);
  break;
case SUB: // 0x19
  scalarResult = SUB_S(SRC_C);
  break;
case SUB_PREV: // 0x1A
  scalarResult = SUB_PREV_S(SRC_C);
  break;
case PRED_SETE: // 0x1B
  scalarResult = PRED_SETE_S(SRC_C);
  break;
case PRED_SETNE: // 0x1C
  scalarResult = PRED_SETNE_S(SRC_C);
  break;
case PRED_SETGT: // 0x1D
  scalarResult = PRED_SETGT_S(SRC_C);
  break;
case PRED_SETGE: // 0x1E
  scalarResult = PRED_SETGE_S(SRC_C);
  break;
case PRED_SET_INV: // 0x1F
  scalarResult = PRED_SET_INV_S(SRC_C);
  break;
case PRED_SET_POP: // 0x20
  scalarResult = PRED_SET_POP_S(SRC_C);
  break;
case PRED_SET_CLR: // 0x21
  scalarResult = PRED_SET_CLR_S(SRC_C);
  break;
case PRED_SET_RESTORE: // 0x22
  scalarResult = PRED_SET_RESTORE_S(SRC_C);
  break;
case KILLE: // 0x23
  scalarResult = KILLE_S(SRC_C);
  break;
case KILLGT: // 0x24
  scalarResult = KILLGT_S(SRC_C);
  break;
case KILLGE: // 0x25
  scalarResult = KILLGE_S(SRC_C);
  break;
case KILLNE: // 0x26
  scalarResult = KILLNE_S(SRC_C);
  break;
case KILLONE: // 0x27
  scalarResult = KILLONE_S(SRC_C);
  break;
case SQRT_IEEE: // 0x28
  scalarResult = SQRT_IEEE_S(SRC_C);
  break;
case MUL_CONST_0: // 0x2a
  // TBD
  break;
case MUL_CONST_1: // 0x2b
  // TBD
  break;
case ADD_CONST_0: // 0x2c
  // TBD
  break;
case ADD_CONST_1: // 0x2d
  // TBD
  break;
case SUB_CONST_0: // 0x2e
  // TBD
  break;
case SUB_CONST_1: // 0x2f
```

```
  // TBD
  break;
case SIN: // 0x30
  scalarResult = SIN_S(SRC_C);
  break;
case COS: // 0x31
  scalarResult = COS_S(SRC_C);
  break;
default:
  break;
}  // switch(scalarOpcode)

// Clamping
//
// Clamping done here so I can call instructions from instructions, HW would likely
// do it in the shader instruction code, but the results should be the same.
//
if (doVectorClamp) {
  vectorResult = ClampVector4(vectorResult);
}

if (doScalarClamp) {
  scalarResult = ClampVector4(scalarResult);
}

//
// SW should note that PreviousScalar does not get masked but does get clamped.
//
PreviousScalar = scalarResult.w;

//
// Destination masking/writing, for now not dealing with exports, just GPRs
//

// Vector Mask/Output
//
// Note vector pipe update happens first.  This is only important when
// scalar and vector destinations are the same GPR.

vectorDST = read_reg(/*stuff*/);

if (vectorWriteMask & 8) {
  vectorDST.w = vectorResult.w;
}
if (vectorWriteMask & 4) {
  vectorDST.z = vectorResult.z;
}
if (vectorWriteMask & 2) {
  vectorDST.y = vectorResult.y;
}
if (vectorWriteMask & 1) {
  vectorDST.x = vectorResult.x;
}

write_reg(vectorDST, /*stuff*/);

// Scalar Mask/Output

scalarDST = read_reg(/*stuff*/);

if (scalarWriteMask & 8) {
  scalarDST.w = scalarResult.w;
}
if (scalarWriteMask & 4) {
  scalarDST.z = scalarResult.z;
}
if (scalarWriteMask & 2) {
  scalarDST.y = scalarResult.y;
}
if (scalarWriteMask & 1) {
  scalarDST.x = scalarResult.x;
}

write_reg(scalarDST, /*stuff*/);
```

```
    //
    // Finally handle exporting AddressReg, PredicateReg and PixelKill to SQ.
    // Predicate and kill instructions should not be coissued together in any
    // fashion because they share an update path. Doing so will result in undefined
    // behavior, but not a chip hang.  Mova should not be coissue with another mova.  Doing
    // so will result in undefined behavior, but not a chip hang.  Mova may be coissued with
    // predicate or kill instructions.
    //
    // Software should also be aware that while the results of PRED*, MOVA*, KILL* instructions
    // obey mask rules for writing into the GPRs, results sent to the SQ are never masked.  In
    // general, this means that these instructions should never be used as "filler" instructions
    // for the appropriate pipe when not coissueing.
    //

    //
    // Actual export of PredicateReg, AddressReg and KillPixel done from instructions
    // in this implementation
    //

    // Done
    return;
}

/* ExportPredicateReg */
void ExportPredicateReg(boolean action)
{
  // Called from predicate instructions (PRED_*)

  if (action == SKIP) {
    PredicateReg = SKIP;
  } else {
    PredicateReg = EXECUTE;
  }

  return;
}

/* ExportKillPixel */
boolean ExportKillPixel(Vector4 result)
{
  Boolean doKill = FALSE;

  // Test for pixel kill

  if (result.w == 1.0f) {
    doKill = TRUE;
  }
  if (result.z == 1.0f) {
    doKill = TRUE;
  }
  if (result.y == 1.0f) {
    doKill = TRUE;
  }
  if (result.x == 1.0f) {
    doKill = TRUE;
  }

  return doKill;
}

/* ExportAddressReg */
int ExportAddressReg(float movaVal)
{
  float scalar_result;
  int addressReg;  // 9-bit signed integer

  // movaVal is an "integer" float, NaN or +/-INF at this point

  //
  // Valid address range is [-255..255]
  // Since we are using a 9-bit signed integer rep, use -256 for out-of-range which will insure
  // that when addressReg+constantOffset < 0 causing absolute constant 0 to be used.
  //
  //
  // Clamp Range
  // Basically: scalar_result = MAX(movaVal, -256.0f) followed by
  //            scalar_result = CNDGT((result + -255.0f), -256.0f, scalar_result);
  //

  // Done this way to send NaNs to -256.0f
  if (movaVal >= -256.0f) {
    scalar_result = movaVal.
  } else {
    scalar_result = -256.0f;
  }

  if (scalar_result > 255.0ff) {
    scalar_result = -256.0f;
  } else {
    scalar_result = scalar_result;
  }

  // scalar_result is now within range [-256..255]

  addressReg = truncate_to_int(scalar_result);  // just truncate to 9-bit signed integer at this point

  return addressReg;

  // Special Considerations (per component)
  // 1.) ExportAddressReg(+/-ZERO)    -> 0     (*)
  // 2.) ExportAddressReg(+NaN)       -> -256  (*)
  // 3.) ExportAddressReg(-NaN)       -> -256  (*)
  // 4.) ExportAddressReg(+INFINITY)  -> -256  (*)
  // 5.) ExportAddressReg(-INFINITY)  -> -256  (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* abs */
Vector4 abs(Vector4 src)
{
  Vector4 result;

  // Absolute value input modifier
  //
  // Note that while the "sign" of a NaN may change, it remains a NaN at the end of the day.

  result.w = src.w & 0x7FFFFFFF;
  result.z = src.z & 0x7FFFFFFF;
  result.y = src.y & 0x7FFFFFFF;
  result.x = src.x & 0x7FFFFFFF;

  return result;

  // Special Considerations (per component)
  // 1.) abs(+/-NaN)        = +NaN        (*)
  // 2.) abs(+/-ZERO)       = +ZERO       (*)
  // 3.) abs(+/-denorm)     = +denorm     (*)
  // 4.) abs(+/-INFINITY)   = +INFINITY   (*)
  // 5.) abs(+/-MAX_FLOAT)  = +MAX_FLOAT  (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* neg */
Vector4 neg(Vector4 src)
{
  Vector4 result;

  // Negate input modifier
  //
  // Note that while the "sign" of a NaN may change, it remains a NaN at the end of the day.

  result.w = src.w ^ 0x80000000;
  result.z = src.z ^ 0x80000000;
  result.y = src.y ^ 0x80000000;
  result.x = src.x ^ 0x80000000;

  return result;

  // Special Considerations (per component)
  // 1.)  neg(+NaN)        = -NaN        (*)
  // 2.)  neg(-NaN)        = +NaN        (*)
  // 3.)  neg(-ZERO)       = +ZERO       (*)
  // 4.)  neg(+ZERO)       = -ZERO       (*)
  // 5.)  neg(+denorm)     = -denorm     (*)
  // 6.)  neg(-denorm)     = +denorm     (*)
  // 7.)  neg(+INFINITY)   = -INFINITY   (*)
  // 8.)  neg(-INFINITY)   = +INFINITY   (*)
  // 9.)  neg(+MAX_FLOAT)  = -MAX_FLOAT  (*)
  // 10.) neg(-MAX_FLOAT)  = +MAX_FLOAT  (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* LoadVector4 */
Vector4 LoadVector4(Vector4 src, unsigned short swizzlePattern, boolean doAbs, boolean doNeg)
{
  Vector4 result;
  boolean doAbs, doNeg;

  //
  // Apply source swizzle
  //
  result = do_swizzle(src, swizzlePattern);

  //
  // Apply source modifiers
  //
  // Absolute value (abs) and Negate (neg) are defined to only modify sign bit.  No special rules
  // apply to +/-ZERO, NaNs, denorms or +/-INFINITY.  Note that while the "sign" of a NaN may change,
  // it remains a NaN at the end of the day.
  //

  if (doAbs) {
    // ABS - absolute value
    result = abs(result);
  }

  if (doNeg) {
    // NEG - negate value
    result = neg(result);
  }

  return result;
}

/* ClampVector4 */
Vector4 ClampVector4(Vector4 src)
{
  Vector4 low_bound  = { 0.0f, 0.0f, 0.0f, 0.0f };
  Vector4 high_bound = { 1.0f, 1.0f, 1.0f, 1.0f };
  Vector4 result;
  boolean dstClampEnabled;

  result.w = src.w;
  result.z = src.z;
  result.y = src.y;
  result.x = src.x;

  // Apply destination clamping (saturation)

  //
  // Clamp range [+0.0f, 1.0f] - Needs to be done like this so NaNs are preserved
  //
  result = MAX_V(low_bound, result);
  result = MIN_V(high_bound, result);
  return result;  // destination masking needs to be coded elsewhere

  // Special Considerations (per component)
  // 1.) ClampVector4(+/-ZERO)    = +ZERO  (*)
  // 2.) ClampVector4(+/-denorm)  = +ZERO  (*)
  // 3.) ClampVector4(+INFINITY)  = +ONE   (*)
  // 4.) ClampVector4(-INFINITY)  = +ZERO  (*)
  // 5.) ClampVector4(NaN)        = NaN    (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}
```

Shader Pipe Helper Functions

```
/* Normalize */
Vector4 Normalize(Vector4 src)
{
  Vector4 result;

  result.w = src.w;
  result.z = src.z;
  result.y = src.y;
  result.x = src.x;

  // Flush denorms to +/-ZERO
  // Note, -denorm -> -ZERO and +denorm -> +ZERO (+/-ZERO remain unchanged)
  // I like to call this demoralized floating point ☺

  // Check for zero exponents
  if ((result.w & 0x7F800000) == 0) {
    result.w &= 0x80000000;
  }

  if ((result.z & 0x7F800000) == 0) {
    result.z &= 0x80000000;
  }

  if ((result.y & 0x7F800000) == 0) {
    result.y &= 0x80000000;
  }

  if ((result.x & 0x7F800000) == 0) {
    result.x &= 0x80000000;
  }

  return result;

  // Special Considerations (per component)
  // 1.) Normalize(+ZERO)    = +ZERO  (*)
  // 2.) Normalize(-ZERO)    = -ZERO  (*)
  // 3.) Normalize(+denorm)  = +ZERO  (*)
  // 4.) Normalize(-denorm)  = -ZERO  (*)
  // 5.) Normalize(NaN)      = NaN    (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}
```

Vector Instructions

```
/* ADD_V - vector pipe */
Vector4 ADD_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result;

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  result.w = input1.w + input2.w;
  result.z = input1.z + input2.z;
  result.y = input1.y + input2.y;
  result.x = input1.x + input2.x;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* MUL_V - vector pipe */
Vector4 MUL_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result;

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  result.w = input1.w * input2.w;
  result.z = input1.z * input2.z;
  result.y = input1.y * input2.y;
  result.x = input1.x * input2.x;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* MULADD_V - vector pipe */
Vector4 MULADD_V(Vector4 SRC_A, Vector4 SRC_B, Vector4 SRC_C)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 input3 = SRC_C;
  Vector4 result;

  input1 = Normalize(input1);
  input2 = Normalize(input2);
  input3 = Normalize(input3);

  result.w = (input1.w * input2.w) + input3.w;
  result.z = (input1.z * input2.z) + input3.z;
  result.y = (input1.y * input2.y) + input3.y;
  result.x = (input1.x * input2.x) + input3.x;

  return result;

  // Special Considerations (per component)
  // 1.) MULADD_V(SRC_A, SRC_B, SRC_C) == ADD_V(MUL_V(SRC_A, SRC_B), SRC_C) is guaranteed for all
  //     cases expect where the multiply goes out of range and the add brings it back within range
}

/* MAX_V - vector pipe */
Vector4 MAX_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result, sub_result;

  Vector4 input1norm = Normalize(input1);
  Vector4 input2norm = Normalize(input2);

  result.w = input2.w;
  result.z = input2.z;
  result.y = input2.y;
  result.x = input2.x;

  sub_result.w = input1norm.w + -input2norm.w;
  sub_result.z = input1norm.z + -input2norm.z;
  sub_result.x = input1norm.x + -input2norm.x;
  sub_result.y = input1norm.y + -input2norm.y;
  if (sub_result.w >= 0.0f) {
    result.w = input1.w;
  }
  if (sub_result.z >= 0.0f) {
    result.z = input1.z;
  }
  if (sub_result.y >= 0.0f) {
    result.y = input1.y;
  }
  if (sub_result.x >= 0.0f) {
    result.x = input1.x;
  }

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* MIN_V - vector pipe */
Vector4 MIN_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result, sub_result;

  input1norm = Normalize(input1);
  input2norm = Normalize(input2);

  result.w = input2.w;
  result.z = input2.z;
  result.y = input2.y;
  result.x = input2.x;

  sub_result.w = input1norm.w + -input2norm.w;
  sub_result.z = input1norm.z + -input2norm.z;
  sub_result.y = input1norm.y + -input2norm.y;
  sub_result.x = input1norm.x + -input2norm.x;

  if (sub_result.w < 0.0f) {
    result.w = input1.w;
  }
  if (sub_result.z < 0.0f) {
    result.z = input1.z;
  }
  if (sub_result.y < 0.0f) {
    result.y = input1.y;
  }
  if (sub_result.x < 0.0f) {
    result.x = input1.x;
  }

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* SETE_V - vector pipe */
Vector4 SETE_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result, sub_result;

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  sub_result.w = input1.w + -input2.w;
  sub_result.z = input1.z + -input2.z;
  sub_result.y = input1.y + -input2.y;
  sub_result.x = input1.x + -input2.x;

  result.w = (sub_result.w == 0.0f) ? 1.0f : 0.0f;
  result.z = (sub_result.z == 0.0f) ? 1.0f : 0.0f;
  result.y = (sub_result.y == 0.0f) ? 1.0f : 0.0f;
  result.x = (sub_result.x == 0.0f) ? 1.0f : 0.0f;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* SETGT_V - vector pipe */
Vector4 SETGT_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result, sub_result;

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  sub_result.w = input1.w + -input2.w;
  sub_result.z = input1.z + -input2.z;
  sub_result.y = input1.y + -input2.y;
  sub_result.x = input1.x + -input2.x;

  result.w = (sub_result.w > 0.0f) ? 1.0f : 0.0f;
  result.z = (sub_result.z > 0.0f) ? 1.0f : 0.0f;
  result.y = (sub_result.y > 0.0f) ? 1.0f : 0.0f;
  result.x = (sub_result.x > 0.0f) ? 1.0f : 0.0f;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* SETGE_V - vector pipe */
Vector4 SETGE_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result, sub_result;

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  sub_result.w = input1.w + -input2.w;
  sub_result.z = input1.z + -input2.z;
  sub_result.y = input1.y + -input2.y;
  sub_result.x = input1.x + -input2.x;

  result.w = (sub_result.w >= 0.0f) ? 1.0f : 0.0f;
  result.z = (sub_result.z >= 0.0f) ? 1.0f : 0.0f;
  result.y = (sub_result.y >= 0.0f) ? 1.0f : 0.0f;
  result.x = (sub_result.x >= 0.0f) ? 1.0f : 0.0f;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* SETNE_V - vector pipe */
Vector4 SETNE_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result, sub_result;

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  sub_result.w = input1.w + -input2.w;
  sub_result.z = input1.z + -input2.z;
  sub_result.y = input1.y + -input2.y;
  sub_result.x = input1.x + -input2.x;

  result.w = (sub_result.w != 0.0f) ? 1.0f : 0.0f;
  result.z = (sub_result.z != 0.0f) ? 1.0f : 0.0f;
  result.y = (sub_result.y != 0.0f) ? 1.0f : 0.0f;
  result.x = (sub_result.x != 0.0f) ? 1.0f : 0.0f;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* TRUNC_V - vector pipe */
Vector4 TRUNC_V(Vector4 SRC_A)
{
  Vector4 input1 = SRC_A;
  Vector4 result;

  input1 = Normalize(input1);

  //
  // trunc() returns integer part of floating point. Sign is preserved. So:
  // trunc(3.0f) = 3.0f
  // trunc(3.4f) = 3.0f
  // trunc(3.5f) = 3.0f
  // trunc(3.6f) = 3.0f
  // trunc(-3.0f) = -3.0f
  // trunc(-3.4f) = -3.0f
  // trunc(-3.5f) = -3.0f
  // trunc(-3.6f) = -3.0f
  //

  result.w = trunc(input1.w);
  result.z = trunc(input1.z);
  result.y = trunc(input1.y);
  result.x = trunc(input1.x);

  return result;

  // Special Considerations (per component)
  // 1.) TRUNC_V(NaN)        = NaN
  // 2.) TRUNC_V(+INFINITY)  = +INFINITY
  // 3.) TRUNC_V(-INFINITY)  = -INFINITY
  // 4.) TRUNC_V(+ZERO)      = +ZERO
  // 5.) TRUNC_V(-ZERO)      = -ZERO
  // 6.) TRUNC_V(+denorm)    = +ZERO (*)
  // 7.) TRUNC_V(-denorm)    = -ZERO (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* FLOOR_V - vector pipe */
Vector4 FLOOR_V(Vector4 SRC_A)
{
  Vector4 input1 = SRC_A;
  Vector4 result;

  input1 = Normalize(input1);  // This will also happen in TRUNC_V, but it doesn’t hurt

  result = TRUNC_V(input1);

  if ( (input1.w < 0.0f) && (input1.w != result.w) )
    result.w += -1.0f;
  if ( (input1.z < 0.0f) && (input1.z != result.z) )
    result.z += -1.0f;
  if ( (input1.y < 0.0f) && (input1.y != result.y) )
    result.y += -1.0f;
  if ( (input1.x < 0.0f) && (input1.x != result.x) )
    result.x += -1.0f;

  return result;
  // Special Considerations (per component)
  // 1.) See TRUNC_V
  // 2.) FLOOR_V(NaN)        = NaN        (*)
  // 3.) FLOOR_V(+INFINITY)  = +INFINITY  (*)
  // 4.) FLOOR_V(-INFINITY)  = -INFINITY  (*)
  // 5.) FLOOR_V(+ZERO)      = +ZERO      (*)
  // 6.) FLOOR_V(-ZERO)      = -ZERO      (*)
  // 7.) FLOOR_V(+denorm)    = +ZERO      (*)
  // 8.) FLOOR_V(-denorm)    = -ZERO      (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* FRACT_V - vector pipe */
Vector4 FRACT_V(Vector4 SRC_A)
{
  Vector4 input1 = SRC_A;
  Vector4 result;

  input1 = Normalize(input1);  // This will also happen in FLOOR_V, but it doesn't hurt

  result = FLOOR_V(input1);

  result.w = input1.w + -result.w;
  result.z = input1.z + -result.z;
  result.y = input1.y + -result.y;
  result.x = input1.x + -result.x;

  return result;

  // Special Considerations (per component)
  // 1.) See FLOOR_V
  // 2.) FRACT_V(NaN)          = NaN          (*)
  // 3.) FRACT_V(+/-INFINITY)  = R400_FP_NAN  (*)
  // 4.) FRACT_V(+/-ZERO)      = +ZERO        (*)
  // 5.) FRACT_V(+/-denorm)    = +ZERO        (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* CNDE_V - vector pipe */
Vector4 CNDE_V(Vector4 SRC_A, Vector4 SRC_B, Vector4 SRC_C)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 input3 = SRC_C;
  Vector4 result;

  input1 = Normalize(input1);

  result.w = (input1.w == 0.0f) ? input2 : input3;
  result.z = (input1.z == 0.0f) ? input2 : input3;
  result.y = (input1.y == 0.0f) ? input2 : input3;
  result.x = (input1.x == 0.0f) ? input2 : input3;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* CNDGE_V - vector pipe */
Vector4 CNDGE_V(Vector4 SRC_A, Vector4 SRC_B, Vector4 SRC_C)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 input3 = SRC_C;
  Vector4 result;

  input1 = Normalize(input1);
  result.w = (input1.w >= 0.0f) ? input2 : input3;
  result.z = (input1.z >= 0.0f) ? input2 : input3;
  result.y = (input1.y >= 0.0f) ? input2 : input3;
  result.x = (input1.x >= 0.0f) ? input2 : input3;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* CNDGT_V - vector pipe */
Vector4 CNDGT_V(Vector4 SRC_A, Vector4 SRC_B, Vector4 SRC_C)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 input3 = LoadVector4(SRC_C);
  Vector4 result;

  input1 = Normalize(input1);

  result.w = (input1.w > 0.0f) ? input2 : input3;
  result.z = (input1.z > 0.0f) ? input2 : input3;
  result.y = (input1.y > 0.0f) ? input2 : input3;
  result.x = (input1.x > 0.0f) ? input2 : input3;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* DOT4_V - vector pipe */
Vector4 DOT4_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result;
  float   scalar_result;

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  scalar_result = (input1.w * input2.w) +
                  (input1.z * input2.z) +
                  (input1.y * input2.y) +
                  (input1.x * input2.x);

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* DOT3_V - vector pipe */
Vector4 DOT3_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result;
  float   scalar_result;

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  scalar_result = (input1.z * input2.z) +
                  (input1.y * input2.y) +
                  (input1.x * input2.x);
  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* DOT2ADD_V - vector pipe */
Vector4 DOT2ADD_V(Vector4 SRC_A, Vector4 SRC_B, Vector SRC_C)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 input3 = SRC_C;
  Vector4 result;
  float   scalar_result;

  input1 = Normalize(input1);
  input2 = Normalize(input2);
  input3 = Normalize(input2);

  scalar_result = (input1.y * input2.y) +
                  (input1.x * input2.x) +
                   input3.x;

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* MAX4_V - vector pipe */
Vector4 MAX4_V(Vector4 SRC_A)
{
  Vector4 input1 = SRC_A;
  Vector4 result;

  // Note order is important

  result = MAX_V(input1.xxxx, MAX_V(input1.yyyy, MAX_V(input1.zzzz, input1.wwww)));

  return result;

  // Special Considerations (per component)
  // 1.) See MAX_V rules
}

/* CUBE_V - vector pipe */
Vector4 CUBE_V(Vector4 SRC_A, Vector4 SRC_B)
{
  // Note:
  //   For ease of hardware implementation the cube instruction takes two operands.
  //   Assuming for a given input register Rn: x channel contains the s coordinate, y channel contains
  //   the t coordinate and z channel contains the r coordinate of cube face vector, then in input
  //   operands to the cube instruction should be setup/swizzled as follows:
  //   SRC_A = Rn.zzxy
  //   SRC_B = Rn.yxzz

  // The swizzling below puts the coordinates back into the canonical str form to show the cube
  // algorithm in a sane fashion for software.  The actual implementation of the algorithm in hardware
  // is somewhat different but will produce the same results as the algorithm below.  While the code
  // below only uses some components of SRC_A all components of SRC_A and SRC_B are used by the HW and
  // must contain the appropriate data as described above.

  Vector4 input1 = SRC_A.zwxy;  // put in strr form
  Vector4 input2 = SRC_B.yxzw;  // put in strr form
```

```
float ma, face_id, sc, tc;
Vector4 result;

VECTOR4 input1norm = Normalize(input1);
VECTOR4 input1norm_abs = abs(input1norm);

// similar to MAX
if (input1norm_abs.z >= input1norm_abs.y) {
  // z wins, now compare against x

  if (input1norm_abs.z >= input1norm_abs.x) {
    // z wins
    ma = input1.z;

    if (input1norm.z < 0) {
      // neg z
      face_id = 5.0f;  // NEG_Z
      sc = -input1.x;
      tc = -input1.y;
    } else {
      // pos z
      face_id = 4.0f;  // POS_Z
      sc = input1.x;
      tc = -input1.y;
    }
  } else {
    // x wins
    ma = input1.x;

    if (input1norm.x < 0) {
      // neg x
      face_id = 1.0f;  // NEG_X
      sc = input1.z;
      tc = -input1.y;
    } else {
      // pos x
      face_id = 0.0f;  // POS_X
      sc = -input1.z;
      tc = -input1.y;
    }
  }
} else {
  // y wins

  if (input1norm_abs.y >= input1norm_abs.x) {
    // y wins, now compare against x
    ma = input1.y;

    if (input1norm.y < 0) {
      // neg y
      face_id = 3.0f;  // NEG_Y
      sc = input1.x;
      tc = -input1.z;
    } else {
      // pos y
      face_id = 2.0f;  // POS_Y
      sc = input1.x;
      tc = input1.z;
    }
  } else {
    // x wins
    ma = input1.x;

    if (input1norm.x < 0) {
      // neg x
      face_id = 1.0f;  // NEG_X
      sc = input1.z;
      tc = -input1.y;
    } else {
      // pos x
      face_id = 0.0f;  // POS_X
      sc = -input1.z;
      tc = -input1.y;
    }
    }
  }

  ma *= 2.0f;

  result.w = face_id;
  result.z = ma;
  result.y = sc;
  result.x = tc;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* PRED_SETE_PUSH_V - vector pipe */
Vector4 PRED_SETE_PUSH_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result;
  float   scalar_result;

  //
  // Vector Predicate instructions designed for:
  //  Predicate counter in SRC_A.w
  //  Condition in SRC_B.w
  //  Return result is updated predicate counter
  //

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  if ( (input2.w == 0.0f) && (input1.w == 0.0f) ) {
    ExportPredicateReg(EXECUTE);
    scalar_result = 0.0f;
  } else {
    ExportPredicateReg(SKIP);
    scalar_result = input1.w + 1.0f
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) PRED_SETE_PUSH_V(NaN, <xnn>)  = NaN, evaluates to SKIP           (*)
  // 2.) PRED_SETE_PUSH_V(<xnn>, NaN)  = valid result, evaluates to SKIP  (*)
  // 3.) PRED_SETE_PUSH_V(NaN1, NaN2)  = NaN1, evaluates to SKIP          (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* PRED_SETNE_PUSH_V - vector pipe */
Vector4 PRED_SETNE_PUSH_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result;
  float   scalar_result;

  //
  // Vector Predicate instructions designed for:
  //  Predicate counter in SRC_A.w
  //  Condition in SRC_B.w
  //  Return result is updated predicate counter
  //

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  if ( (input2.w != 0.0f) && (input1.w == 0.0f) ) {
    ExportPredicateReg(EXECUTE);
    scalar_result = 0.0f;
  } else {
    ExportPredicateReg(SKIP);
    scalar_result = input1.w + 1.0f
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) PRED_SETNE_PUSH_V(NaN, <xnn>)  = NaN, evaluates to SKIP           (*)
  // 2.) PRED_SETNE_PUSH_V(<xnn>, NaN)  = Depends on <xnn>                 (*)
  // 3.) PRED_SETNE_PUSH_V(NaN1, NaN2) = NaN1, evaluates to SKIP          (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* PRED_SETGT_PUSH_V - vector pipe */
Vector4 PRED_SETGT_PUSH_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result;
  float   scalar_result;

  //
  // Vector Predicate instructions designed for:
  //  Predicate counter in SRC_A.w
  //  Condition in SRC_B.w
  //  Return result is updated predicate counter
  //

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  if ( (input2.w > 0.0f) && (input1.w == 0.0f) ) {
    ExportPredicateReg(EXECUTE);
    scalar_result = 0.0f;
  } else {
    ExportPredicateReg(SKIP);
    scalar_result = input1.w + 1.0f
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) PRED_SETGT_PUSH_V(NaN, <xnn>)  = NaN, evaluates to SKIP           (*)
  // 2.) PRED_SETGT_PUSH_V(<xnn>, NaN)  = valid result, evaluates to SKIP  (*)
  // 3.) PRED_SETGT_PUSH_V(NaN1, NaN2) = NaN1, evaluates to SKIP          (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* PRED_SETGE_PUSH_V - vector pipe */
Vector4 PRED_SETGE_PUSH_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result;
  float   scalar_result;

  //
  // Vector Predicate instructions designed for:
  //  Predicate counter in SRC_A.w
  //  Condition in SRC_B.w
  //  Return result is updated predicate counter
  //

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  if ( (input2.w >= 0.0f) && (input1.w == 0.0f) ) {
    ExportPredicateReg(EXECUTE);
    scalar_result = 0.0f;
  } else {
    ExportPredicateReg(SKIP);
    scalar_result = input1.w + 1.0f
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) PRED_SETGE_PUSH_V(NaN, <xnn>)  = NaN, evaluates to SKIP           (*)
  // 2.) PRED_SETGE_PUSH_V(<xnn>, NaN)  = valid result, evaluates to SKIP  (*)
  // 3.) PRED_SETGE_PUSH_V(NaN1, NaN2)  = NaN1, evaluates to SKIP          (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* KILLE_V - vector pipe */
Vector4 KILLE_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result, sub_result;

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  // Basically works same as SETE_V
  sub_result.w = input1.w + -input2.w;
  sub_result.z = input1.z + -input2.z;
  sub_result.y = input1.y + -input2.y;
  sub_result.x = input1.x + -input2.x;

  result.w = (sub_result.w == 0.0f) ? 1.0f : 0.0f;
  result.z = (sub_result.z == 0.0f) ? 1.0f : 0.0f;
  result.y = (sub_result.y == 0.0f) ? 1.0f : 0.0f;
  result.x = (sub_result.x == 0.0f) ? 1.0f : 0.0f;

  if (ExportKillPixel(result) == TRUE) {
    KillPixel = TRUE;
  }

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* KILLGT_V - vector pipe */
Vector4 KILLGT_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 sub_result;
  input1 = Normalize(input1);
  input2 = Normalize(input2);

  // Basically works same as SETGT_V
  sub_result.w = input1.w + -input2.w;
  sub_result.z = input1.z + -input2.z;
  sub_result.y = input1.y + -input2.y;
  sub_result.x = input1.x + -input2.x;

  result.w = (sub_result.w > 0.0f) ? 1.0f : 0.0f;
  result.z = (sub_result.z > 0.0f) ? 1.0f : 0.0f;
  result.y = (sub_result.y > 0.0f) ? 1.0f : 0.0f;
  result.x = (sub_result.x > 0.0f) ? 1.0f : 0.0f;

  if (ExportKillPixel(result) == TRUE) {
    KillPixel = TRUE;
  }

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* KILLGE_V - vector pipe */
Vector4 KILLGE_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 sub_result;

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  // Basically works same as SETGE_V
  sub_result.w = input1.w + -input2.w;
  sub_result.z = input1.z + -input2.z;
  sub_result.y = input1.y + -input2.y;
  sub_result.x = input1.x + -input2.x;

  result.w = (sub_result.w >= 0.0f) ? 1.0f : 0.0f;
  result.z = (sub_result.z >= 0.0f) ? 1.0f : 0.0f;
  result.y = (sub_result.y >= 0.0f) ? 1.0f : 0.0f;
  result.x = (sub_result.x >= 0.0f) ? 1.0f : 0.0f;

  if (ExportKillPixel(result) == TRUE) {
    KillPixel = TRUE;
  }

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* KILLNE_V - vector pipe */
Vector4 KILLNE_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 sub_result;

  input1 = Normalize(input1);
  input2 = Normalize(input2);

  // Basically works same as SETNE_V
  sub_result.w = input1.w + -input2.w;
  sub_result.z = input1.z + -input2.z;
  sub_result.y = input1.y + -input2.y;
  sub_result.x = input1.x + -input2.x;

  result.w = (sub_result.w != 0.0f) ? 1.0f : 0.0f;
  result.z = (sub_result.z != 0.0f) ? 1.0f : 0.0f;
  result.y = (sub_result.y != 0.0f) ? 1.0f : 0.0f;
  result.x = (sub_result.x != 0.0f) ? 1.0f : 0.0f;

  if (ExportKillPixel(result) == TRUE) {
    KillPixel = TRUE;
  }

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* DST_V - vector pipe */
Vector4 DST_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 input2 = SRC_B;
  Vector4 result;

  input1norm = Normalize(input1);
  input2norm = Normalize(input2);

  result.w = input2.w;
  result.z = input1.z;
  result.y = input1norm.y * input2norm.y;
  result.x = 1.0f;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* MOVA_V - vector pipe */
Vector4 MOVA_V(Vector4 SRC_A, Vector4 SRC_B)
{
  Vector4 input1 = SRC_A;
  Vector4 result;
  float scalar_result;

  // SRC_A.w only for address

  input1 = Normalize(input1);

  //
  // Do round to nearest
  //
  scalar_result = input1.w + 0.5f;

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  result = FLOOR_V(result);

  AddressReg = ExportAddressReg(result.w);

  // return max.  Note if SRC_A = SRC_B then this is like a mov
  result = MAX_V(SRC_A, SRC_B);

  return result;

  // Special Considerations (per component)
  // 1.)  See FLOOR_V
  // 2.)  See MAX_V
  // 3.)  See ExportAddressReg() for conversion to signed 9-bit integer
}
```

Scalar Instructions

```
/* ADD_S - scalar pipe */
Vector4 ADD_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  scalar_result = input1.w + input1.x;

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result.w;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* ADD_PREV_S - scalar pipe */
Vector4 ADD_PREV_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  scalar_result = input1.w + PreviousScalar;

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* MUL_S - scalar pipe */
Vector4 ADD_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  scalar_result = input1.w * input1.x;

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result.w;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* MUL_PREV_S - scalar pipe */
Vector4 MUL_PREV_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  scalar_result = input1.w * PreviousScalar;
  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* MUL_PREV2_S - scalar pipe */
Vector4 MUL_PREV2_S(Vector4 SRC_C)
{
  // All shader models
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  // Used to emulate LIT instruction

  input1 = Normalize(input1);

  if ( (PreviousScalar == -MAX_FLOAT) ||
       (PreviousScalar == -INFINITY)  ||
       (PreviousScalar is NaN)        ||
       (input1.x <= 0.0f)             ||
       (input1.x is NaN) ) {
    scalar_result = -MAX_FLOAT;  // alternatively, could return -INFINITY
  } else {
    scalar_result = input1.w * PreviousScalar;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* MAX_S - scalar pipe */
Vector4 MAX_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result, sub_result;

  Vector4 input1norm = Normalize(input1);

  scalar_result = input1.x;

  sub_result = input1norm.w + -input1norm.x;

  if (sub_result >= 0.0f) {
    scalar_result = input1.w;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* MIN_S - scalar pipe */
Vector4 MIN_S(Vector4 SRC_C)
```

```
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result, sub_result;

  input1norm = Normalize(input1);

  scalar_result = input1.x;

  sub_result = input1norm.w + -input1norm.x;

  if (sub_result < 0.0f) {
    scalar_result = input1.w;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* SETE_S - scalar pipe */
Vector4 SETE_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  scalar_result = (input1.w == 0.0f) ? 1.0f : 0.0f;

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* SETGT_S - scalar pipe */
Vector4 SETGT_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  scalar_result = (input1.w > 0.0f) ? 1.0f : 0.0f;

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* SETGE_S - scalar pipe */
Vector4 SETGE_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  scalar_result = (input1.w >= 0.0f) ? 1.0f : 0.0f;

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* SETNE_S - scalar pipe */
Vector4 SETNE_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  scalar_result = (input1.w != 0.0f) ? 1.0f : 0.0f;

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* TRUNC_S - scalar pipe */
Vector4 TRUNC_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  // SRC_C.w only

  input1 = Normalize(input1);

  //
  // trunc() returns integer part of floating point. Sign is preserved. So:
  // trunc(3.0f) = 3.0f
  // trunc(3.4f) = 3.0f
  // trunc(3.5f) = 3.0f
  // trunc(3.6f) = 3.0f
  // trunc(-3.0f) = -3.0f
  // trunc(-3.4f) = -3.0f
  // trunc(-3.5f) = -3.0f
  // trunc(-3.6f) = -3.0f
  //

  scalar_result = trunc(input1.w);

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) TRUNC_S(NaN)        = NaN
  // 2.) TRUNC_S(+INFINITY)  = +INFINITY
  // 3.) TRUNC_S(-INFINITY)  = -INFINITY
  // 4.) TRUNC_S(+ZERO)      = +ZERO
  // 5.) TRUNC_S(-ZERO)      = -ZERO
  // 6.) TRUNC_S(+denorm)    = +ZERO      (*)
  // 7.) TRUNC_S(-denorm)    = -ZERO      (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* FLOOR_S - scalar pipe */
Vector4 FLOOR_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  // SRC_C.w only

  input1 = Normalize(input1);  // Also, done in TRUNC_S, but does not hurt

  result = TRUNC_S(input1);

  scalar_result = result.w;

  if ( (input1.w < 0.0f) && (input1.w != scalar_result) )
    scalar_result += -1.0f;

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) See TRUNC_S
  // 2.) FLOOR_S(NaN)        = NaN        (*)
  // 3.) FLOOR_S(+INFINITY)  = +INFINITY  (*)
  // 4.) FLOOR_S(-INFINITY)  = -INFINITY  (*)
  // 5.) FLOOR_S(+ZERO)      = +ZERO      (*)
  // 6.) FLOOR_S(-ZERO)      = -ZERO      (*)
  // 7.) FLOOR_S(+denorm)    = +ZERO      (*)
  // 8.) FLOOR_S(-denorm)    = -ZERO      (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* FRACT_S - scalar pipe */
Vector4 FRACT_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  // SRC_C.w only

  input1 = Normalize(input1);

  result = FLOOR_S(input1);

  scalar_result = input1.w + -result.w;

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) See FLOOR_S
  // 2.) FRACT_S(NaN)          = NaN          (*)
  // 3.) FRACT_S(+/-INFINITY)  = R400_FP_NAN  (*)
  // 4.) FRACT_S(+/-ZERO)      = +ZERO        (*)
  // 5.) FRACT_S(+/-denorm)    = +ZERO        (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* EXP_IEEE_S - scalar pipe */
Vector4 EXP_IEEE_S(Vector4 SRC_C)
{
  // All shader models.
  //
  // No IEEE rules for EXP, but this could be thought of as IEEE version.
  // EXP plays nice around +/-ZERO so no need for clamped version.

  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  // Base 2
  if (input1.w == 0.0f) {
    // Our approximation has other nice accuracy properties, but for now just ensure this
    // for non-fringe input.  Also see special considerations below.
    scalar_result = 1.0f;
  } else {
    scalar_result = Approximate2ToX(input1.w);  // IEEE like behavior
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) EXP_IEEE_S(NaN)         = NaN
  // 2.) EXP_IEEE_S(+ZERO)       = 1.0f       (*)
  // 3.) EXP_IEEE_S(-ZERO)       = 1.0f       (*)
  // 4.) EXP_IEEE_S(+denorm)     = 1.0f       (*)
  // 5.) EXP_IEEE_S(-denorm)     = 1.0f       (*)
  // 6.) EXP_IEEE_S(+INFINITY)   = +INFINITY
  // 7.) EXP_IEEE_S(-INFINITY)   = 0.0f
  // 8.) EXP_IEEE_S(+MAX_FLOAT)  = +INFINITY  // Note this returns same result as +INFINITY
  // 9.) EXP_IEEE_S(-MAX_FLOAT)  = 0.0f       // Note this returns same result as -INFINITY
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* LOG_IEEE_S - scalar pipe */
Vector4 LOG_IEEE_S(Vector4 SRC_C)
{
  // Shader models >= 3.0
  //
  // No IEEE rules for LOG, but this could be thought of as IEEE version.

  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  // Base 2
  if (input1.w == 1.0f) {
    // Our approximation has other nice accuracy properties, but for now just ensure this
    // for non-fringe input.  Also see special considerations below.
    scalar_result = 0.0f;
  } else {
    scalar_result = ApproximateLog2(input1.w);  // IEEE like behavior
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) LOG_IEEE_S(NaN)        = NaN
  // 2.) LOG_IEEE_S(+ZERO)      = -INFINITY
  // 3.) LOG_IEEE_S(-ZERO)      = -INFINITY
  // 4.) LOG_IEEE_S(+denorm)    = -INFINITY    (*)
  // 5.) LOG_IEEE_S(-denorm)    = -INFINITY    (*)
  // 6.) LOG_IEEE_S(+INFINITY)  = +INFINITY
  // 7.) LOG_IEEE_S(-INFINITY)  = R400_FP_NAN
  // 8.) LOG_IEEE_S(<-fnz>)     = R400_FP_NAN
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* LOG_CLAMPED_S - scalar pipe */
Vector4 LOG_CLAMPED_S(Vector4 SRC_C)
{
  // Shader models <= 2.0

  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  // Base 2
  if (input1.w == 1.0f) {
    // Our approximation has other nice accuracy properties, but for now just ensure this
    // for non-fringe input.  Also see special considerations below.
    scalar_result = 0.0f;
  } else {
    result = LOG_IEEE_S(input1);
    scalar_result = result.w;
  }

  // clamp result
  if (scalar_result == -INFINITY) {
    scalar_result = -MAX_FLOAT;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) See LOG_IEEE_S
  // 2.) LOG_CLAMPED_S(NaN)        = NaN
  // 3.) LOG_CLAMPED_S(+ZERO)      = -MAX_FLOAT    (*)
  // 4.) LOG_CLAMPED_S(-ZERO)      = -MAX_FLOAT    (*)
  // 5.) LOG_CLAMPED_S(+denorm)    = -MAX_FLOAT    (*)
  // 6.) LOG_CLAMPED_S(-denorm)    = -MAX_FLOAT    (*)
  // 7.) LOG_CLAMPED_S(+INFINITY)  = +INFINITY
  // 8.) LOG_CLAMPED_S(-INFINITY)  = R400_FP_NAN
  // 9.) LOG_CLAMPED_S(<-fnz>)     = R400_FP_NAN
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* RECIP_IEEE_S - scalar pipe */
Vector4 RECIP_IEEE_S(Vector4 SRC_C)
{
  // Shader models >= 3.0

  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  if (input1.w == 1.0f) {
    // Our approximation has other nice accuracy properties, but for now just ensure this
    // for non-fringe input.  Also see special considerations below.
    scalar_result = 1.0f;
  } else {
    scalar_result = ApproximateReciprocal(input1.w);  // IEEE like behavior
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) RECIP_IEEE_S(NaN)        = NaN
  // 2.) RECIP_IEEE_S(+ZERO)      = +INFINITY
  // 3.) RECIP_IEEE_S(-ZERO)      = -INFINITY
  // 4.) RECIP_IEEE_S(+denorm)    = +INFINITY  (*)
  // 5.) RECIP_IEEE_S(-denorm)    = -INFINITY  (*)
  // 6.) RECIP_IEEE_S(+INFINITY)  = +ZERO
  // 7.) RECIP_IEEE_S(-INFINITY)  = -ZERO
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* RECIP_CLAMPED_S - scalar pipe */
Vector4 RECIP_CLAMPED_S(Vector4 SRC_C)
{
  // Shader models <= 2.0

  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  if (input1.w == 1.0f) {
    // Our approximation has other nice accuracy properties, but for now just ensure this
    // for non-fringe input.  Also see special considerations below.
    scalar_result = 1.0f;
  } else {
    scalar_result = RECIP_IEEE_S(input1);
    scalar_result = input1.w;
  }

  // clamp result
  if (scalar_result == -INFINITY) {
    scalar_result = -MAX_FLOAT;
  }
  if (scalar_result == +INFINITY) {
    scalar_result = +MAX_FLOAT;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) See RECIP_IEEE_S
  // 2.) RECIP_CLAMPED_S(NaN)        = NaN
  // 3.) RECIP_CLAMPED_S(+ZERO)      = +MAX_FLOAT  (*)
  // 4.) RECIP_CLAMPED_S(-ZERO)      = -MAX_FLOAT  (*)
  // 5.) RECIP_CLAMPED_S(+denorm)    = +MAX_FLOAT  (*)
  // 6.) RECIP_CLAMPED_S(-denorm)    = -MAX_FLOAT  (*)
  // 7.) RECIP_CLAMPED_S(+INFINITY)  = +ZERO
  // 8.) RECIP_CLAMPED_S(-INFINITY)  = -ZERO
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* RECIP_FF_S - scalar pipe */
Vector4 RECIP_FF_S(Vector4 SRC_C)
{
  // Useful for emulating parts of fixed function pipeline

  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  if (input1.w == 1.0f) {
    // Our approximation has other nice accuracy properties, but for now just ensure this
    // for non-fringe input.  Also see special considerations below.
    scalar_result = 1.0f;
  } else {
    scalar_result = RECIP_IEEE_S(input1);
    scalar_result = input1.w;
  }

  // clamp result
  if (scalar_result == -INFINITY) {
    scalar_result = -ZERO;
  }
  if (scalar_result == +INFINITY) {
    scalar_result = +ZERO;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) See RECIP_IEEE_S
  // 2.) RECIP_FF_S(NaN)        = NaN
  // 3.) RECIP_FF_S(+ZERO)      = +ZERO  (*)
  // 4.) RECIP_FF_S(-ZERO)      = -ZERO  (*)
  // 5.) RECIP_FF_S(+denorm)    = +ZERO  (*)
  // 6.) RECIP_FF_S(-denorm)    = -ZERO  (*)
  // 7.) RECIP_FF_S(+INFINITY)  = +ZERO
  // 8.) RECIP_FF_S(-INFINITY)  = -ZERO
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* SQRT_IEEE_S - scalar pipe */
Vector4 SQRT_IEEE_S(Vector4 SRC_C)
{
  // All shader models
  // Can be useful for normal compression

  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  // SQRT instruction is IEEE 754 compliant
  if (input.w == 1.0f) {
    // Our approximation has other nice accuracy properties, but for now just ensure this
    // for non-fringe input.  Also see special considerations below.
    scalar_result = 1.0f;
  } else {
    scalar_result = ApproximateSquareRoot(input.w);  // IEEE like behavior
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) SQRT_IEEE_S(NaN)        = NaN
  // 2.) SQRT_IEEE_S(+ZERO)      = +ZERO
  // 3.) SQRT_IEEE_S(-ZERO)      = -ZERO        //  As defined by IEEE
  // 4.) SQRT_IEEE_S(+denorm)    = +ZERO        (*)
  // 5.) SQRT_IEEE_S(-denorm)    = -ZERO        (*)
  // 6.) SQRT_IEEE_S(+INFINITY)  = +INFINITY
  // 7.) SQRT_IEEE_S(-INFINITY)  = R400_FP_NAN
  // 8.) SQRT_IEEE_S(<-fnz>)     = R400_FP_NAN
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* RECIPSQRT_IEEE_S - scalar pipe */
Vector4 RECIPSQRT_CLAMPED_S(Vector4 SRC_C)
{
  // Shader models >= 3.0

  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  if (input.w == 1.0f) {
    // Our approximation has other nice accuracy properties, but for now just ensure this
    // for non-fringe input.  Also see special considerations below.
    scalar_result = 1.0f;
  } else {
    result = ApproximateRecipSquareRoot(input.w);  // IEEE like behavior
    scalar_result = result.w;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) ApproximateRecipSquareRoot(SRC_C) == RECIP_IEEE_S(SQRT_IEEE_S(SRC_C)) is not guaranteed.
  // 2.) RECIPSQRT_IEEE_S(NaN)        = NaN
  // 3.) RECIPSQRT_IEEE_S(+ZERO)      = +INFINITY
  // 4.) RECIPSQRT_IEEE_S(-ZERO)      = -INFINITY
  // 5.) RECIPSQRT_IEEE_S(+denorm)    = +INFINITY    (*)
  // 6.) RECIPSQRT_IEEE_S(-denorm)    = -INFINITY    (*)
  // 7.) RECIPSQRT_IEEE_S(+INFINITY)  = +ZERO
  // 8.) RECIPSQRT_IEEE_S(-INFINITY)  = R400_FP_NAN
  // 9.) RECIPSQRT_IEEE_S(<-fnz>)     = R400_FP_NAN
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* RECIPSQRT_CLAMPED_S - scalar pipe */
Vector4 RECIPSQRT_CLAMPED_S(Vector4 SRC_C)
{
  // Shader models <= 2.0

  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  if (input.w == 1.0f) {
    // Our approximation has other nice accuracy properties, but for now just ensure this
    // for non-fringe input.  Also see special considerations below.
    scalar_result = 1.0f;
  } else {
    result = RECIP_SQRT_IEEE_S(input1);
    scalar_result = result.w;
  }

  // clamp result
  if (scalar_result == -INFINITY) {
    scalar_result = -MAX_FLOAT;
  }
  if (scalar_result == +INFINITY) {
    scalar_result = +MAX_FLOAT;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) See RECIP_SQRT_IEEE_S
  // 2.) RECIPSQRT_CLAMPED_S(NaN)        = NaN
  // 3.) RECIPSQRT_CLAMPED_S(+ZERO)      = +MAX_FLOAT    (*)
  // 4.) RECIPSQRT_CLAMPED_S(-ZERO)      = -MAX_FLOAT    (*)
  // 5.) RECIPSQRT_CLAMPED_S(+denorm)    = +MAX_FLOAT    (*)
  // 6.) RECIPSQRT_CLAMPED_S(-denorm)    = -MAX_FLOAT    (*)
  // 7.) RECIPSQRT_CLAMPED_S(+INFINITY)  = +ZERO
  // 8.) RECIPSQRT_CLAMPED_S(-INFINITY)  = R400_FP_NAN
  // 9.) RECIPSQRT_CLAMPED_S(<-fnz>)     = R400_FP_NAN
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* RECIPSQRT_FF_S - scalar pipe */
Vector4 RECIPSQRT_FF_S(Vector4 SRC_C)
{
  // Useful for emulating parts of fixed function pipeline

  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  if (input.w == 1.0f) {
    // Our approximation has other nice accuracy properties, but for now just ensure this
    // for non-fringe input.  Also see special considerations below.
    scalar_result = 1.0f;
  } else {
    result = RECIP_SQRT_IEEE_S(input1);
    scalar_result = result.w;
  }

  // clamp result
  if (scalar_result == -INFINITY) {
    scalar_result = -ZERO;
  }
  if (scalar_result == +INFINITY) {
    scalar_result = +ZERO;
  }
  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) See RECIP_SQRT_IEEE_S
  // 2.) RECIPSQRT_FF_S(NaN)        = NaN
  // 3.) RECIPSQRT_FF_S(+ZERO)      = +ZERO        (*)
  // 4.) RECIPSQRT_FF_S(-ZERO)      = -ZERO        (*)
  // 5.) RECIPSQRT_FF_S(+denorm)    = +ZERO        (*)
  // 6.) RECIPSQRT_FF_S(-denorm)    = -ZERO        (*)
  // 7.) RECIPSQRT_FF_S(+INFINITY)  = +ZERO
  // 8.) RECIPSQRT_FF_S(-INFINITY)  = R400_FP_NAN
  // 9.) RECIPSQRT_FF_S(<-fnz>)     = R400_FP_NAN
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* MOVA_S - scalar pipe */
Vector4 MOVA_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;

  // SRC_C.w only

  input1 = Normalize(input1);  // Also, done in FLOOR_S, but does not hurt

  //
  // Do round to nearest
  //
  input1.w += 0.5f;
  result = FLOOR_S(input1);

  AddressReg = ExportAddressReg(result.w);

  result = MAX_S(SRC_C);

  return result;

  // Special Considerations (per component)
  // 1.)  See FLOOR_S
  // 2.)  See MAX_S
  // 3.)  See ExportAddressReg() for conversion to signed 9-bit signed integer
}

/* MOVA_FLOOR_S - scalar pipe */
Vector4 MOVA_FLOOR_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;

  // SRC_C.w only

  input1 = Normalize(input1);  // Also, done in FLOOR_S, but does not hurt

  //
  // Do floor
  //
  result = FLOOR_S(input1);

  AddressReg = ExportAddressReg(result.w);

  result = MAX_S(SRC_C);

  return result;

  // Special Considerations (per component)
  // 1.)  See FLOOR_S
  // 2.)  See MAX_S
  // 3.)  See ExportAddressReg() for conversion to signed 9-bit integer
}

/* SUB_S - scalar pipe */
Vector4 SUB_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  // Need SUB because neg input modifier applies to all channels of SRC_C

  scalar_result = input1.w + -input1.x;

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result.w;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* SUB_PREV_S - scalar pipe */
Vector4 SUB_PREV_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  // Need SUB_PREV because neg input modifier applies to all channels of SRC_C

  scalar_result = input1.w + -PreviousScalar;

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* PRED_SETE_S - scalar pipe */
Vector4 PRED_SETE_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float   scalar_result;

  //
  // Scalar predicate instructions designed for:
  //  Predicate counter in SRC_C.w
  //  Condition in SRC_C.w
  //  Return result is updated predicate counter
  //

  input1 = Normalize(input1);

  if (input1.w == 0.0f) {
    ExportPredicateReg(EXECUTE);
    scalar_result = 0.0f;
  } else {
    ExportPredicateReg(SKIP);
    scalar_result = 1.0f;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) PRED_SETE_S(NaN)  = 1.0f, evaluates to SKIP  (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* PRED_SETGT_S - scalar pipe */
Vector4 PRED_SETGT_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float   scalar_result;

  //
  // Scalar predicate instructions designed for:
  //  Predicate counter in SRC_C.w
  //  Condition in SRC_C.w
  //  Return result is updated predicate counter
  //

  input1 = Normalize(input1);

  if (input1.w > 0.0f) {
    ExportPredicateReg(EXECUTE);
    scalar_result = 0.0f;
  } else {
    ExportPredicateReg(SKIP);
    scalar_result = 1.0f;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) PRED_SETGT_S(NaN)  = 1.0f, evaluates to SKIP  (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* PRED_SETGE_S - scalar pipe */
Vector4 PRED_SETGE_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float   scalar_result;

  //
  // Scalar predicate instructions designed for:
  //  Predicate counter in SRC_C.w
  //  Condition in SRC_C.w
  //  Return result is updated predicate counter
  //

  input1 = Normalize(input1);

  if (input1.w >= 0.0f) {
    ExportPredicateReg(EXECUTE);
    scalar_result = 0.0f;
  } else {
    ExportPredicateReg(SKIP);
    scalar_result = 1.0f;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) PRED_SETGE_S(NaN)  = 1.0f, evaluates to SKIP  (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* PRED_SETNE_S - scalar pipe */
Vector4 PRED_SETNE_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float   scalar_result;

  //
  // Scalar predicate instructions designed for:
  //  Predicate counter in SRC_C.w
  //  Condition in SRC_C.w
  //  Return result is updated predicate counter
  //

  input1 = Normalize(input1);

  if (input1.w != 0.0f) {
    ExportPredicateReg(EXECUTE);
    scalar_result = 0.0f;
  } else {
    ExportPredicateReg(SKIP);
    scalar_result = 1.0f;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) PRED_SETNE_S(NaN)  = 0.0f, evaluates to EXECUTE  (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* PRED_SET_INV_S - scalar pipe */
Vector4 PRED_SET_INV_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float   scalar_result;

  //
  // Scalar predicate instructions designed for:
  //  Predicate counter in SRC_C.w
  //  Condition in SRC_C.w
  //  Return result is updated predicate counter
  //

  input1norm = Normalize(input1);

  if (input1norm.w == 1.0f) {
    ExportPredicateReg(EXECUTE);
    scalar_result = 0.0f;
  } else {
    ExportPredicateReg(SKIP);
    if (input1norm.w == 0.0f) {
      scalar_result = 1.0f;
    } else {
      scalar_result = input1.w;
    }
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) PRED_SET_INV_S(NaN)  = NaN, evaluates to SKIP  (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* PRED_SET_POP_S - scalar pipe */
Vector4 PRED_SET_POP_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float   scalar_result;

  //
  // Scalar predicate instructions designed for:
  //  Predicate counter in SRC_C.w
  //  Condition in SRC_C.w
  //  Return result is updated predicate counter
  //

  input1 = Normalize(input1);

  scalar_result = input1.w + -1.0f;

  if (scalar_result <= 0.0f) {
    ExportPredicateReg(EXECUTE);
    scalar_result = 0.0f;
  } else {
    ExportPredicateReg(SKIP);
    scalar_result = scalar_result;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) PRED_SET_POP_S(NaN)  = NaN, evaluates to SKIP  (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* PRED_SET_CLR_S - scalar pipe */
Vector4 PRED_SET_CLR_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float   scalar_result;

  //
  // Scalar predicate instructions designed for:
  //  Predicate counter in SRC_C.w
  //  Condition in SRC_C.w
  //  Return result is updated predicate counter
  //

  // SRC_C not used

  ExportPredicateReg(SKIP);

  scalar_result = +MAX_FLOAT;

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) None
}

/* PRED_SET_RESTORE_S - scalar pipe */
Vector4 PRED_SET_RESTORE_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float   scalar_result;

  //
  // Scalar predicate instructions designed for:
  //  Predicate counter in SRC_C.w
  //  Condition in SRC_C.w
  //  Return result is updated predicate counter
  //

  Vector4 input1norm = Normalize(input1);

  if (input1norm.w == 0.0f) {
    ExportPredicateReg(EXECUTE);
    scalar_result = 0.0f;
  } else {
    ExportPredicateReg(SKIP);
    scalar_result = input1.w;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) PRED_SET_RESTORE_S(NaN)  = NaN, evaluates to SKIP  (*)
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* KILLE_S - scalar pipe */
Vector4 KILLE_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  scalar_result = 0.0f;

  if (input1.w == 0.0f) {
    scalar_result = 1.0f;
  }
  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  if (ExportKillPixel(result) == TRUE) {
    KillPixel = TRUE;
  }

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* KILLGT_S - scalar pipe */
Vector4 KILLGT_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  scalar_result = 0.0f;

  if (input1.w > 0.0f) {
    scalar_result = 1.0f;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  if (ExportKillPixel(result) == TRUE) {
    KillPixel = TRUE;
  }

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* KILLGE_S - scalar pipe */
Vector4 KILLGE_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  scalar_result = 0.0f;

  if (input1.w >= 0.0f) {
    scalar_result = 1.0f;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  if (ExportKillPixel(result) == TRUE) {
    KillPixel = TRUE;
  }

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}
```

```
/* KILLNE_S - scalar pipe */
Vector4 KILLNE_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  scalar_result = 0.0f;

  if (input1.w != 0.0f) {
    scalar_result = 1.0f;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  if (ExportKillPixel(result) == TRUE) {
    KillPixel = TRUE;
  }

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* KILLONE_S - scalar pipe */
Vector4 KILLONE_S(Vector4 SRC_C)
{
  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  scalar_result = 0.0f;

  if (input1.w == 1.0f) {
    scalar_result = 1.0f;
  }

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  if (ExportKillPixel(result) == TRUE) {
    KillPixel = TRUE;
  }

  return result;

  // Special Considerations (per component)
  // 1.) None - follows floating point rules
}

/* SIN_S - scalar pipe */
Vector4 SIN_S(Vector4 SRC_C)
{
  // All shader models
  //
  // Valid input domain: [-PI, +PI] (implementation may clamp input values to this range except
  // for +INFINITY, -INFINITY and NaN)

  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);
  scalar_result = ApproximateSin(input.w);

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) SIN_S(NaN)         = NaN
  // 2.) SIN_S(+ZERO)       = +ZERO
  // 3.) SIN_S(-ZERO)       = -ZERO
  // 4.) SIN_S(+denorm)     = +ZERO        (*)
  // 5.) SIN_S(-denorm)     = -ZERO        (*)
  // 6.) SIN_S(+INFINITY)   = R400_FP_NAN
  // 7.) SIN_S(-INFINITY)   = R400_FP_NAN
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}

/* COS_S - scalar pipe */
Vector4 COS_S(Vector4 SRC_C)
{
  // All shader models
  //
  // Valid input domain: [-PI, +PI] (implementation may clamp input values to this range except
  // for +INFINITY, -INFINITY and NaN)

  Vector4 input1 = SRC_C;
  Vector4 result;
  float scalar_result;

  input1 = Normalize(input1);

  scalar_result = ApproximateCos(input.w);

  result.w = scalar_result;
  result.z = scalar_result;
  result.y = scalar_result;
  result.x = scalar_result;

  return result;

  // Special Considerations (per component)
  // 1.) COS_S(NaN)         = NaN
  // 2.) COS_S(+ZERO)       = 1.0f
  // 3.) COS_S(-ZERO)       = 1.0f
  // 4.) COS_S(+denorm)     = 1.0f        (*)
  // 5.) COS_S(-denorm)     = 1.0f        (*)
  // 6.) COS_S(+INFINITY)   = R400_FP_NAN
  // 7.) COS_S(-INFINITY)   = R400_FP_NAN
  //
  // (*) = Given our floating point rules, the code above will produce these results without special
  //       handling and are shown only for clarity.
}
```

R400 Shader/Texture Pipe Macros

WORKING IN PROGRESS – IGNORE FOR NOW

MOV

The R400 does not have a native MOV instruction. The MOV instruction can be implemented via a MAX (or MAX) instruction using argument selection bits in the ALU instruction appropriately.

```
MOV R1, R0
Becomes:
MAX R1, R0, R0  // vector


MOV R1.x___, R0.yyyy
```

Becomes:

```
MAX R1.x___, R0.yyyy, R0.yyyy  // scalar
```

YUV → RGB Conversion

First a couple notes:
1.) U == Cb, V == Cr
2.) with default swizzle, dst GPR after FetchTextureMap holds (Z=Cb, Y=Y, X=Cr)
3.) R400 only supports replicating Cb and Cr values (no interpolating)

```
-----Shader using shader pipe to debias Cr/Cb----

Constants (XYZW)
C0 = { ( (-16.0f * 0.00456621f) + (-128.0f *  0.0f)        +  (-128.0f *  0.00625893f) ),
       ( (-16.0f * 0.00456621f) + (-128.0f * -0.00153632f) +  (-128.0f * -0.00318811f) ),
       ( (-16.0f * 0.00456621f) + (-128.0f *  0.00791071f) +  (-128.0f *  0.0f)        ),
          1.0f }                                       // debias Y/Cb/Cr
C1 = { 0.00456621f,  0.00456621f, 0.00456621f, 0.0f }  // Y
C2 = { 0.0f,        -0.00153632f, 0.00791071f, 0.0f }  // Cb/U
C3 = { 0.00625893f, -0.00318811f, 0.0f,        0.0f }  // Cr/V

texture0 = YUV texture (packed 422)
R0 = source texture coordinate

FetchTextureMap R1.xyz1, R0, texture0
MAD R2, R1.yyyy, C1, C0
MAD R2, R1.zzzz, C2, R2
MAD_sat R2, R1.xxxx, C3, R2

This will put into R2 X=R, Y=G, Z=B, W=1.0f where RGB=[0..1]

The texture pipe/constant should be programmed like this:
DATA_FORMAT    = FMT_Cr_Y1_Cb_Y0 or FMT_Y1_Cr_Y0_Cb
NUM_FORMAT_ALL = INT
FORMAT_COMP_X  = unsigned
FORMAT_COMP_Y  = unsigned
FORMAT_COMP_Z  = unsigned
FORMAT_COMP_W  = unsigned (really a don't care)
BORDER_COLOR   = ACbYCr black
EXP_ADJUST_ALL = 0

-----Shader using texture pipe to debias Cr/Cb-----------

Constants (XYZW)
C0 = { (-16.0f * 0.00456621f), (-16.0f * 0.00456621f), (-16.0f * 0.00456621f), 1.0f }  //debias Y
C1 = { 0.00456621f,  0.00456621f, 0.00456621f, 0.0f }                                  // Y
C2 = { 0.0f,        -0.00153632f, 0.00791071f, 0.0f }                                  // U/Cb
C3 = { 0.00625893f, -0.00318811f, 0.0f,        0.0f }                                  // V/Cr

texture0 = YUV texture (packed 422)
R0 = source texture coordinate

FetchTextureMap R1.xyz1, R0, texture0
MAD R2, R1.yyyy, C1, C0
MAD R2, R1.zzzz, C2, R2
MAD_sat R2, R1.xxxx, C3, R2

This will put into R2 X=R, Y=G, Z=B, W=1.0f where RGB=[0..1]

The texture pipe/constant should be programmed like this:
DATA_FORMAT    = FMT_Cr_Y1_Cb_Y0 or FMT_Y1_Cr_Y0_Cb
NUM_FORMAT_ALL = INT
FORMAT_COMP_X  = unsigned bias
FORMAT_COMP_Y  = unsigned
FORMAT_COMP_Z  = unsigned bias
FORMAT_COMP_W  = unsigned (really a don't care)
BORDER_COLOR   = ACbYCr black
EXP_ADJUST_ALL = 0
```

LIT

```
SETGT   Rout@z  = Rin.xxxx, Czero;             LOG       Rtemp  = Rin y;
MUL     Rout@y  = Rout.zzzz, Rin.xxxx;         MUL_PREV2 Rtemp  = Rin w x;
MOV     Rout@xw = Cone;                        EXP       Rout@z = Rtemp;
```

POW

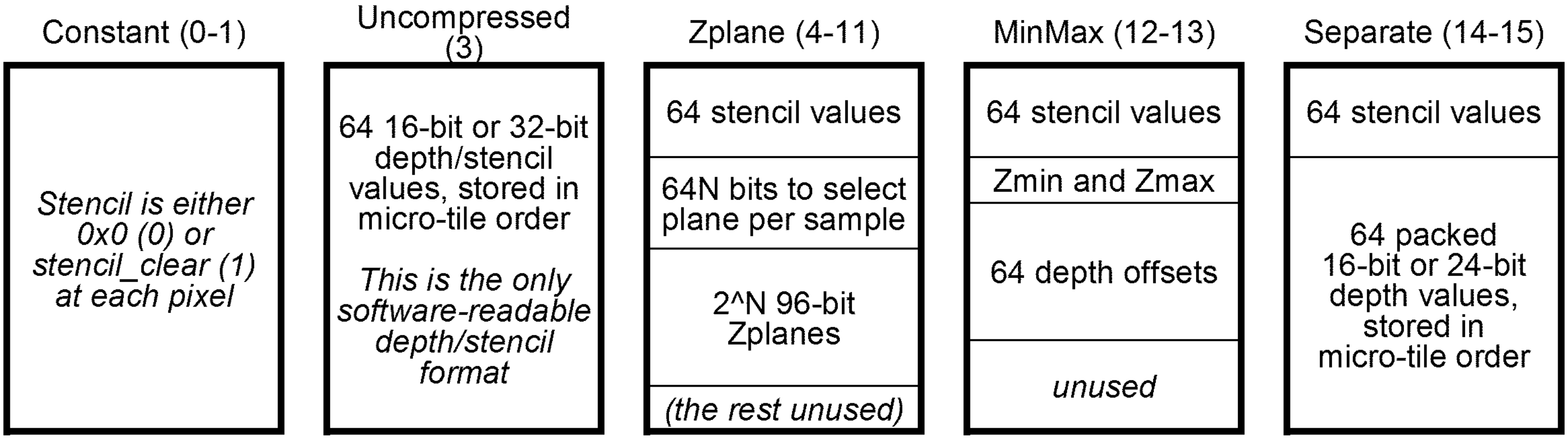
POW - vector x power y
Instruction: POW DST, SRC1, SRC2
Description: Computes x power y.
Input power must be a scalar. Scalar result is replicated to all 4 output channels.
This is a macro instruction, which takes 3 instruction slots.
pow(x,y) could be expanded as exp(y * log(x)).
DST should be a temporary register
Macro expansion:

```
log DST, SRC1
mul DST, DST, SRC1
exp DST, DST
```

SINCOS

Cubemap Texture Fetch

Texture With Border Color Fetch



Constant (0-1)

Stencil is either 0x0 (0) or stencil_clear (1) at each pixel

Uncompressed (3)

64 16-bit or 32-bit depth/stencil values, stored in micro-tile order

This is the only software-readable depth/stencil format

Zplane (4-11)

64 stencil values

64N bits to select plane per sample

2^N 96-bit Zplanes

(the rest unused)

MinMax (12-13)

64 stencil values

Zmin and Zmax

64 depth offsets

unused

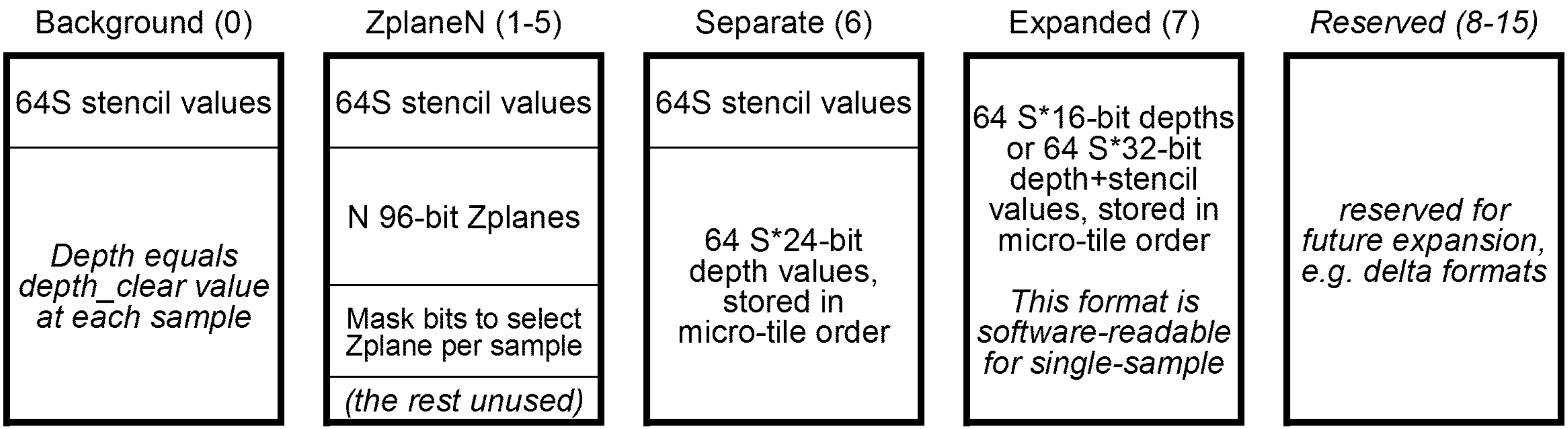
Separate (14-15)

64 stencil values

64 packed 16-bit or 24-bit depth values, stored in micro-tile order

127	96	95	64	63	32	31	0
T 7	T 6	T 5	T 4	T 3	T 2	T 1	T 0
T 15	T 14	T 13	T 12	T11	T 10	T 9	T 8
T 23	T 22	T 21	T 20	T 19	T 18	T 17	T 16

127			96	95			64	63			32	31			0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32



Background (0)

64S stencil values

Depth equals depth_clear value at each sample

ZplaneN (1-5)

64S stencil values

N 96-bit Zplanes

Mask bits to select Zplane per sample

(the rest unused)

Separate (6)

64S stencil values

64 S*24-bit depth values, stored in micro-tile order

Expanded (7)

64 S*16-bit depths or 64 S*32-bit depth+stencil values, stored in micro-tile order

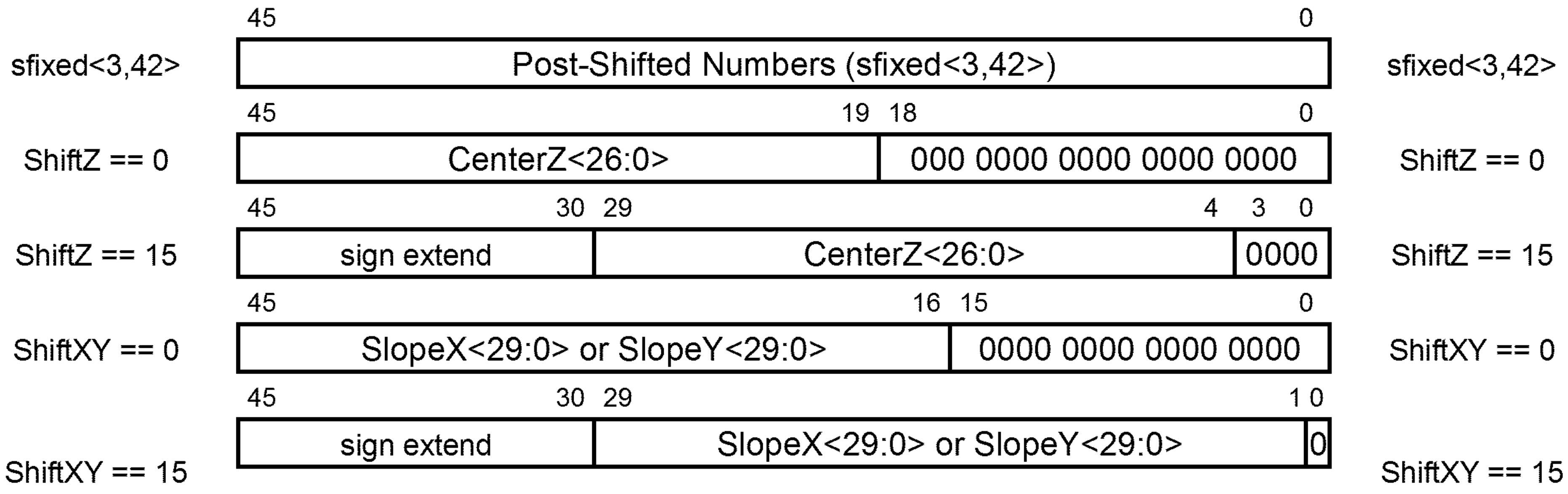
This format is software-readable for single-sample

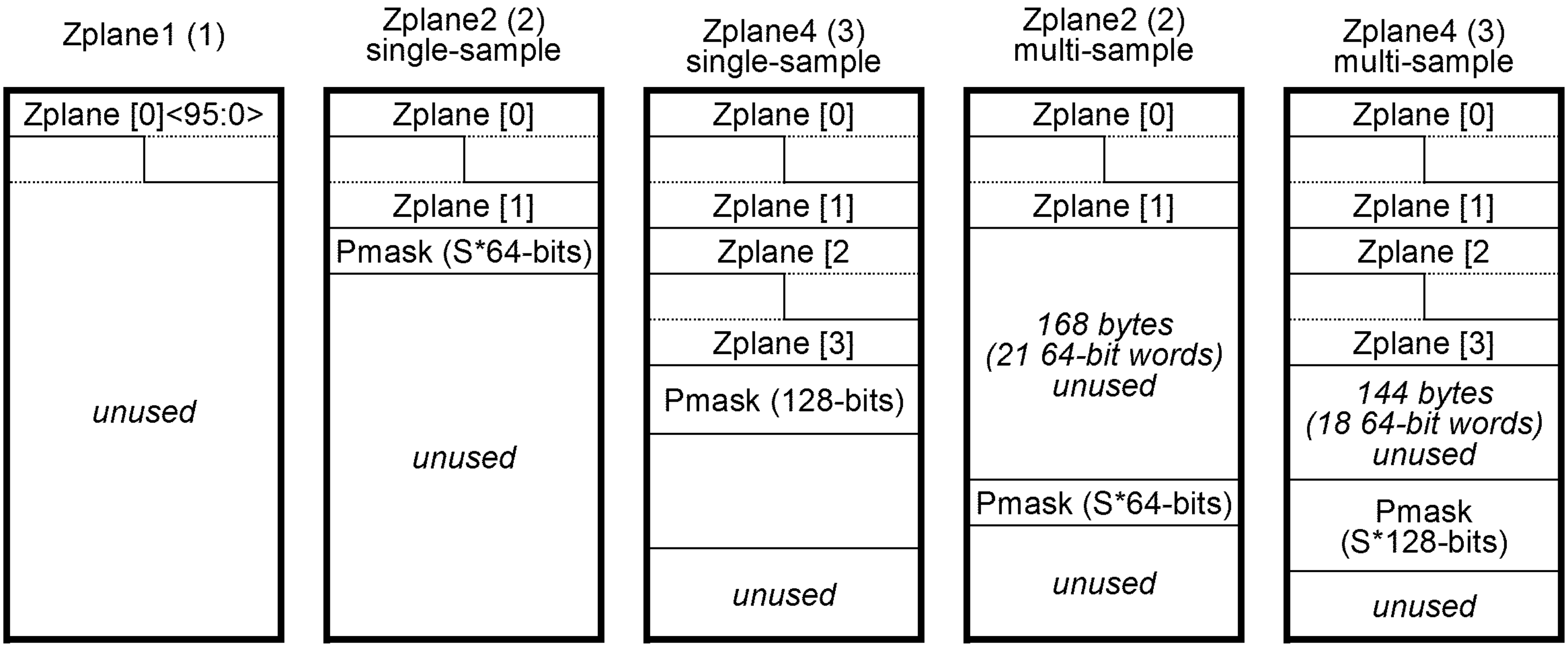
Reserved (8-15)

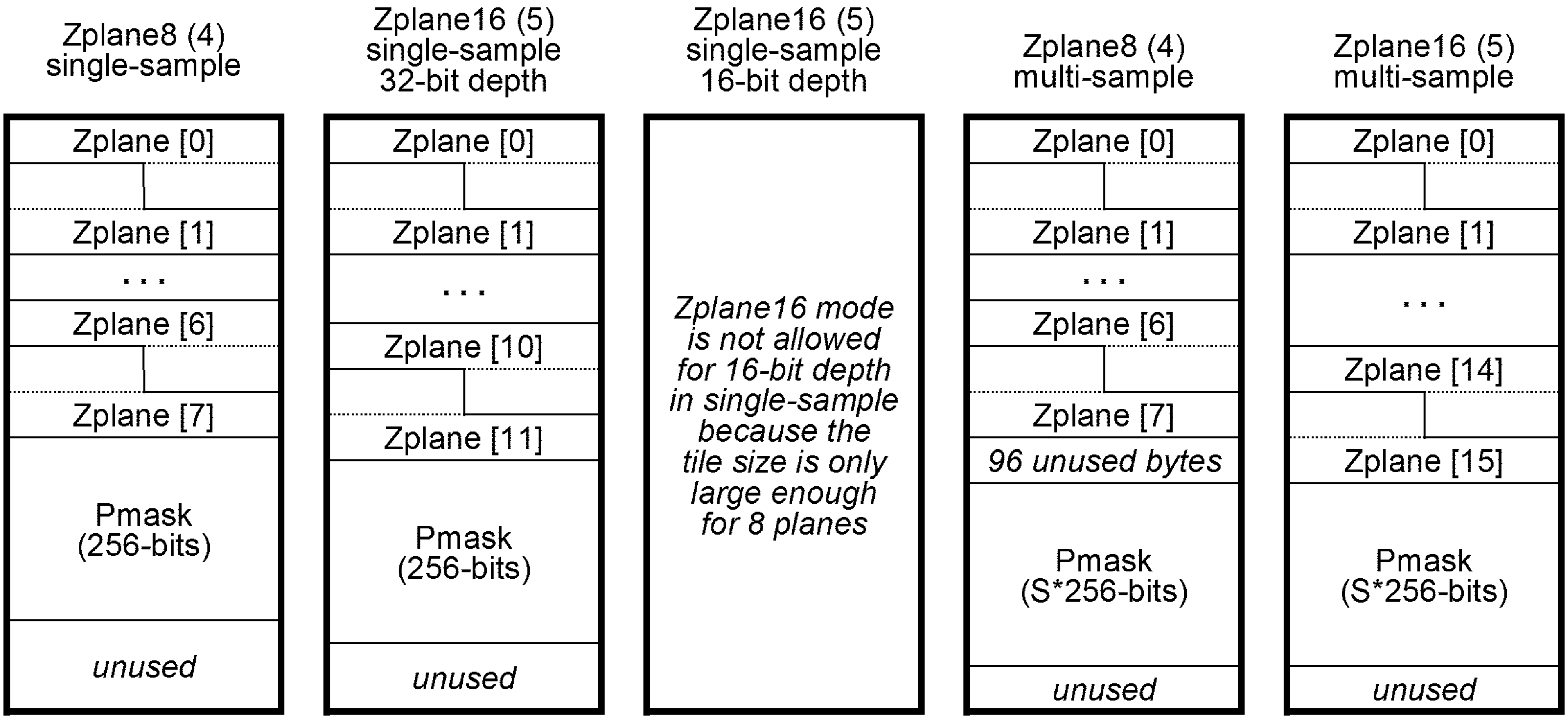
reserved for future expansion, e.g. delta formats

59 30	29 0
SlopeY<29:0> (sbfixed<3,26>)	SlopeX<29:0> (sbfixed<3,26>)

95 92	91 65	64	63 60
ShiftZ<3:0>	CenterZ<26:0> (sbfixed<3,23>)	MultiSample	ShiftXY<3:0>


45
0
sfixed<3,42>
Post-Shifted Numbers (sfixed<3,42>)
sfixed<3,42>
45
19
18
0
ShiftZ == 0
CenterZ<26:0>
000 0000 0000 0000 0000
ShiftZ == 0
45
30
29
4
3
0
ShiftZ == 15
sign extend
CenterZ<26:0>
0000
ShiftZ == 15
45
16
15
0
ShiftXY == 0
SlopeX<29:0> or SlopeY<29:0>
0000 0000 0000 0000
ShiftXY == 0
45
30
29
1
0
sign extend
SlopeX<29:0> or SlopeY<29:0>
0
ShiftXY == 15
ShiftXY == 15


Zplane1 (1)
Zplane [0]<95:0>
unused
Zplane2 (2)
single-sample
Zplane [0]
Zplane [1]
Pmask (S*64-bits)
unused
Zplane4 (3)
single-sample
Zplane [0]
Zplane [1]
Zplane [2
Zplane [3]
Pmask (128-bits)
unused
Zplane2 (2)
multi-sample
Zplane [0]
Zplane [1]
168 bytes
(21 64-bit words)
unused
Pmask (S*64-bits)
unused
Zplane4 (3)
multi-sample
Zplane [0]
Zplane [1]
Zplane [2
Zplane [3]
144 bytes
(18 64-bit words)
unused
Pmask
(S*128-bits)
unused



Zplane8 (4)
single-sample

Zplane [0]

Zplane [1]

. . .

Zplane [6]

Zplane [7]

Pmask
(256-bits)

unused

Zplane16 (5)
single-sample
32-bit depth

Zplane [0]

Zplane [1]

. . .

Zplane [10]

Zplane [11]

Pmask
(256-bits)

unused

Zplane16 (5)
single-sample
16-bit depth

Zplane16 mode is not allowed for 16-bit depth in single-sample because the tile size is only large enough for 8 planes

Zplane8 (4)
multi-sample

Zplane [0]

Zplane [1]

. . .

Zplane [6]

Zplane [7]

96 unused bytes

Pmask
(S*256-bits)

unused

Zplane16 (5)
multi-sample

Zplane [0]

Zplane [1]

. . .

Zplane [14]

Zplane [15]

Pmask
(S*256-bits)

unused

	64N-1 56N	56N-1 48N	48N-1 40N	40N-1 32N	32N-1 24N	24N-1 16N	16N-1 8N	8N-1 0
0	pmsk(0-7,7)	pmsk(0-7,5)	pmsk(0-7,6)	pmsk(0-7,4)	pmsk(0-7,3)	pmsk(0-7,1)	pmsk(0-7,2)	pmsk(0-7,0)

N-bit Pmask, 1-sample: 16N-bit interleave of even/odd scanlines

	64N-1 48N	48N-1 32N	32N-1 16N	16N-1 0
0	pmask (0-7, 3)	pmask (0-7, 2)	pmask (0-7, 1)	pmask (0-7, 0)
1	pmask (0-7, 7)	pmask (0-7, 6)	pmask (0-7, 5)	pmask (0-7, 4)

N-bit Pmask, 2-sample: 16N-bit interleave of even/odd scanlines

	64N-1 48N	48N-1 32N	32N-1 16N	16N-1 0
0	pmask (4-7, 1)	pmask (4-7, 0)	pmask (0-3, 1)	pmask (0-3, 0)
1	pmask (4-7, 3)	pmask (4-7, 2)	pmask (0-3, 3)	pmask (0-3, 2)
2	pmask (4-7, 5)	pmask (4-7, 4)	pmask (0-3, 5)	pmask (0-3, 4)
3	pmask (4-7, 7)	pmask (4-7, 6)	pmask (0-3, 7)	pmask (0-3, 6)

N-bit Pmask, 4-sample: 16N-bit interleave of even/odd scanlines

	64N-1 48N	48N-1 32N	32N-1 16N	16N-1 0
0	pmask (2-3, 1)	pmask (2-3, 0)	pmask (0-1, 1)	pmask (0-1, 0)
1	pmask (6-7, 1)	pmask (6-7, 0)	pmask (4-5, 1)	pmask (4-5, 0)
2	pmask (2-3, 3)	pmask (2-3, 2)	pmask (0-1, 3)	pmask (0-1, 2)
3	pmask (6-7, 3)	pmask (6-7, 2)	pmask (4-5, 3)	pmask (4-5, 2)
4	pmask (2-3, 5)	pmask (2-3, 4)	pmask (0-1, 5)	pmask (0-1, 4)
5	pmask (6-7, 5)	pmask (6-7, 4)	pmask (4-5, 5)	pmask (4-5, 4)
6	pmask (2-3, 7)	pmask (2-3, 6)	pmask (0-1, 7)	pmask (0-1, 6)
7	pmask (6-7, 7)	pmask (6-7, 6)	pmask (4-5, 7)	pmask (4-5, 6)

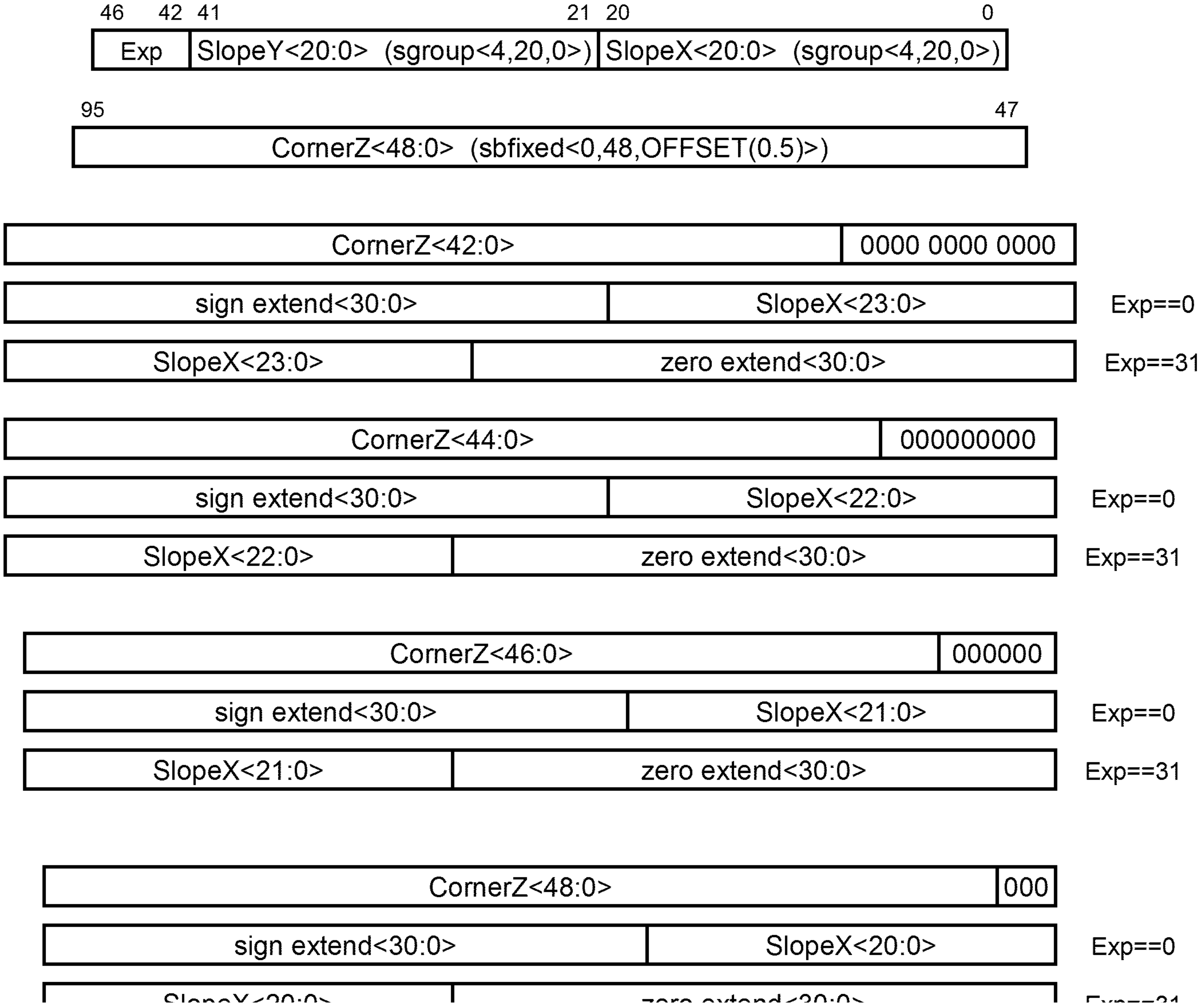
N-bit Pmask, 8-sample: 16N-bit interleave of even/odd scanlines

<table>
<tr><th></th><th>64N-1 56N</th><th>56N-1 48N</th><th>48N-1 40N</th><th>40N-1 32N</th><th>32N-1 24N</th><th>24N-1 16N</th><th>16N-1 8N</th><th>8N-1 0</th></tr>
<tr><td>0</td><td>pmsk(0a-2b,3)</td><td>pmsk(5b-7c,1)</td><td>pmsk(0a-2b,2)</td><td>pmsk(5b-7c,0)</td><td colspan="2">pmask (0a-5a, 1)</td><td colspan="2">pmask (0a-5a,0)</td></tr>
<tr><td>1</td><td colspan="2">pmask (0a-5a, 5)</td><td colspan="2">pmask (0a-5a,4)</td><td colspan="2">pmask (2c-7c, 3)</td><td colspan="2">pmask (2c-7,2)</td></tr>
<tr><td>2</td><td colspan="2">pmask (2c-7c, 7)</td><td colspan="2">pmask (2c-7c,6)</td><td>pmsk(0a-2b,7)</td><td>pmsk(5b-7c,5)</td><td>pmsk(0a-2b,6)</td><td>pmsk(5b-7c,4)</td></tr>
</table>

N-bit Pmask, 3-sample: 16N-bit interleave of even/odd scanlines

	64N-1 48N	48N-1 32N	32N-1 16N	16N-1 0
0	pmask (2e-5b, 1)	pmask (2e-5b,0)	pmask (0a-2d, 1)	pmask (0a-2d,0)
1	pmask (0a-2d, 3)	pmask (0a-2d,2)	pmask (5c-7f,1)	pmask (5c-7f,0)
2	pmask (5c-7f,3)	pmask (5c-7f,2)	pmask (2e-5b, 3)	pmask (2e-5b,2)
3	pmask (2e-5b, 5)	pmask (2e-5b,4)	pmask (0a-2d, 5)	pmask (0a-2d,4)
4	pmask (0a-2d, 7)	pmask (0a-2d,6)	pmask (5c-7f,5)	pmask (5c-7f,4)
5	pmask (5c-7f,7)	pmask (5c-7f,6)	pmask (2e-5b, 7)	pmask (2e-5b,6)

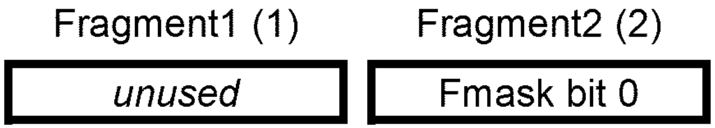
N-bit Pmask, 6-sample: 16N-bit interleave of even/odd scanlines


46
42
41
21
20
0
Exp
SlopeY<20:0> (sgroup<4,20,0>)
SlopeX<20:0> (sgroup<4,20,0>)
95
47
CornerZ<48:0> (sbfixed<0,48,OFFSET(0.5)>)
CornerZ<42:0>
0000 0000 0000
sign extend<30:0>
SlopeX<23:0>
Exp==0
SlopeX<23:0>
zero extend<30:0>
Exp==31
CornerZ<44:0>
000000000
sign extend<30:0>
SlopeX<22:0>
Exp==0
SlopeX<22:0>
zero extend<30:0>
Exp==31
CornerZ<46:0>
000000
sign extend<30:0>
SlopeX<21:0>
Exp==0
SlopeX<21:0>
zero extend<30:0>
Exp==31
CornerZ<48:0>
000
sign extend<30:0>
SlopeX<20:0>
Exp==0
SlopeX<20:0>
zero extend<30:0>
Exp==31

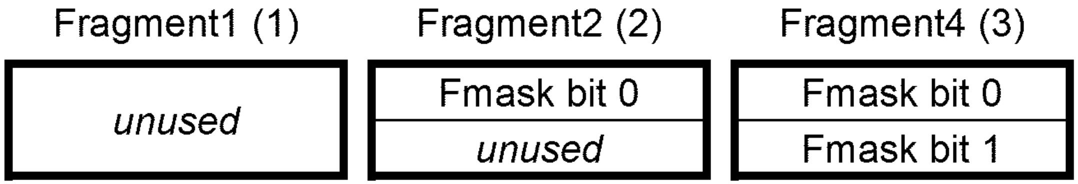
	Background (0)	FragmentN (1-4)	*Reserved (5-6)*	Expanded (7)	*Reserved (8-15)*
Array 0	*Color equals color_clear value at sample 0 of each pixel*	64 color values per 8x8 tile, in micro-tile order, for fragment 0	*Reserved for other formats, e.g. compressed fragment masks*	64 color values per 8x8 tile, in micro-tile order, for sample 0	*Reserved, e.g. for specifying alpha saturation*
	. . .	. . .	. . .	. . .	. . .
Array S-1	*Color equals color_clear value at sample S-1 of each pixel*	64 color values per 8x8 tile, in micro-tile order, for fragment F-1	*Reserved for other formats, e.g. compressed fragment masks*	64 color values per 8x8 tile, in micro-tile order, for sample S-1	*Reserved, e.g. for specifying alpha saturation*

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
word 0	(7,1)	(6,1)	(5,1)	(4,1)	(3,1)	(2,1)	(1,1)	(0,1)	(7,0)	(6,0)	(5,0)	(4,0)	(3,0)	(2,0)	(1,0)	(0,0)
word 1	(7,3)	(6,3)	(5,3)	(4,3)	(3,3)	(2,3)	(1,3)	(0,3)	(7,2)	(6,2)	(5,2)	(4,2)	(3,2)	(2,2)	(1,2)	(0,2)
word 2	(7,5)	(6,5)	(5,5)	(4,5)	(3,5)	(2,5)	(1,5)	(0,5)	(7,4)	(6,4)	(5,4)	(4,4)	(3,4)	(2,4)	(1,4)	(0,4)
word 3	(7,7)	(6,7)	(5,7)	(4,7)	(3,7)	(2,7)	(1,7)	(0,7)	(7,6)	(6,6)	(5,6)	(4,6)	(3,6)	(2,6)	(1,6)	(0,6)

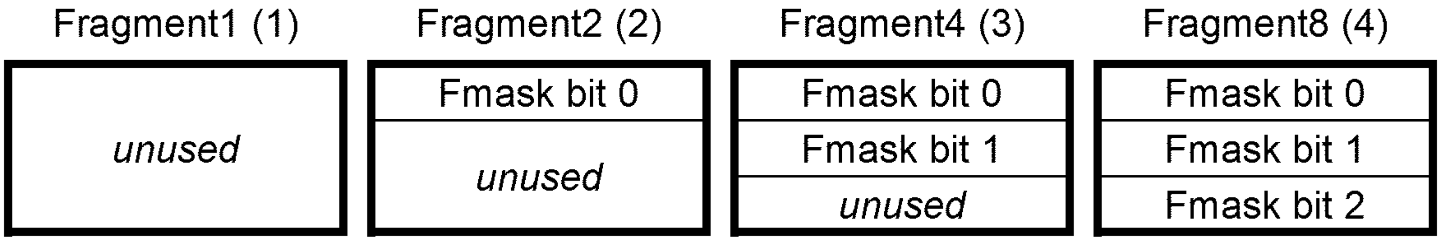
	63 48	47 32	31 16	15 0
Fmask bit for S samples	word 3, sample 0	word 2, sample 0	word 1, sample 0	word 0, sample 0
	. . .			
	word 3, sample S-1	word 2, sample S-1	word 1, sample S-1	word 0, sample S-1



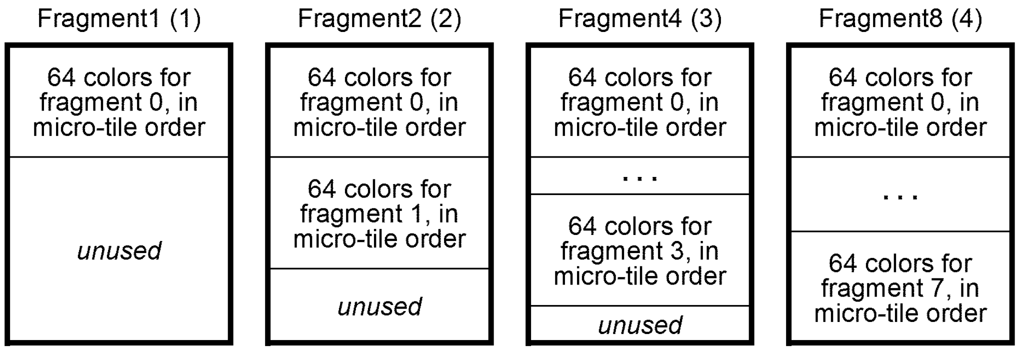
Cmask Modes for 2 Samples per Pixel

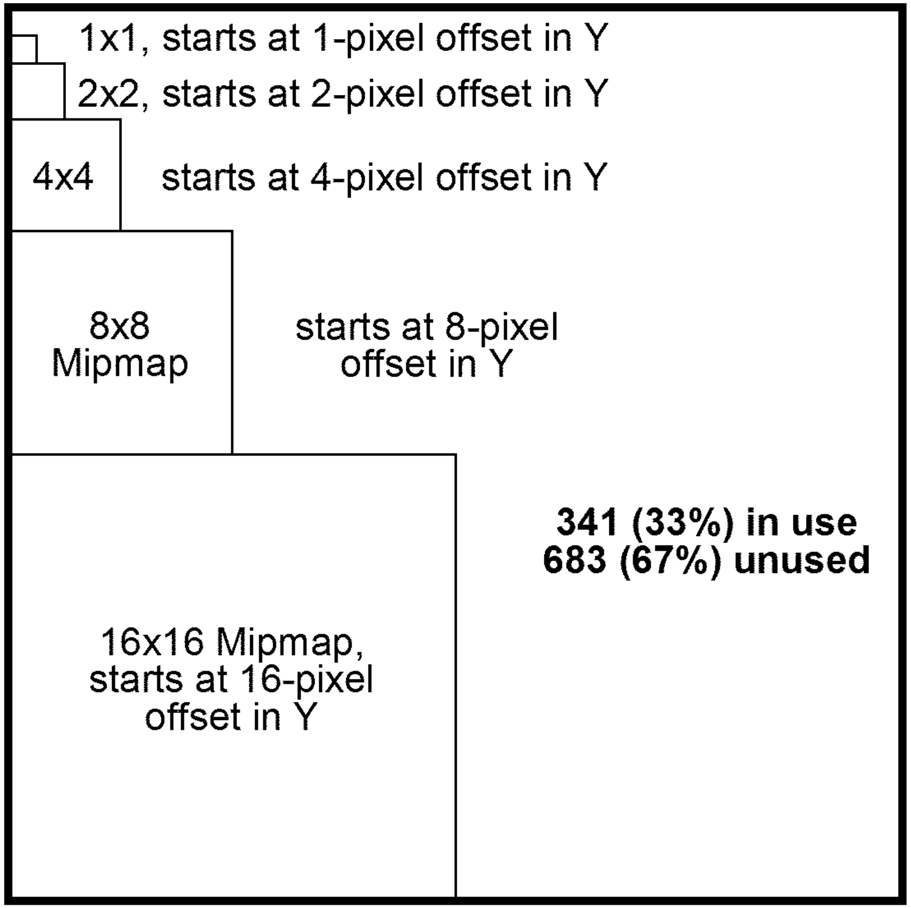


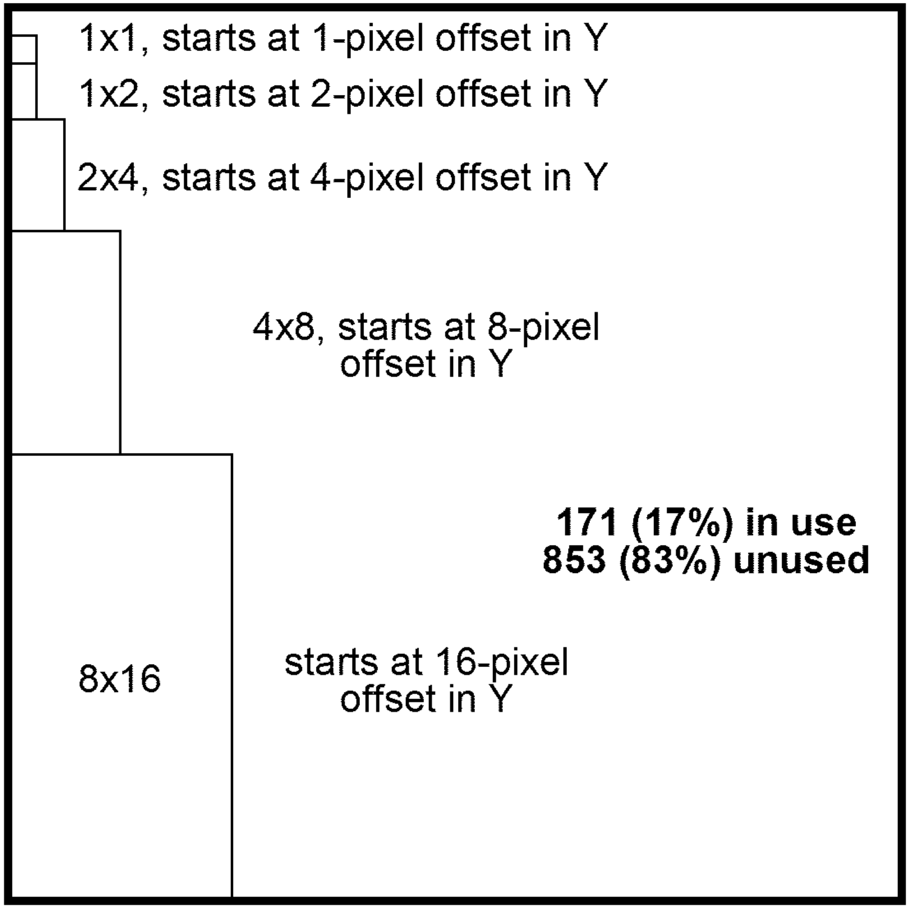
Cmask Modes for 3 or 4 Samples per Pixel

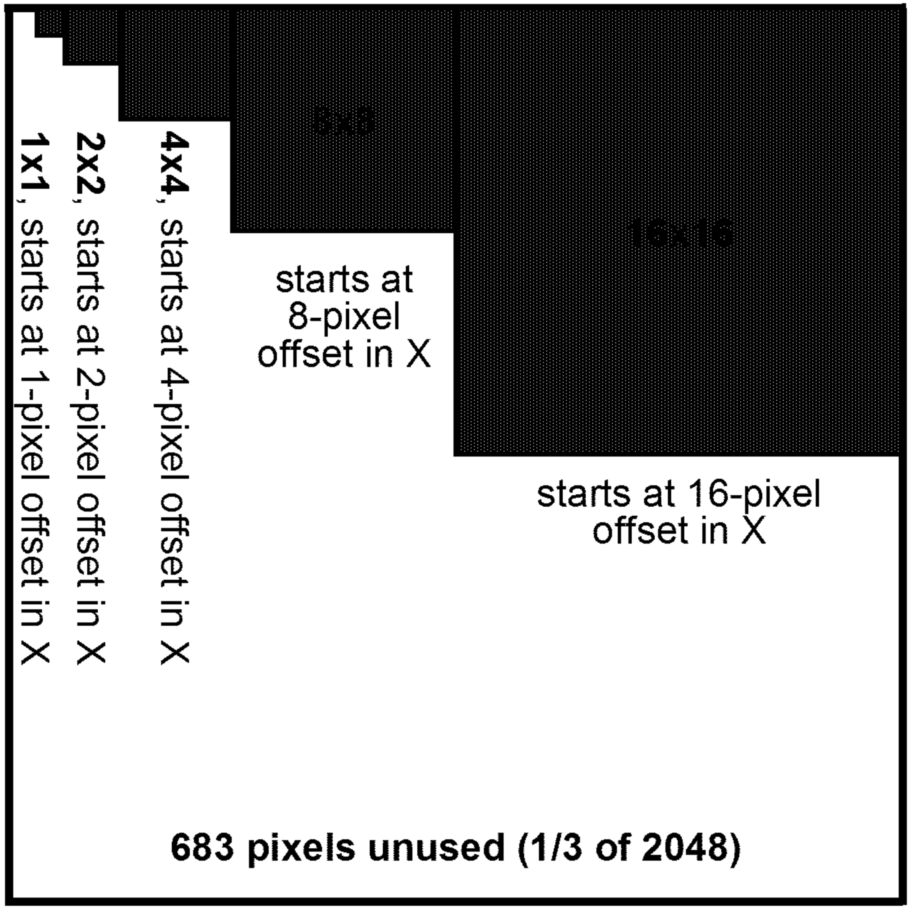


Cmask Modes for 6 or 8 Samples per Pixel

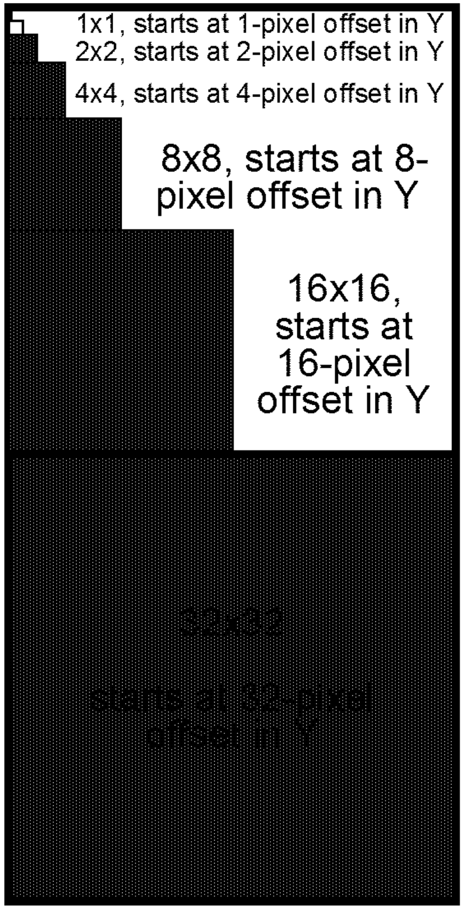

Fragment1 (1)
64 colors for fragment 0, in micro-tile order
unused
Fragment2 (2)
64 colors for fragment 0, in micro-tile order
64 colors for fragment 1, in micro-tile order
unused
Fragment4 (3)
64 colors for fragment 0, in micro-tile order
. . .
64 colors for fragment 3, in micro-tile order
unused
Fragment8 (4)
64 colors for fragment 0, in micro-tile order
. . .
64 colors for fragment 7, in micro-tile order


1x1, starts at 1-pixel offset in Y
2x2, starts at 2-pixel offset in Y
4x4
starts at 4-pixel offset in Y
8x8
Mipmap
starts at 8-pixel offset in Y
341 (33%) in use
683 (67%) unused
16x16 Mipmap, starts at 16-pixel offset in Y


1x1, starts at 1-pixel offset in Y
1x2, starts at 2-pixel offset in Y
2x4, starts at 4-pixel offset in Y
4x8, starts at 8-pixel offset in Y
171 (17%) in use
853 (83%) unused
8x16
starts at 16-pixel offset in Y


1x1, starts at 1-pixel offset in X
2x2, starts at 2-pixel offset in X
4x4, starts at 4-pixel offset in X
8x8
starts at 8-pixel offset in X
16x16
starts at 16-pixel offset in X
683 pixels unused (1/3 of 2048)

683 pixels unused



1x1, starts at 1-pixel offset in Y

2x2, starts at 2-pixel offset in Y

4x4, starts at 4-pixel offset in Y

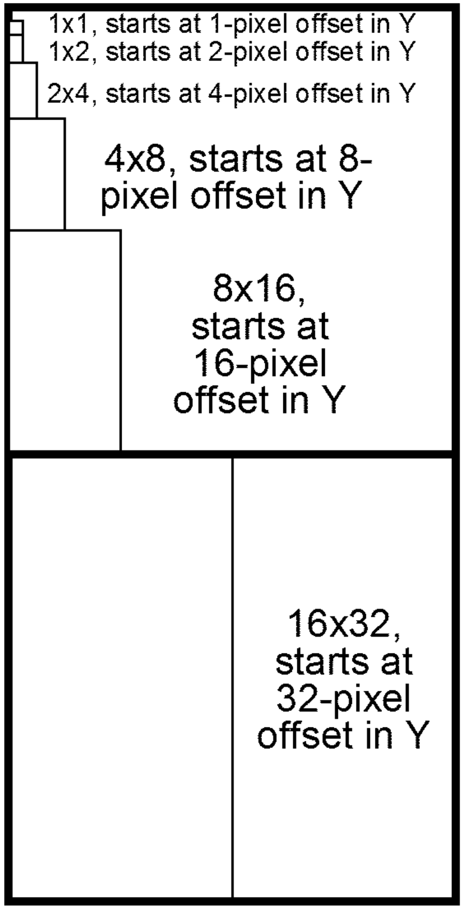
8x8, starts at 8-pixel offset in Y

16x16, starts at 16-pixel offset in Y

32x32

starts at 32-pixel offset in Y

1365 pixels unused



1x1, starts at 1-pixel offset in Y

1x2, starts at 2-pixel offset in Y

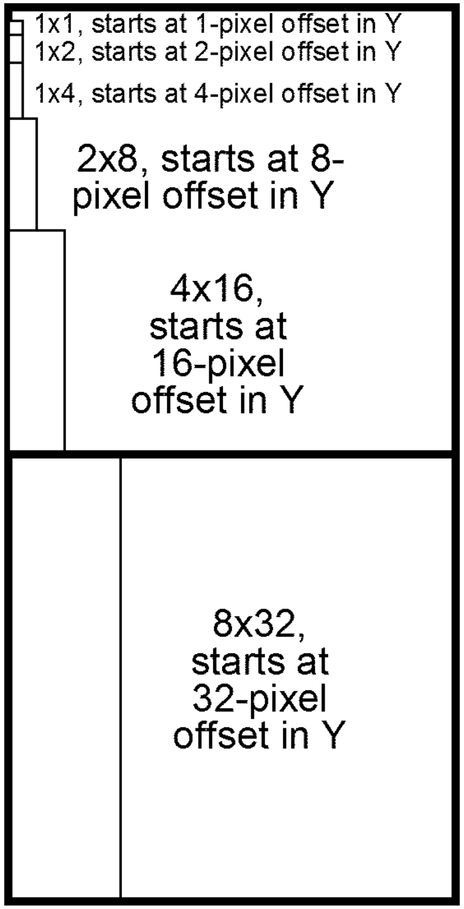
2x4, starts at 4-pixel offset in Y

4x8, starts at 8-pixel offset in Y

8x16, starts at 16-pixel offset in Y

16x32, starts at 32-pixel offset in Y

2046 pixels unused



1x1, starts at 1-pixel offset in Y

1x2, starts at 2-pixel offset in Y

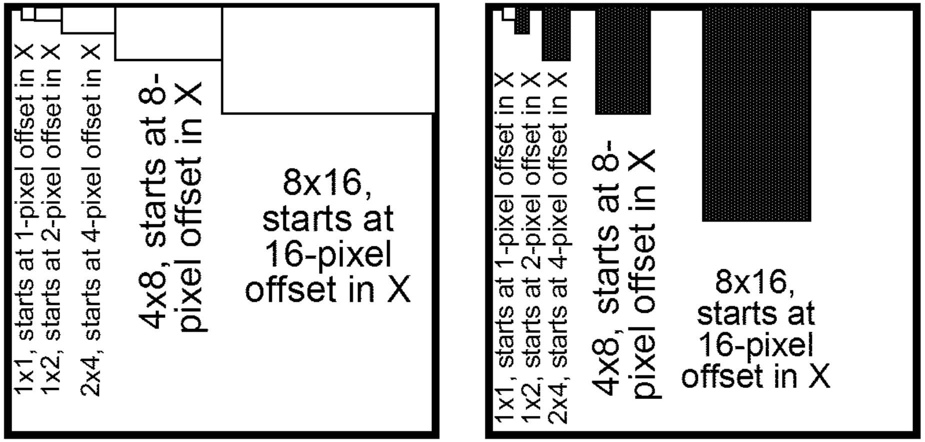
1x4, starts at 4-pixel offset in Y

2x8, starts at 8-pixel offset in Y

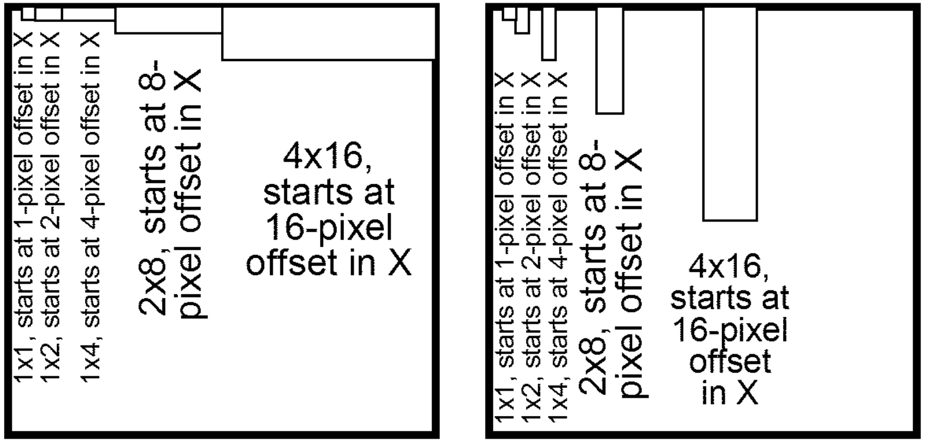
4x16, starts at 16-pixel offset in Y

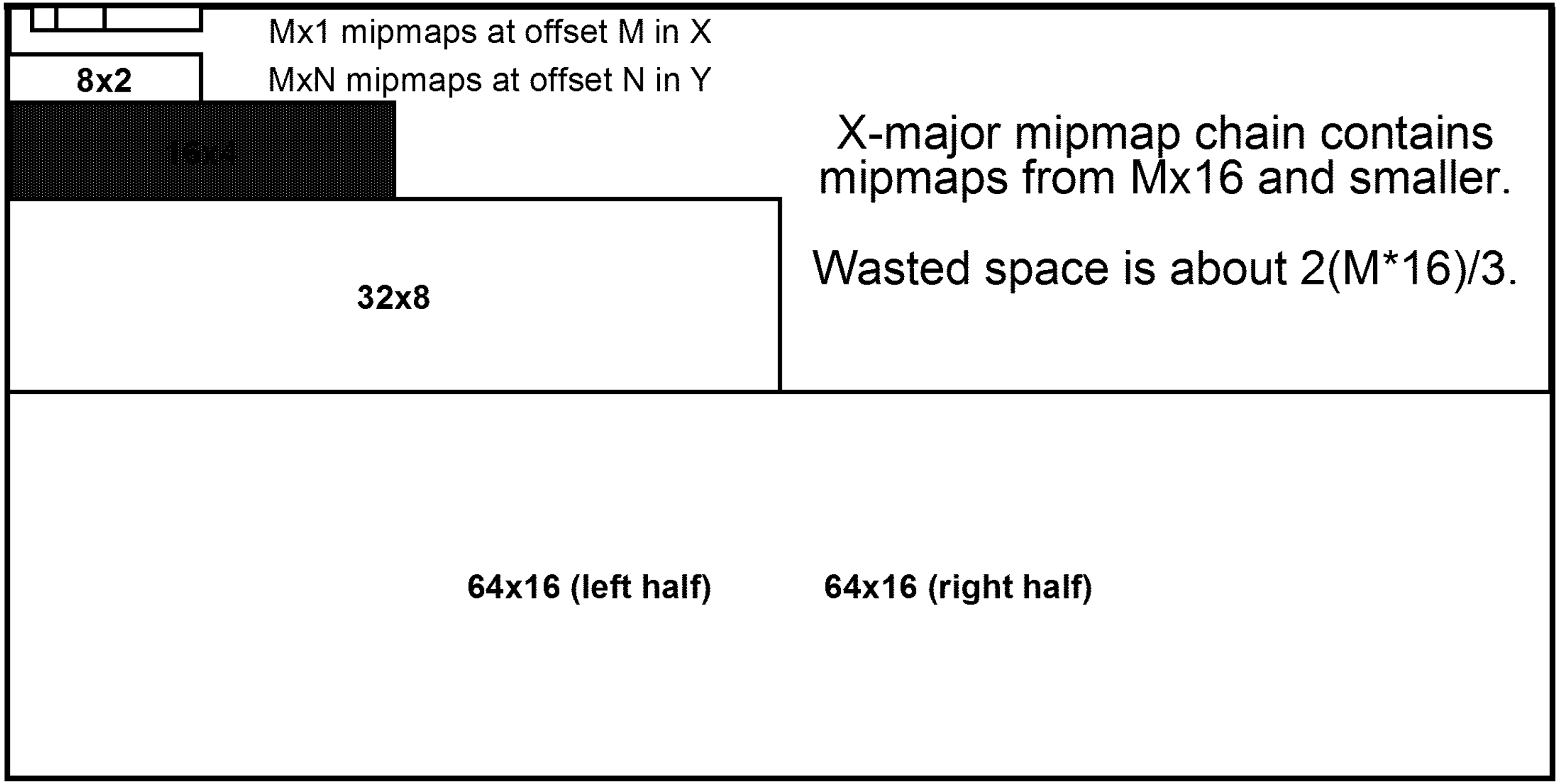
8x32, starts at 32-pixel offset in Y

2:1 Size Ratio (853 pixels unused)


1x1, starts at 1-pixel offset in X
1x2, starts at 2-pixel offset in X
2x4, starts at 4-pixel offset in X
4x8, starts at 8-pixel offset in X
8x16, starts at 16-pixel offset in X
1x1, starts at 1-pixel offset in X
1x2, starts at 2-pixel offset in X
2x4, starts at 4-pixel offset in X
4x8, starts at 8-pixel offset in X
8x16, starts at 16-pixel offset in X

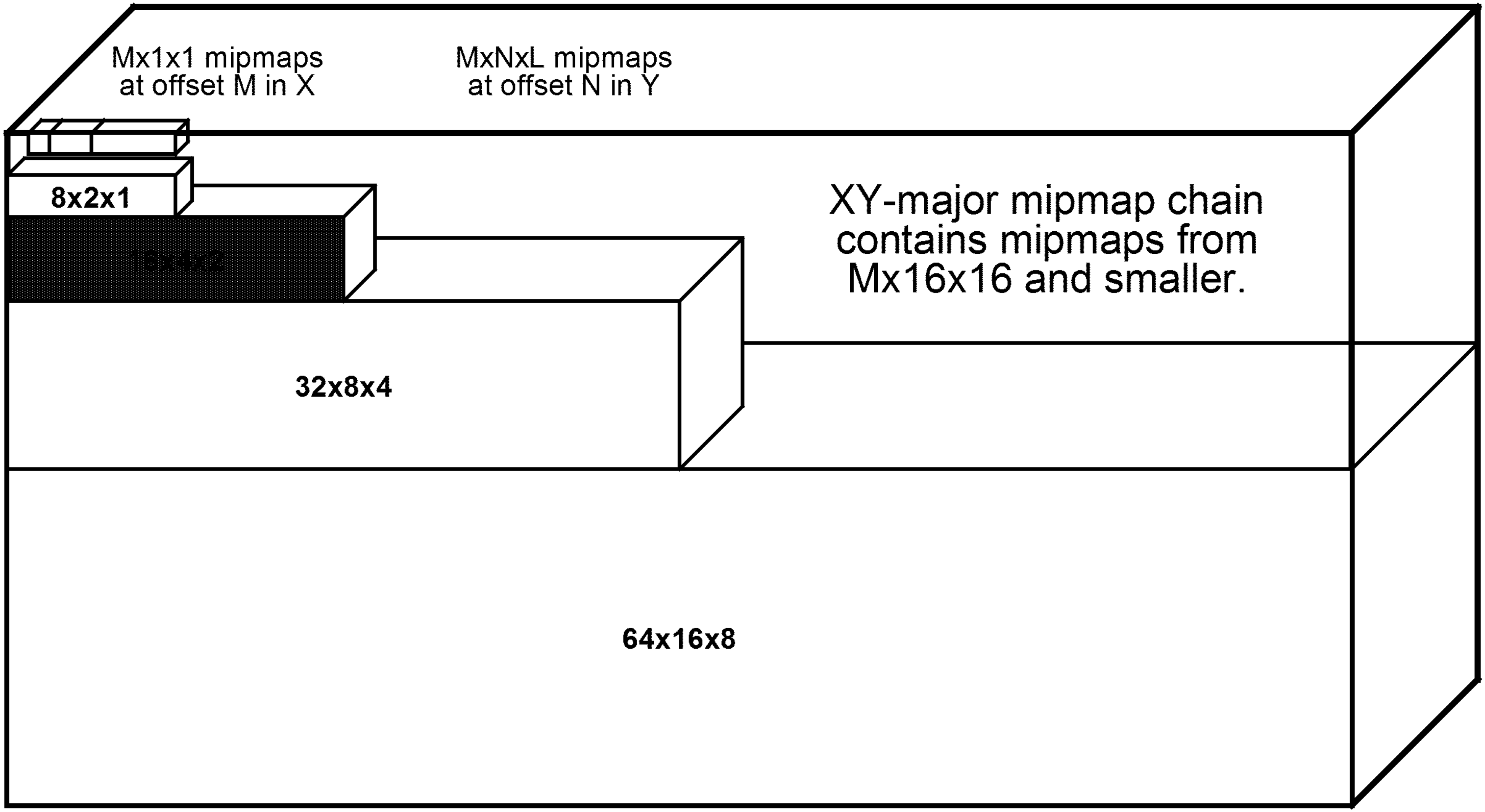
4:1 Size Ratio (937 pixels unused)


1x1, starts at 1-pixel offset in X
1x2, starts at 2-pixel offset in X
1x4, starts at 4-pixel offset in X
2x8, starts at 8-pixel offset in X
4x16, starts at 16-pixel offset in X
1x1, starts at 1-pixel offset in X
1x2, starts at 2-pixel offset in X
1x4, starts at 4-pixel offset in X
2x8, starts at 8-pixel offset in X
4x16, starts at 16-pixel offset in X

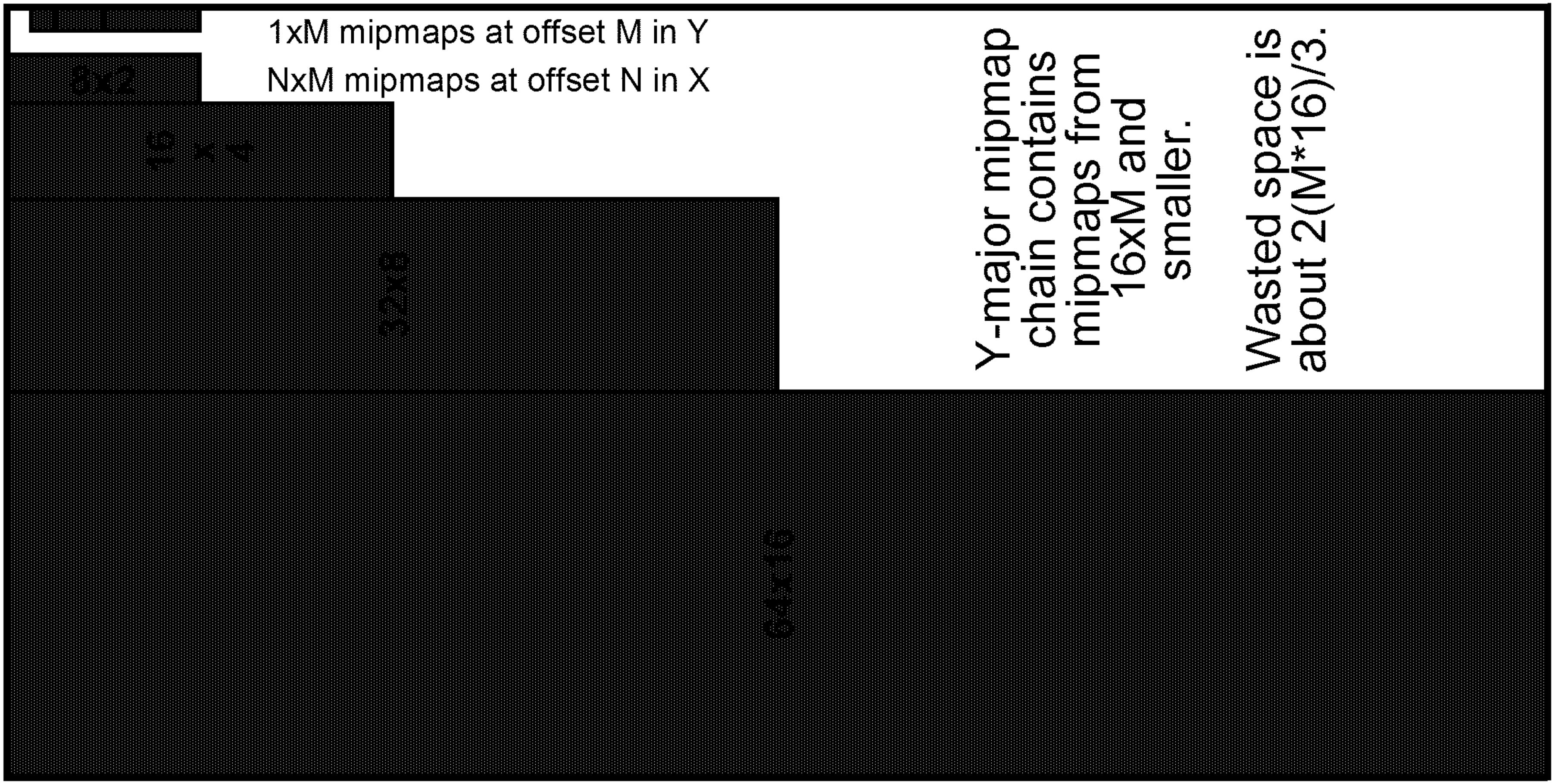


X-major mipmap chain contains mipmaps from Mx16 and smaller.

Wasted space is about 2(M*16)/3.



XY-major mipmap chain contains mipmaps from Mx16x16 and smaller.



1xM mipmaps at offset M in Y

NxM mipmaps at offset N in X

Y-major mipmap chain contains mipmaps from 16xM and smaller.

Wasted space is about 2(M*16)/3.

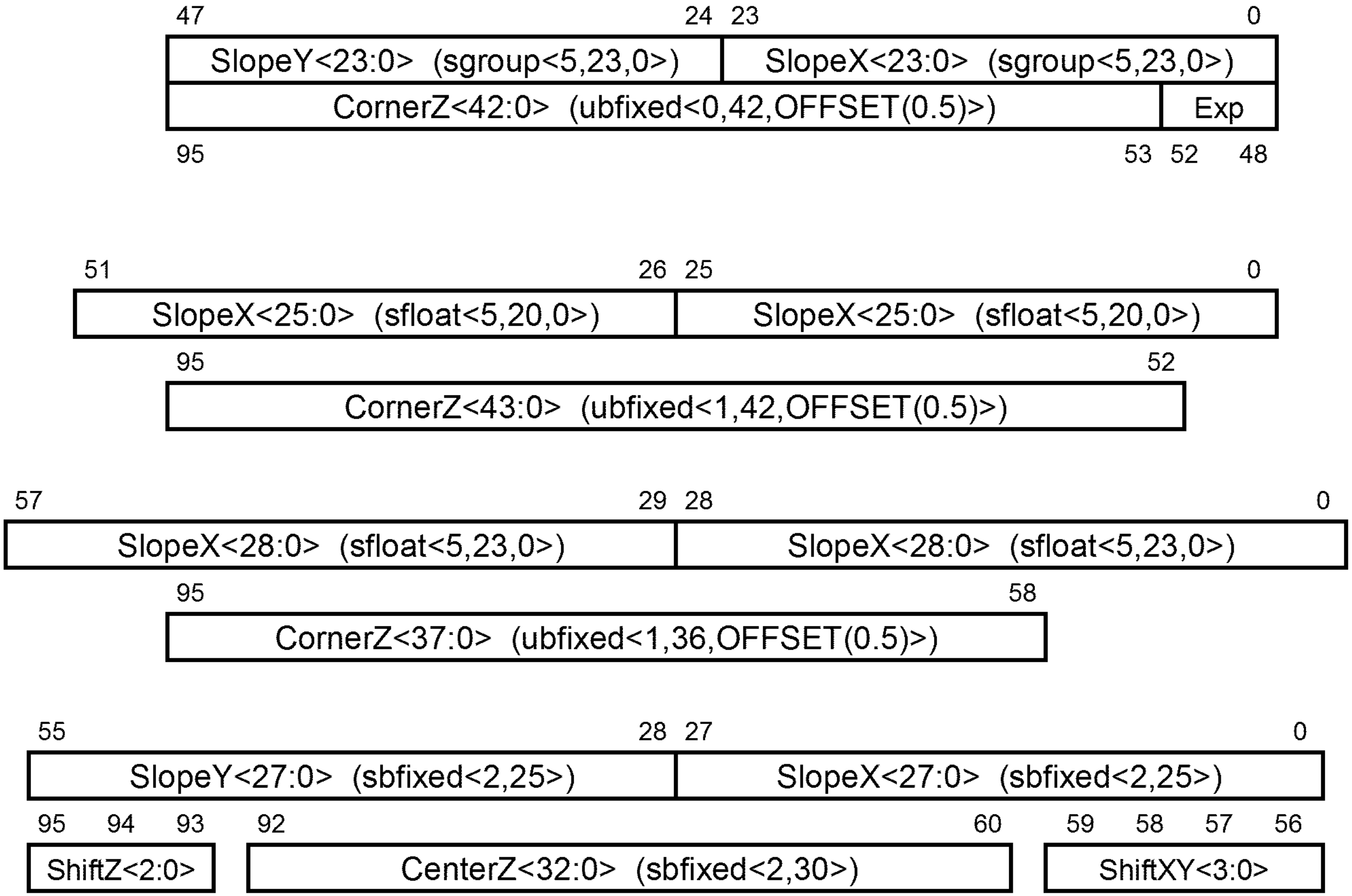
95 53	52 48	47 24	23 0
CornerZ<42:0> (ubfixed<0,42,OFFSET(0.25)>)	Exp	SlopeY<23:0> (sgroup<5,23,0>)	SlopeX<23:0> (sgroup<5,23,0>)

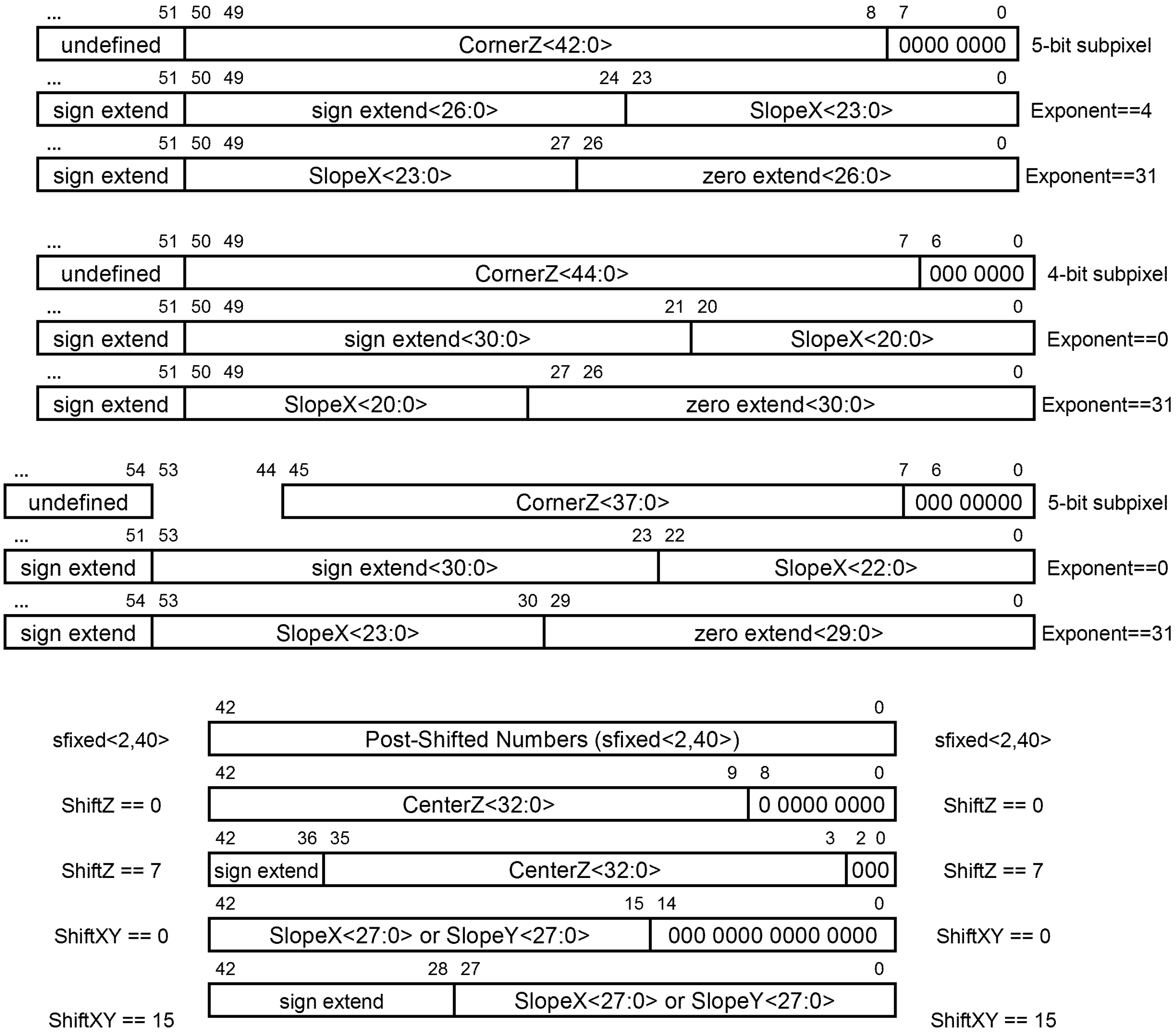
95 58	57 29	28 0
CornerZ<37:0> (ubfixed<1,36,OFFSET(0.25)>)	SlopeX<28:0> (sfloat<5,23,0>)	SlopeX<28:0> (sfloat<5,23,0>)

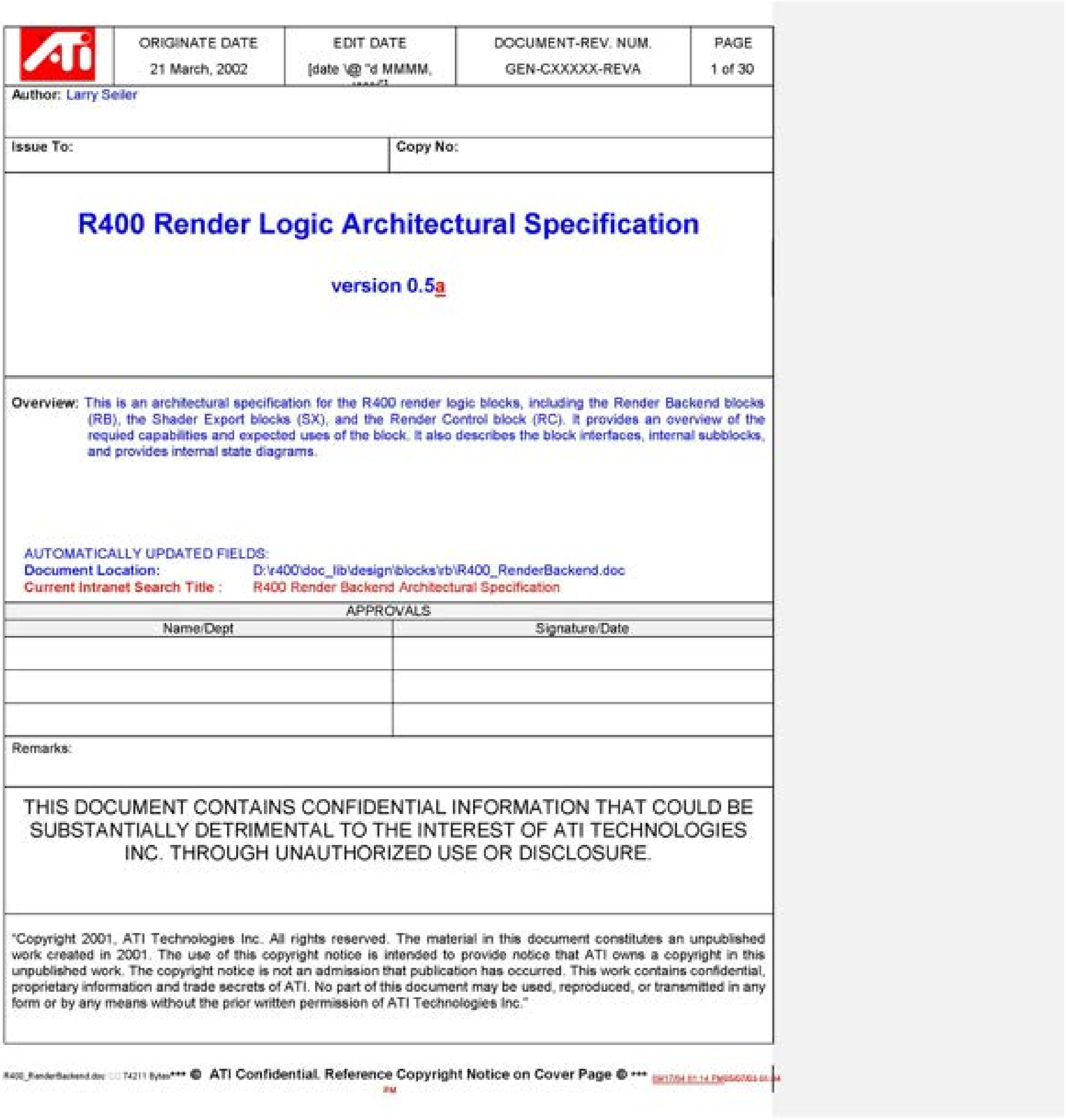
95 60	59 56	55 28	27 0
CornerZ<35:0> (ubfixed<0,42,OFFSET(0.25)>)	Exp	SlopeY<27:0> (sgroup<4,27,0>)	SlopeX<27:0> (sgroup<4,27,0>)

89 45	44 0
SlopeX<44:0> (sfixed<0,44,0>)	SlopeX<44:0> (sfixed<0,44,0>)

127 90
CornerZ<37:0> (ubfixed<0,37,OFFSET(0.25)>)


47
24
23
0
SlopeY<23:0> (sgroup<5,23,0>)
SlopeX<23:0> (sgroup<5,23,0>)
CornerZ<42:0> (ubfixed<0,42,OFFSET(0.5)>)
Exp
95
53
52
48
51
26
25
0
SlopeX<25:0> (sfloat<5,20,0>)
SlopeX<25:0> (sfloat<5,20,0>)
95
52
CornerZ<43:0> (ubfixed<1,42,OFFSET(0.5)>)
57
29
28
0
SlopeX<28:0> (sfloat<5,23,0>)
SlopeX<28:0> (sfloat<5,23,0>)
95
58
CornerZ<37:0> (ubfixed<1,36,OFFSET(0.5)>)
55
28
27
0
SlopeY<27:0> (sbfixed<2,25>)
SlopeX<27:0> (sbfixed<2,25>)
95
94
93
92
60
59
58
57
56
ShiftZ<2:0>
CenterZ<32:0> (sbfixed<2,30>)
ShiftXY<3:0>


... 51 50 49 8 7 0
undefined
CornerZ<42:0>
0000 0000
5-bit subpixel
... 51 50 49 24 23 0
sign extend
sign extend<26:0>
SlopeX<23:0>
Exponent==4
... 51 50 49 27 26 0
sign extend
SlopeX<23:0>
zero extend<26:0>
Exponent==31
... 51 50 49 7 6 0
undefined
CornerZ<44:0>
000 0000
4-bit subpixel
... 51 50 49 21 20 0
sign extend
sign extend<30:0>
SlopeX<20:0>
Exponent==0
... 51 50 49 27 26 0
sign extend
SlopeX<20:0>
zero extend<30:0>
Exponent==31
... 54 53 44 45 7 6 0
undefined
CornerZ<37:0>
000 00000
5-bit subpixel
... 51 53 23 22 0
sign extend
sign extend<30:0>
SlopeX<22:0>
Exponent==0
... 54 53 30 29 0
sign extend
SlopeX<23:0>
zero extend<29:0>
Exponent==31
42 0
sfixed<2,40>
Post-Shifted Numbers (sfixed<2,40>)
sfixed<2,40>
42 9 8 0
ShiftZ == 0
CenterZ<32:0>
0 0000 0000
ShiftZ == 0
42 36 35 3 2 0
ShiftZ == 7
sign extend
CenterZ<32:0>
000
ShiftZ == 7
42 15 14 0
ShiftXY == 0
SlopeX<27:0> or SlopeY<27:0>
000 0000 0000 0000
ShiftXY == 0
42 28 27 0
sign extend
SlopeX<27:0> or SlopeY<27:0>
ShiftXY == 15
ShiftXY == 15



	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	21 March, 2002	[date \@ "d MMMM, yyyy"]	GEN-CXXXXX-REVA	1 of 30

Author: Larry Seiler

Issue To:	Copy No:

R400 Render Logic Architectural Specification

version 0.5a

Overview: This is an architectural specification for the R400 render logic blocks, including the Render Backend blocks (RB), the Shader Export blocks (SX), and the Render Control block (RC). It provides an overview of the requied capabilities and expected uses of the block. It also describes the block interfaces, internal subblocks, and provides internal state diagrams.

AUTOMATICALLY UPDATED FIELDS:
Document Location: D:\r400\doc_lib\design\blocks\rb\R400_RenderBackend.doc
Current Intranet Search Title : R400 Render Backend Architectural Specification

APPROVALS	
Name/Dept	Signature/Date

Remarks:

THIS DOCUMENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATI TECHNOLOGIES INC. THROUGH UNAUTHORIZED USE OR DISCLOSURE.

"Copyright 2001, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished work created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this unpublished work. The copyright notice is not an admission that publication has occurred. This work contains confidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or transmitted in any form or by any means without the prior written permission of ATI Technologies Inc."

Table Of Contents

Table Of Figures

Table Of Tables

Revision Changes:

Rev 0.1 (Larry Seiler) Date: April 6, 2001	First draft
Rev 0.2 (Larry Seiler) Date: June, 2001	Major updates on Parameter Buffer and Tile Logic blocks. Lesser changes to other blocks.
Rev 0.3 (Larry Seiler) Date: November 7, 2001 – December 20, 2001	Changes to reflect the new shader pipe design.
Rev 0.4 (Larry Seiler) Date: January 30, 2002	Major changes in the block overview descriptions, addition of a section on data flows, changes in the SX block description. Detailed RB block descriptions are temporarily removed, since they are out of date.
Rev 0.5 (Larry Seiler) Date: July 8, 2002	SX export buffer no longer stores depth compressed, various elaborations and extended explanations.
Rev 0.5a (Larry Seiler) September 17, 2004	Fixed a bug in the Stencil Function Pass/Fail table

Formatted: Font: Not Bold

Open Issues and Known Deficiencies:

The export buffer and SP->SX interface would be much simpler with a wider data bus.	
Work out details of the shadow buffer paths through the RB.	
Verify that the surface and context synchronization schemes meet our needs	
Work out details of Gaussian filter multi-sample resolve through the RB.	
How does vertex export address generation work – is the plan of record stable?	
Do we need to support sample dropping?	
Should we still support the fog blend?	
What changes need to be made in the SX for handling cached parameters?	
NOTE: bus names in the figures are not up to date in rev 0.4	

1. Overview

The R400 render logic blocks consist of the Render Control block (RC), the Shader Export blocks (SX), and the Render Backend blocks (RB). Together, these blocks accept depths, colors, and coverage masks in order to perform hierarchical tests, depth/stencil tests and destination color blend operations. They input 8x8 tiles and 2x2 quads from the Scan Converter (SC) and Shader Pipe (SP) blocks, read data from the Memory Controllers (MC), and write out the results. The RB blocks also compute shadow buffer coverage and filter anti-aliased pixels in response to commands from the texture pipe (TP). Finally, they pass through other types of export data computed in the Shader Pipes and write it to linear arrays in memory.

The figure below illustrates the render logic blocks and how they relate to other R400 blocks. In the upper half of the figure, the Render Central block (RC) distributes render tile commands to the other blocks, collects hierarchical cull results for the SC, and collects synchronization signals for the CP. The two Shader Export blocks (SX) each buffer the data that is output from a pair of Shader Pipes, as well as performing operations such as alpha test.

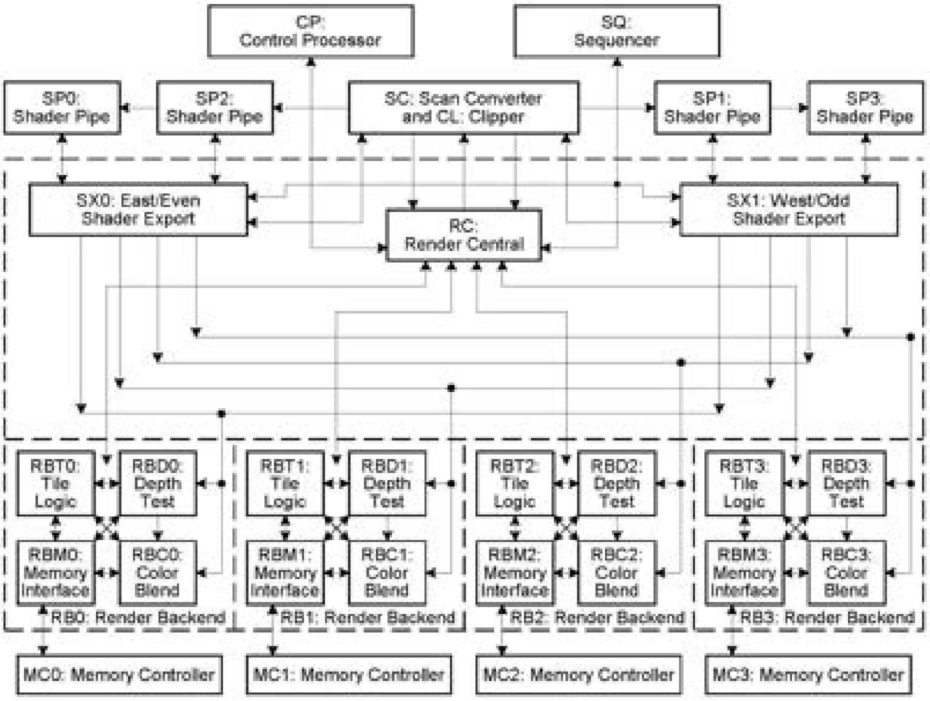


Figure 1: RB Top Level Block Diagram

The lower half of the figure shows the four Render Backend blocks (RB) and their associated Memory Controller blocks (MC). Dashed lines enclose the four sub-blocks that compose each RB. Each RB owns ¼ of the 8x8 tiles in the frame buffer and processes all render commands directed at their portion of the frame buffer. The RC and each SX block have separate busses that route to each of the four RB blocks, since each Shader Pipe (SP) can process pixels that belong to any of the RBs. These are long routing tracks in the physical layout, since the upper three blocks are physically placed near the PA block and the SP blocks, whereas the RB blocks are physically placed near the corresponding Memory Controllers.

The RB processes pixels in three sub-blocks, each of which may require memory accesses, which are managed by the fourth sub-block. The Tile Logic block (RBT) performs hierarchical depth and stencil tests in order to kill quads before they are sent to the Shader Pipe. This block manages a tile data cache that stores hierarchical and compression information about each 8x8 tile. The Depth Test block (RBD) performs depth and stencil tests, as well as shadow buffer comparisons. The Color Blend block (RBC) performs all back end color processing, including multi-sample fragment processing. It manages a fragment mask cache and a color cache. The color cache also stores non-color data that is exported to linear arrays. Finally, the Memory Interface block (RBM) performs data compression/decompression, manages the interface to the Memory Controller (MC) and converts between the data format in memory and the data format in the internal caches, when they differ.

1.1 RC: Render Central Block

The Render Central block (RC) distributes render tile commands from the Scan Converter (SC) to the RB blocks and collects synchronization signals for the CP. This block isolates the SC and CP from the details of how many RB blocks there are and how they divide up the rendering task. The R400 is able to run with one, two, or all four RBs enabled, in order to support smaller memory interfaces. The diagram below illustrates the basic elements of the RC.

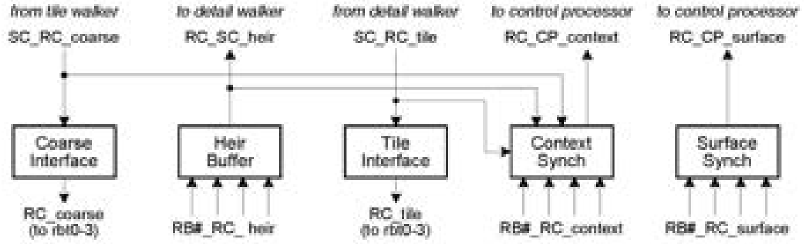


Figure 2: RC: Render Central Block Diagram

First, the SC's coarse walker finds tiles that may need to be rendered and sends them to the RC. The RC forwards this data to the four RB tile logic blocks (RBT), which each perform hierarchical tests on the tiles that they own. The RC then gathers the results of these hierarchical tests, puts them back in the original order and returns the results to the SC for use in the detail walker. The detail walker computes the valid samples inside each quad that passes the hierarchical tests, dropping a tile if none of its quads passed the test. If a quad contains any valid samples, the SC sends a quad command to the SX and SP blocks. Additionally the SC sends the RC a 16-bit mask of which quads are being processed out of each tile that passed the hierarchical tests, even if no quads contain valid samples in that tile. The RC forwards this data to the RB that controls that tile. This allows the RBs to read pixel data while the quads are being processed in the SP. The RBs need to keep track of what quads are in the shader pipe, so they also use this mask to determine which quads were killed by the SC detail walker.

The Context Synch block tells the CP when the render logic is finished processing each render context. Each tile has associated with it a state ID that specifies which render context the render logic should use to process that tile. The CP is free to reuse a context after the render is finished with it, since the render logic cannot finish a context until the SC, SP, and TP are also finished with it. Those blocks do not report when they have completed each context. A context is active in the RB when the SC issues a tile that uses that context. The context is finished when the SC issues a tile with a different context and all of the individual tiles marked with the previous context are finished. A tile is finished when it fails the hierarchical test, when the detail walker finds no valid samples in it, or when an RB reports to the Context Synch block that it has completed that tile. Therefore, the Context Synch block contains a count of the number of outstanding tiles per context. When a count goes to zero, the Context Synch block reports to the CP that the specified context is now free.

Finally, Surface Synch block supports synchronizing accesses to surfaces in frame buffer memory. Each RB receives requests directly from the register interface to synchronize on a specific range of FB addresses. The RB blocks respond to such a request by flushing their caches of all data within that surface. Each RB sub-block reports to the RC when it has finished flushing the surface data. The RC gathers these results and reports to the CP when the surface has been removed from the RB caches.

(NOTE: Add an interface to report pixels that pass the depth test; for the RB_ZPASS_DATA register.)

1.2 SX: Shader Export Block

The Shader Export Blocks (SX0 and SX1) store and process results exported from the shader pipes. There are three broad classes of exports: vertex parameters and positions, which are produced by vertex shaders; colors and depths, which are produced by pixel shaders, and raw data to be written to a linear array in memory. Raw data is produced by multi-pass shaders and by debug exports. The figure below illustrates the busses and basic internal blocks for each SX. SX0 is associated with SP0 and SP2. SX1 is associated with SP1 and SP3. They also communicate with each other and both of them communicate with the SC, SQ and all four RBs.

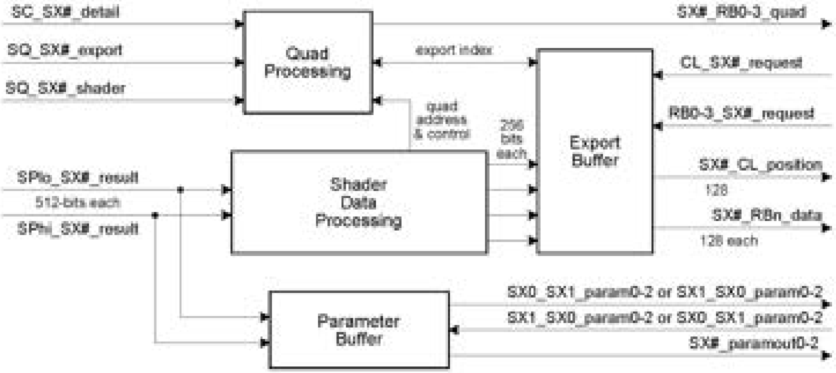


Figure 3: SX: Shader Export Block Diagram

The Parameter Selection sub-block handles vertex parameters. This includes texture coordinates, colors, and all other data specified per-vertex, except for the vertex position. Under control of the SQ, this sub-block reads three parameters from a cache for the three vertices of the current triangle. The three parameters could come from any of the shader pipes and are used by all of them, so the two SX blocks need to exchange data. The Parameter Selection sub-block then pre-processes the parameters and forwards the results to the Shader Pipe interpolators. All four interpolators receive the same parameter data. Accordingly, the two SX blocks replicate some of the multiplexing and pre-processing.

The Shader Data Processing sub-block processes all export data from the SP, except for the vertex parameters. It performs the alpha test and other tests on color data before writing them to the export buffer. It passes most other exports straight through to the Export Buffer.

The Export Buffer stores export data, other than parameters. The CL (Clipper) reads position data from a reserved portion of the buffer. The RBs read all other export data. Data is stored in the Export Buffer in a swizzled form that allows the four RBs to stream out 128-bit data without conflicts. The Export Buffer also reports the space available to the SQ, which waits to schedule a clause until there is room in the Export Buffer.

Finally, the Quad Processing sub-block sends quad commands to the RBs that correspond to the data in the Export Buffer. This sub-block receives quad commands from the SC's detail walker and buffers them until the corresponding color data emerges from the Shader Pipe. It then modifies each pixel's sample mask based on the result of the alpha test and SP pixel kill instructions and appends the location of the pixel data in the export buffer. This sub-block also generates quad commands for export data other than colors and positions. These quad commands include the memory address at which the RB must write the data, as well as the location of the non-pixel data in the export buffer. The memory address replaces the sample mask that is contained in a pixel quad command.

1.3 RBT: Tile Logic Block

The Tile Logic block performs hierarchical depth/stencil tests. It stores Zmask, Smask and Cmask compression codes for all tiles currently active in the RB. It also stores an in-flight count for each quad and tile currently being rendered. The in-flight count tracks the number of operations currently in the pipeline for a quad or tile. The in-flight counts are incremented when the RBT first receives a tile from the Scan Converter (SC) and are decremented when the tile or quad is removed from the RB, either because it is killed or because the RB finishes processing the tile or quad. The figure below illustrates the internal structure of the Tile Logic block.

The upper part of the figure shows the path used for hierarchical tests. The RC_coarse bus sends per-tile information to the Tile Logic block, including a mask of the quads in the tile that may be covered (the coarse mask). The Coarse buffer stores this information for tiles that belong to this particular Tile Logic block. The Tile Test block performs an hierarchical depth and stencil test on the tile, using per-tile hierarchical data in the Tile Cache. The results are buffered and then the Quad Test block performs a 2x2 hierarchical test on any quads that passed the 8x8 tile test, using per-quad hierarchical data in the Quad Cache. Finally, the RB#_RC_hier bus passes a mask of the result per quad back to the Render Central block.

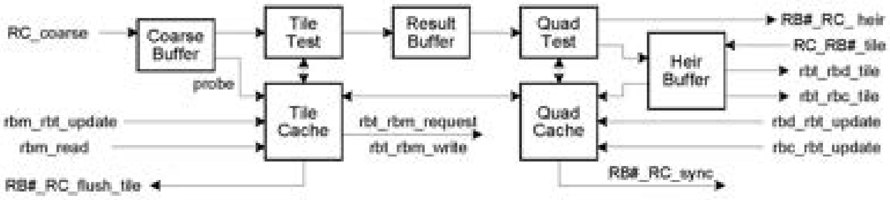


Figure 4: RBT: Tile Logic Block Diagram

The Tile Cache performs several actions in addition to providing per-tile hierarchical data for the Tile Test. First, the Coarse Buffer probes the Tile Cache to see whether each tile is present in the cache. This occurs in parallel with and in advance of the Tile Test, so that the Tile Cache can send a read request to the RB Memory Interface block early enough to cover the read latency. The Tile Cache reads 256-bit values that store data for 8 tiles. It may also read data for adjacent sets of tiles, to increase the chance that tiles are already in the cache when they arrive from the RC. These extra reads don't cost much memory bandwidth, since the Memory Controller requires multiple reads on the same page for efficiency. Finally, the Tile Cache sends tiles that pass the tile HiZ test to the RB Memory Interface block so that depth can be prefetched.

Next, the Tile Cache probes the Quad Cache for each tile that passes the hierarchical tile test. If the Quad Cache doesn't contain an entry for that tile, it allocates one, initializing each quad with the hierarchical data for the tile. This way, the Quad Cache does not need to read data from memory. The Depth Test (RBD) block sends revised stencil and depth ranges to the Quad Cache as it processes quads, so that the quad cache becomes more precise over time. The Quad Cache in turn updates the Tile Cache with more accurate hieararchical data. A quad or a tile can only be fully updated if no operations are pending for that quad or tile. Therefore, in-flight counts must be tested to ensure that the quad cache and tile cache account for quads in-flight that have not yet been processed by the RBD. The rbd_rbt_update bus reports these changes in the depth range, as well as reporting changes in the Zmask and Smask fields for each tile. The rbc_rbt_update bus reports when the Color Blend block is done with each quad, as well as reporting changes in the Cmask for each tile.

Finally, the Heir Buffer receives tile data from the SC detail walker (via the RC) and passes tile data to the RBD and RBC blocks. The data from the SC updates the in-flight counts of the quads in each tile, to account for quads killed by the detail walker. The RBC and RBD receive a mask of the quads being processed in the Shader Pipe. The RBD also receives Zplanes and hierarchical test results for individual quads.

1.4 RBD: Depth Test Block

The Depth Logic block processes 2x2 quads. In the normal mode of operation, the Depth Logic uses a Zplane to specify the depths at each sample in the quad. Alternately, the pixel shader program may compute an explicit depth per pixel. The Depth Logic block performs the depth and stencil tests, modifies the quad sample mask accordingly, updates the depth and stencil cache, and forwards the results to the RB Color Blend block. The figure below summarizes the processing in the Depth Test block.

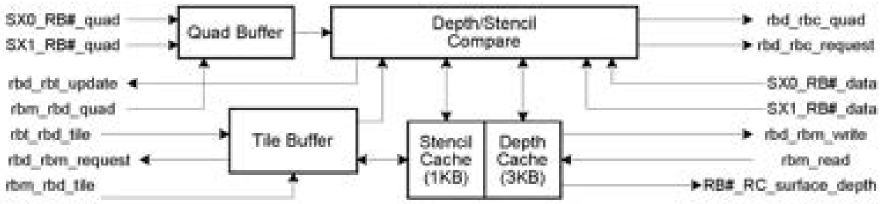


Figure 5: RBD: Depth Test Block Diagram

The Depth Test logic also computes the minimum and maximum Z within each quad that it processes and determines whether the quad contains stencil values that are equal to, greater than, or less than a specified compare value for the surface. The Depth Test logic returns this information to the Tile logic in order to update the hierarchical data for the quad.

Finally, the Depth Test logic performs shadow buffer operations. The RB Memory Interface accepts shadow buffer requests from the Memory Hub/Texture Central and converts them into tile and quad commands. The RBD stores these in separate logical queues so that shadow buffer requests can be completed ahead of regular rendering commands that are already in the queue. This is necessary to avoid deadlocks, since some quad commands currently in the queue may not be able to complete until after the shadow buffer is computed.

1.5 RBC: Color Blend Block

The Color Blend block processes quads of color data. It performs fog interpolation, alpha blending, number format conversions, gamma/degamma conversion, raster-ops, and color keying. It also processes multi-sample pixels that are defined by multiple fragments, where a fragment is a color together with a mask of the samples within a pixel where that color is visible. The Color Blend block also passes along non-blendable data that is being exported to memory.

The figure below shows the basic logic of the Color Blend block. The block receives a quad command from the Depth Test block, which specifies a quad address and a mask of the valid samples. The block looks this up in the Fragment Cache, which stores the mapping of samples to fragments for the currently active pixels. The Fragment Control block produces a sequence of blend commands based on the intersection of the quad sample mask and the fragment masks for those pixels and passes those commands to the Blend Control.

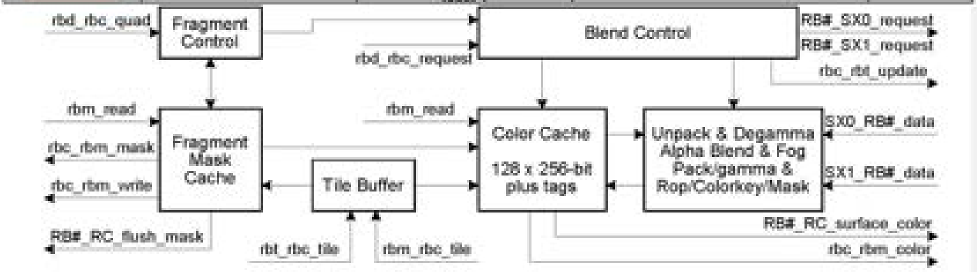


Figure 6: RBC: Color Blend Block Diagram

The Blend Control block reads the color data from the SX export buffer and controls the traditional backend blending functions on it, as well as gamma/degamma conversions and filtering anti-aliased pixels to a single color per pixel. All frame buffer pixels are converted to an internal floating point format for alpha blending, then are converted back. The Blend Control block also synchronizes the quad processing so that a quad command does not read pixel data until any previous write to the same pixel has completed.

The Color Cache contains 64 512-bit lines and has two 256-bit ports that alternate between serving the internal blend logic and the RBM block. This allows the blend logic to process a quad every other clock and allows the RBM to saturate both the memory read bus and memory write bus. Non-blendable data passes through the Blend Control block and into the color cache, where it is written out to memory in the usual way. This allows the RBM to gather this data into multi-bursts for memory efficiency, as it does for ordinary color data. The Color cache also supports flushing cache entries for surface synchronization and for auto-flush.

1.6 RBM: Memory Interface Block

The RB Memory Interface block transfers data between the other three RB sub-blocks and the Memory Controller (MC). The MC uses short access request queues with just-in-time transfers of new read requests and write requests. When the MC processes an access request, the RBM must quickly move a new request into the queue. If possible, the RBM provides data that is likely to be on the same page as the previous requests, so that the MC can perform an efficient series of burst accesses.

{Issues: some data needs to be compressed/decompressed. The RBM also reads commands from the Texture Unit to perform shadow buffer comparisons or to filter anti-aliased pixels. We need to avoid deadlocks, which can arise if the RBM commits a read that cannot complete until some other read completes. E.g., if it commits a depth read, then it must be possible to load that read data into the cache. If the queue fills up, so that the cache won't have room until a color read completes, we could have deadlock. This didn't occur when depth and color reads used separate queues.}

{The following figure needs to be vastly modified.}

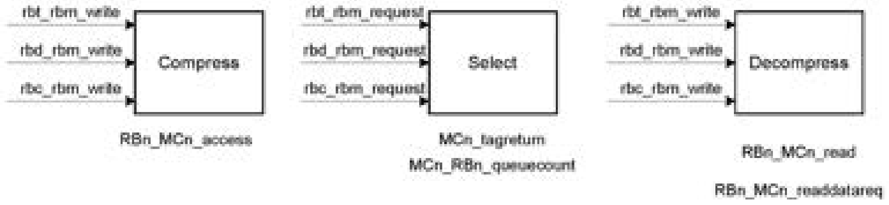


Figure 7: RBM: Memory Interface Block Diagram

The RB Memory Interface block compresses and decompresses tile, depth and color data, if necessary, and issues memory read and write requests to the MC. It also writes shadow buffer comparison results and filtered anti-aliased pixels to the Texture Unit.

2. Render Logic Data Flows

This section describes the sequence of data transfers between the render logic blocks and other R400 blocks. There are three basic types of transfers: pixel render transfers, pass through transfers, and texture unit transfers.

2.1 Pixel Render Transfers

The basic operation of the RB is to render pixels to the frame buffer. The following list describes the data transfers involved in rendering pixels.

1. SC->RC->RBT: The SC tile walker sends the RC an 8x8 tile, together with its address, a Zplane, and a coarse mask of the quads within the tile that might need to be rendered. The RC forwards each tile to all four RB tile logic blocks, each of which buffers just the tiles it owns.
2. RBT<->MC: The RB tile logic blocks read and write per-tile data as necessary in order to compute hierarchical tests on their tiles.
3. RBT->RC->SC: The four RB tile logic blocks return their hierarchical test results to the RC. The RC assembles the hierarchical test results into their original order and forwards them to the SC. It also keeps track of whether an entire context is culled by the hierarchical test (see final step).
4. The following sequence of tile operations occurs, roughly sequentially but with some lookahead:
    a. SC->RC->RBT: The SC sends each tile that was not culled by the hierarchical test to the RC, which forwards it to the RB tile logic. The tile logic buffers this as the list of upcoming tiles to render.
    b. RBT<->MC: The tile logic issues color mask reads and writes to load the mask cache with multi-sample fragment data for the color cache. Reads and writes can occur in parallel due to the MC buffer design.
    c. RBD<->MC: The tile logic issues depth data reads and writes to load the depth cache.
    d. RBC<->MC: The tile logic issues color data reads and writes to load the color cache. For multi-sample pixels, this operation uses the data in the mask cache and so much wait for the mask cache to be valid.
    e. RB->MC: The RB must sometimes clear cache entries in response to synchronization requests or autoflush timeouts.
5. The following sequence of quad operations occurs, in parallel with the above tile operations:
    a. SX->SC: The SX tells the SC how much space is available in the export buffers. A clause cannot begin until there is room to buffer all of the data that will be generated by the clause.
    b. SC->SX: The SC sends quad addresses and sample masks to the SX blocks. The SX buffers them until the shader pipe emits the pixel shader data corresponding to each quad.
    c. SP->SX: The SP sends color and/or depth data to the SX, which stores them in the export buffer. The SX also performs the alpha test and modifies the corresponding quad mask based on that test.
    d. SX->RBD: The SX forwards quads to the RBD so that it can performs the depth test.
    e. RBD<-SX or RBT: The RBD gets depth values for a quad from one of two places. If the shader program computes depths, it reads them from the SX export buffer. Otherwise, it reads a Zplane from the RBT, which received the Zplane from the SC in step 4a.
    f. RBD->RBC: The RB depth test logic forwards the quad to the RB color blend logic.
    g. RBC<-SX: The RB color blend logic reads the pixel shader color data from the SX export buffer, performs the backend color blend operations, and writes the result into the color cache.
6. RBC->RC->CP: When all processing is complete on the last quad of a context, the RB informs the RC. When a context is complete in all four RBs, the RC informs the CP. Note that a context may end early if its quad commands were culled by the hierarchical test.

2.2 Pass Through Transfers

The render logic also supports three types of pass through transfer, in which it transfers data generated by a shader program to its destination without modification. The most common pass through transfer moves position data from the vertex shader to the clipper (CL). The list below describes the steps involved.

1. SP->SX: Each SP vertex shader exports a vector of 16 4-vectors to its SX. Each 4-vector is either an (x,y,z,w) position or a sprite width/height with edge masks. The SX puts this in a reserved portion of the export buffer.
2. CL<-SX: The Clipper requests one position at a time from each SX.
3. SX->SQ: The SX tells the Sequencer when there is room in the buffer for another vector of positions.

The other two pass through transfers move data from a pixel or vertex shader program to memory. The list below describes the steps involved to transfer data from the pixel shader program. As for pixel rendering, each transfer groups together results from the four vertex units that make up one of the pixel shaders.

1. SP->SX: Each SP pixel shader exports a vector of four quads to its SX. Each quad consists of four 4-vectors, e.g. four RGBA colors. The SX puts this data into the export buffer.
2. SX->RB: The SX computes a frame buffer address for each of the four quads and generates four quad commands that contains those addresses. The SX sends each quad command to the appropriate RB.
3. RB<-SX: After processing all previous quad commands, the RB reads the quad from the SX export buffer.
4. RB->MC: The RB writes the quad to memory.

Finally, the following list describes vertex shader program exports. They differ from pixel shader program exports because vertex programs compute vectors of 64 independent vertices, unlike pixel shader programs, which compute vectors of 16 2x2 quads. Therefore, the SX groups together sets of four exports from the same vector unit, instead of grouping together four pixels from adjacent vector units.

1. SP->SX: Each SP vertex shader exports a vector of 16 4-vectors to its SX. The SX gathers together groups of four exports from the same vertex shader clause.
2. SX->RB: The SX computes a frame buffer address for each group of four exports and generates quad commands that contain those addresses. The SX sends each quad command to the appropriate RB.
3. RB<-SX: After processing all previous quad commands, the RB reads the data from the SX export buffer.
4. RB->MC: The RB writes the four 4-vectors to memory.

{Add in detail about caching and transfers within the RB sub-blocks?}

2.3 Texture Unit Transfers

Finally, the RB performs several types of commands for Texel Central (TC). In effect, the RB generates textures on the fly for the TC, based on data in existing color buffers or depth buffers. This section describes the data flows for each of these operations.

The following list describes the steps necessary to filter a multi-sample color buffer and return the data to TC as a texture.

1. TC->MH->MC->RBM: The TC determines that it needs to read a tile of multi-sample color data. The TC sends a command to the RBM that specifies the color surface and the block of pixels within that surface. The TC sends its command over existing datapaths through the MH to the MC that hols the multi-sample color data. The MC routes the request to its assocated RB's Memory Interface block.
2. RBM->RBC: The RBM sends the RBC a tile request. The RBC buffers this in a logically separate queue from the tile data that it receives from the RBT, though it could be a reserved portion of the same physical queue. A separate logical queue is necessary so that the RBC can process the filter request without waiting for normal render requests that are currently in the queue to complete.
3. RBC<->MC: The RBC reads the multi-sample color data and stores it in the color cache.
4. RBM->RBD: In parallel with the preceeding step, the RBM creates a sequence of quad commands that specify filtering the desired pixels. The RBM puts these into a logically separate queue from the quad commands that the RBD receives from the SX blocks, so that the RB can process filter requests ahead of pending render commands.
5. RBD->RBC: The RBD passes quad filter commands to the RBC. The RBM sends them via the RBD so that they are processed in sequence with any shadow buffer commands.
6. RBC->RBM: The RBC computes the filtered pixel values and passes the results directly to the RBM. This data does not go into the color cache since there is no need to either save the data for later or group together multiple writes into an efficient memory access.
7. RBM->MC->MH->TC: The RBM forwards the filtered pixels back to the TC, via its existing datapath to the MC and the existing datapath from the MC to the MH.

A shadow buffer operation compares a Zplane against an existing depth buffer. For each of a group of pixel positions in the depth buffer, it computes what fraction of the Zplane is visible. The TC filters these results to determine whether pixels on a triangle are shadowed. See the R400 Shadows specification for a detailed explanation of the shadow buffer algorithm.

1. TC->MH->MC->RBM: The TC determines that it needs more shadow coverage values. The TC sends a command to the RBM that specifies the Zplane, the depth surface and the block of pixels within that surface. The TC sends its command over existing datapaths through the MH to the MC that hols the multi-sample color data. The MC routes the request to its assocated RB's Memory Interface block.
2. RBM->RBD: The RBM sends the RBD a tile request. The RBD buffers this in a logically separate queue from the tile data that it receives from the RBT, though it could be a reserved portion of the same physical queue. A separate logical queue is necessary so that the RBD can process the filter request without waiting for normal render requests that are currently in the queue to complete.
3. RBD<->MC: The RBD reads the depth data and stores it in the depth cache.
4. RBM->RBD: In parallel with the preceeding step, the RBM creates a sequence of quad commands that specify filtering the desired pixels. The RBM puts these into a logically separate queue from the quad commands that the RBD receives from the SX blocks, so that the RB can process filter requests ahead of pending render commands. This is the same queue that is used for all other texture requests to the RB.
5. RBD->RBM: The RBD computes the shadow coverage values and passes the results directly to the RBM. This data does not go into the depth cache.
6. RBM->MC->MH->TC: The RBM forwards the filtered pixels back to the TC, via its existing datapath to the MC and the existing datapath from the MC to the MH.

Shadow buffering requires computing and storing the shadow buffer at multiple levels of detail, so that the TC can read from a shadow buffer that has approximately the correct pixel spacing. Unlike ordinary texture maps, a lower resolution depth buffer is simply a decimated version of the original depth buffer. The following sequence may be used to compute a decimated depth buffer. Results of this operation are written to the depth buffer that is specified in the RB depth surface descriptor.

1. TC->MH->MC->RBM: The TC receives a command to read the depth surface in decimate mode. It sends a command to the RBM that specifies the depth surface to be decimated. The TC sends its command over existing datapaths through the MH to the MC that hols the multi-sample color data. The MC routes the request to its assocated RB's Memory Interface block.
2. RBM->RBD: The RBM sends the RBD a tile request. The RBD buffers this in a logically separate queue from the tile data that it receives from the RBT, though it could be a reserved portion of the same physical queue. A separate logical queue is necessary so that the RBD can process the filter request without waiting for normal render requests that are currently in the queue to complete.
3. RBD->RBM->MC: The RBD issues a read request for the depth source data, specifying that it is for a decimation command. It also issues a read request for the corresponding tile of the destination depth buffer.
4. MC->RBM: The MC returns the depth data to be decimated, but it is not written into the depth cache. Instead, the depth tile is processed by the RBM.
5. RBM->RBD: The RBM uses the source depth tile to create a series of quad and tile commands that it sends to the RBD quad queue as for the shadow buffer command. If the source depth tile is stored in Zplane format, the RBD sends the Zplanes to the RBD tile queue and sends the RBD quad queue commands to render those Zplanes at a decimated set of sample positions. If the source depth tile is stored as individual depth values, the RMB sends a a decimated subset of the depth values to the RBD tile queue and sends commands to render them to the RBD quad queue.
6. RBD->RBM: The RBD processes the decimation quad commands, which cause it to update the destination depth buffer in the depth cache. Eventually the RBD writes the depth data out of its cache into memory.

(Note: the decimation command could use tile read commands produced by the standard rendering process. It would still produce quad commands from the source depth data, which would replace the quad commands produced by the shader pipe.)

3. SX: Shader Export Block

The Shader Export Blocks (SX0 and SX1) store and process results exported from the shader pipe. There are six classes of exports that the SX processes:

Parameters: produced by vertex shader programs. Three per clock are sent to the shader pipes (SP).
Positions: produced by vertex shader programs and passed serially to the Clipper block (CL).
Colors: produced by pixel shader programs. Up to four different RGBA color buffers are used in the RBC.
Depths: optionally produced by pixel shader programs. If used, this replaces the Zplane in the RBD.
Multipass data: produced by either vertex or pixel shader programs. The RBC stores these to a linear list.
Addresses: each vertex or pixel shader clause can specify an address for the multipass data it outputs.

The figure below illustrates the busses and basic internal blocks for each SX. SX processing can be divided into four broad groups: parameter selection, quad processing, shader data processing, and the export buffer. SX0 inputs data from and outputs data to SP0, SP2, and SX1. SX1 inputs data from and outputs data to SP1 and SP3, and SX0. Both of them communicate with the SC, SQ and all four RBs. In this figure, and through the remainder of this section, SP0 and SP1 are called SPlo and SP2 and SP3 are called SPhi. SX# indicates the specific SX block.

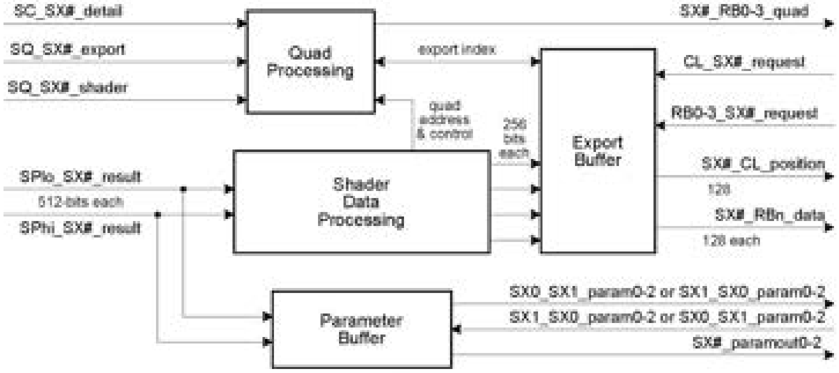


Figure 8: SX: Shader Export Block Diagram

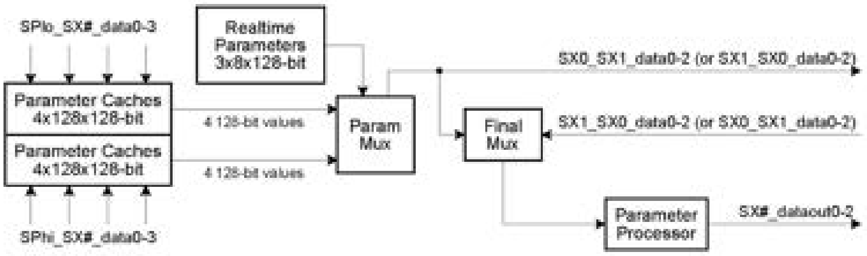
The Shader Export block buffers the remaining types of export data, except for addresses. Buffered positions are passed to the Clipper (CL), buffered depths are passed to one of the four depth test blocks (RBD), and buffered colors and multi-pass data are passed to one of the color blend blocks (RBC). Multipass data is simply routed through the RB to be stored in memory without interpretation, at offsets from a specified address. This data may be used to store results from a high order surface program, a multi-pass vertex program, or a multi-pass pixel program. Each shader pipe must export an address per clause to specify where the RB stores multipass data.

3.1 SX Parameter Buffer Block

The SX parameter selection logic processes vertex parameter data, which the SX receives from its pair of shader pipes on the data busses. The SX caches recently computed parameters from vertex shader programs. In order to execute a pixel shader, each shader pipe needs to select three vertices from the caches, based on the current triangle, and interpolate their parameters. All four SP's receive the same parameters, since they all work on the same triangle at once.

Together, the two SX blocks contain 16 separate parameter caches, each of which stores 128 128-bit parameters. Each vertex can have up to 16 parameters, so this cache stores parameters for 128 or more vertices, depending on the number of parameters per vertex. A single vertex shader program computes parameters for 64 vertices, so the parameter cache can store the results of two or more vertex shader clauses. This cache seems generous in size, but there is a problem. The three vertices that form a triangle may have been computed in any of the vector units and therefore may be in any of the parameter caches. The SPs require three parameters on every clock, so the Sequencer (SQ) must ensure that the three vertices are stored in different caches. The SQ requests a value from three of the 16 caches on each clock. The SX Parameter Block performs the necessary multiplexing.

The figure below shows the internal structure of the SX Parameter Buffer Block. The SQ sends control bits to the SX to indicate when to store the SP result into the vertex buffer. Each SX then reads a parameter from each of its eight parameter caches and selects three of these values to send to the other SX block. Alternately, the SX reads three realtime parameters, which are loaded into a local memory over the register bus. The Final Mux block selects three parameters out of the three from this SX and the three from the other SX. Both SX blocks choose the same three parameters.



SPlo_SX#_data0-3
Realtime Parameters 3x8x128-bit
SX0_SX1_data0-2 (or SX1_SX0_data0-2)
Parameter Caches 4x128x128-bit
4 128-bit values
Param Mux
Final Mux
SX1_SX0_data0-2 (or SX0_SX1_data0-2)
Parameter Caches 4x128x128-bit
4 128-bit values
Parameter Processor
SX#_dataout0-2
SPhi_SX#_data0-3

Figure 9: SX: Parameter Selection Logic

The Parameter Processor performs three operations on the parameters before forwarding them to the shader pipes. First, it selects the provoking vertex parameter value for flat shading (performs parameter selection). Second, it calculates the delta differences between the parameter values at vertices (Parameter Difference). These deltas are then used as input values into the interpolator units. Finally, it performs a cylindrical wrap adjustment on the delta parameters (Cylindrical Wrap). All the functions are implemented on per channel basis. {The control/state signals are part of the "SQ_SP: Interpolator Bus" interface.}

{Note: We could insert a global block between the two SX blocks that does the final mux and forwards the results to the four shader blocks. This doesn't increase the number of global wires, though it may increase the wire length. It eliminates one of the two copies of the Final Mux block, but much more importantly, it provides a single location for inserting the realtime parameters and for including logib to perform several operations on the three parameters that otherwise would have to be performed in all four shader pipes.}

3.2 SX Shader Data Processing Block

On each clock, the Shader Data Processing block inputs four 128-bit RGBA values from each of two Shader Pipes. If they are export values (other than parameter exports), this sub-block processes the data and passes it on to be stored in the Export Buffer sub-block. It also extracts information from the export data and sends it to the Quad Control sub-block. The operation performed depends on the type of export data. The figure below shows the internal structure of the Shader Data Processing Block. The following text describes the processing that occurs for each type of export, except for parameter exports, which are described above.

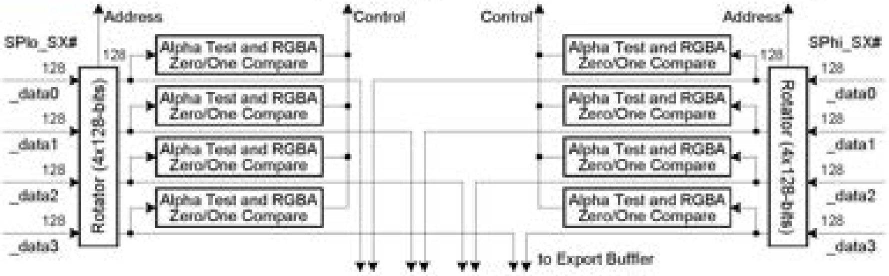


Figure 10: SX: Shader Data Logic

A pixel shader may export one or more RGBA color vectors for rendering to the destination color buffer(s). The Alpha Test and Zero/One Compare blocks operate on this RGBA data. These blocks compare all four components to zero and one and also compare alpha to the Alpha Test value. The Quad Control sub-block uses the results of these comparisons to cull quads. Alpha test culling is defined in OpenGL and DirectX. The zero/one comparisons allow optimizations for blend modes that become no-ops if color or alpha components are equal to zero or one. (cite the register fields that control how this data is used.) The four 128-bit vectors that come from each SP are part of a single 2x2 quad. Four susbsequent clocks send four different quads.

There are two kinds of export data that pass through the Shader Data Block unchanged. The first type is depth data that may be exported from a pixel shader for use in rendering the pixels. As for rendered colors, the four vectors that come from a single SP on a single clock are part of the same 2x2 quad. The second type is position data, which vertex shaders output to specify the (X, Y, Z, W) position of a vertex, plus optionally a sprite width, sprite height, and edge flags.

Multi-pass vertex shaders may export vectors to memory, instead of to the parameter cache. In that case, the vertex shader must specify the address at which to write the vectors. This block forwards these address exports directly to the Quad Processing block, where they select the RB that writes the vectors to memory.

For all remaining export data, the data output from the four vector units of a shader are not related to each other and may be processed by different RBs. Therefore the SX block associates groups of four exports from the same vector unit for these exports. The rotator block rotates the vectors so that the export buffer can group them by vector unit. The following section on the Export Buffer Data Format describes this in detail.

Finally, note that the Alpha Test/Compare blocks only operate on unrotated vectors. Therefore, the Rotator can be placed either before or after the Alpha Test/Compare blocks, depending on implementation constraints.

(**Note**: The Shader Pipe can send up to four color values per pixel. Under DX9, alpha test only applies to the primary color. We may need to support something more general for DX10.)

3.3 SX Quad Processing Block

The SX Quad Processing Block is illustrated in the figure below. The Detail Buffer stores quad commands from the SC until the pixel shader program produces the corresponding color data. The size of the Detail Buffer (over both SX blocks) determines the maximum number of quads that can be in flight in the SP at the same time. The Quad Control and Quad Buffer blocks are described below.

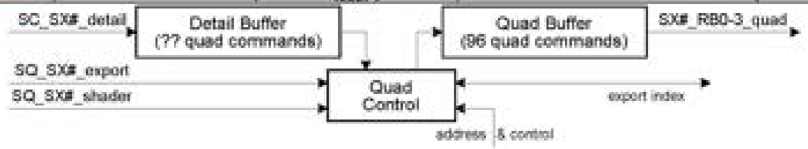


Figure 11: SX: Quad Processing Logic

The Quad Control block allocates data in the export buffer, tells the Sequencer (SQ) how much space is available in the export buffer, and puts quad commands into the Quad Buffer. A quad command specifies a quad for one of the RBs to process, including the sample mask, the quad's position within its tile, and the location of associated data in the export buffer. The Quad Control block operates on three types of data: positions, pixels, and pass-through data.

When the Quad Control block receives position data, it stores it in a reserved portion of the export buffer. Position data consists of either one or two 128-bit words per vertex, depending on the context. In addition to the (x,y,z,w) location of the vertex, the position can include a sprite width, sprite height, and edge flags. The export buffer can store four single-size or two double-size position data results, so that the export buffer can shift out one set of positions while a second set are being computed. A vertex shader that exports position data cannot start until there is room in the export buffer.

When the Sequencer tells the Quad Control block to start a pixel export clause, the Quad Control block allocates space in the Export Buffer for one to four colors per pixel and an optional depth value, depending on the context. It also allocates a space in the Quad Buffer. The Quad Control block then computes an address in the export buffer for each export, depending on the export instruction. The Quad Control block also removes the next quad command from the Detail Buffer, modifies the sample mask based on the comparisons performed by the Shader Data block, and writes the result into the Quad Buffer, together with the location of the pixel data in the export buffer.

The Quad Control block handles clauses that export pass-through data somewhat differently. As before, it allocates space for the export data in the export buffer, but this data does not have corresponding quad commands in the Detail Buffer. Instead, the Quad Control block allocates space in the Quad Buffer and generates quad commands for the pass-through data. Each such command specifies up to four exports from a single vertex and contains a device address in place of the sample mask. The Quad Control block uses a base address context register to generate these device addresses. For vertex exports, it generates the device address by adding the base address to a special address export, which is described in the preceeding section. For all other exports, it generates the device address by incrementing from the base address.

Finally, the Quad Buffer outputs the quad commands to the four RBs. Each RB gets one quad command every four clocks from each SX block, so the Quad Buffer only needs a single read port. {NOTE: There are two basic ways to distribute the quad commands. First, all quad commands could be sent to all RBs, with each RB taking the ones that it owns. This requires buffering for at least 32 quad commands per RB, to avoid stalling when one RB gets two full tiles before another RB gets any. Second, the Quad Buffer could send a separate stream of quad commands to each RB. This requires tracking the next quad command for each RB. Possibly this can tie into the means of figuring out how much free space exists in the export buffer.}

3.4 SX Export Buffer Logic

Finally, the Export Buffer block stores all export data, except for parameters, in eight 80x128-bit buffer memories, which are organized into four buffer pairs. Each buffer stores 16 words of position data and 64 words of other exports. Across both SX blocks, this is sufficient to buffer two vectors of 64 position data (XYZW and sprite size per position) plus 16 8x8 tiles of pixels (with one RGBA color per pixel).

Four of the eight buffers may be read per clock to send results to the RB blocks or the CL block (clipper), using the multiplexers in the figure below. The data must be swizzled so that the five output multiplexers each get one 128-bit word from each of four buffers. The precise storage formats are described in the following subsection. Each output multiplexor reads from buffer pair zero to three sequentially. The four RB output multiplexers are offset so that each occupies a non-competing time slice. The Clipper (CL) output multiplexor steals a time slice from one of the RBs whenever the Clipper needs more position data. The Buffer Read Control block accepts read requests from the CL and the four RBs and schedules whether the CL or an RB gets to use each time slice.

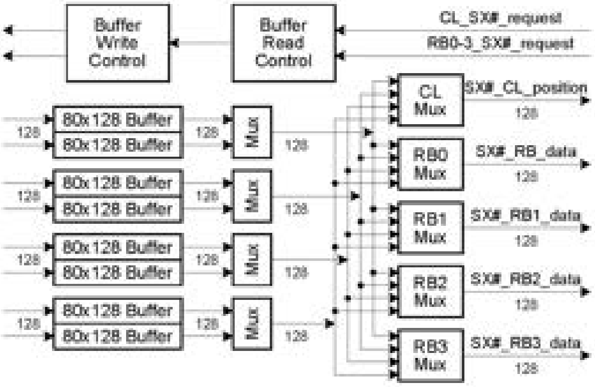


Figure 12: SX: Export Buffer Logic

The SX keeps track of free space in the buffers by storing a bit per access unit, which it sets when it loads data and clears when the RB accesses that data. This requires 64*2 bits per SX, since the RBs read four 128-bit words at a time and each SX has 64*8 128-bit words for non-position exports. The RB is able to skip reading data that it doesn't need by setting a Discard bit when it sends the SX a buffer address.

(**Note**: at one time the export data needed to be maskable by 32-bit words, since the shader instructions could specify which of the four components to write on an export command. This feature has been removed, so that each shader always exports four 32-bit components to the SX. However, the RB may be programmed to ignore some of the components.)

3.5 Export Buffer Data Format

The Export Buffer stores data as vectors of four 32-bit IEEE floating point values, with one exception that is described below. The data format in the export buffer is designed to allow writing shader export data into it at full speed and to allow the four RBs to read data out without interfering with each other. The exact format depends on whether the data is a pixel shader export or a vertex/debug export.

The figure below shows how pixel export data is mapped to the four pairs of export buffer memories. Each SP outputs four 128-bit vectors in parallel on each of four sequential clocks. A single 256-bit wide export buffer memory stores vectors from the two SPs that are associated with the SX block. SX0 receives data from SP0 and SP2 (the even SPs) and SX1 receives data from SP1 and SP3 (the odd SPs). All of the export formats store the corresponding data from the two SPs in the same buffer memory. These figures mark the shaders as SPlo and SPhi, which are SP0 and SP2 for SX0 or SP1 and SP3 for SX1. A single quad can store up to 5 contiguous vectors for one to four colors, followed by an optional vector containing a depth value.

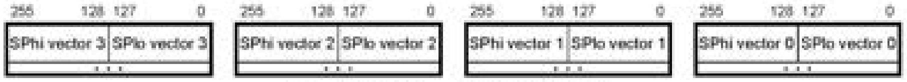


Figure 13: SX: Export Buffer Data Order

(Include another figure that shows how multiple colors and depth are stored. All of the exports for a given quad are stored at sequential locations, before the exports for the next quad. For R400, depth occupies a full 128-bits per pixel, even though only one component is used. If depth is exported, it occupies the first location, followed by 1-4 color buffers.)

The figure below shows how vertex and debug exports map to the four memory buffers. Vertex and debug exports are rotated depending on the export address. Given export address N, (N mod 4) determines the rotation. This is necessary since the four vectors that a single SP outputs in parallel are not related to each other for vertex exports, unlike pixel exports where they are the four pixels of a 2x2 quad. So for vertex exports, the SX groups together four exports from a single vector unit. The figure shows that each row consists of four values with the same vector number, at different values of N mod 4. This allows the RBs to read out either SPhi or SPlo from a single row to get 512-bits of contiguous data.

255 128	127 0
SPhi vector 0 N mod 4 = 3	SPlo vector 0 N mod 4 = 3
SPhi vector 1 N mod 4 = 0	SPlo vector 1 N mod 4 = 0
SPhi vector 2 N mod 4 = 1	SPlo vector 2 N mod 4 = 1
SPhi vector 3 N mod 4 = 2	SPlo vector 3 N mod 4 = 2
. . .	

255 128	127 0
SPhi vector 0 N mod 4 = 2	SPlo vector 0 N mod 4 = 2
SPhi vector 1 N mod 4 = 3	SPlo vector 1 N mod 4 = 3
SPhi vector 2 N mod 4 = 0	SPlo vector 2 N mod 4 = 0
SPhi vector 3 N mod 4 = 1	SPlo vector 3 N mod 4 = 1
. . .	

255 128	127 0
SPhi vector 0 N mod 4 = 1	SPlo vector 0 N mod 4 = 1
SPhi vector 1 N mod 4 = 2	SPlo vector 1 N mod 4 = 2
SPhi vector 2 N mod 4 = 3	SPlo vector 2 N mod 4 = 3
SPhi vector 3 N mod 4 = 0	SPlo vector 3 N mod 4 = 0
. . .	

255 128	127 0
SPhi vector 0 N mod 4 = 0	SPlo vector 0 N mod 4 = 0
SPhi vector 1 N mod 4 = 1	SPlo vector 1 N mod 4 = 1
SPhi vector 2 N mod 4 = 2	SPlo vector 2 N mod 4 = 2
SPhi vector 3 N mod 4 = 3	SPlo vector 3 N mod 4 = 3
. . .	

Figure 14: SX: Export Buffer Data Order

Vertex shaders that export 1-4 vectors use the above format. Vertex shaders that export 5-8 vectors use two copies of the above format, one immediately following the other. Vertex shaders that export 9-12 vectors use three copies of the above format. Therefore, each vertex shader clause requires export buffer space for 0, 4, 8, or 12 exports. (Note: what about unaligned exports?)

The figure below shows the three data formats used in the export buffer. Normally, each 128-bit vector represents four 32-bit IEEE floating point values, which the RB may pack to a smaller format. The components are specified in little-endian order, with X or Red in the low order position and W or Alpha in the high order position. The SP may perform multiple exports to the same export address, using the word select to write different words.

127 96	95 64	63 32	31 0
W or Alpha (32-bit IEEE)	Z or Blue (32-bit IEEE)	Y or Green (32-bit IEEE)	X or Red (32-bit IEEE)

127 96	95 64	63 32	31 0
Depth (32-bit IEEE)	*unused*	*unused*	*unused*

127 102	101 96	95 6	5 64	63 38	37 32	31 6	5 0
Alpha (top 26-bits of 32-bit IEEE)	Fog 31:26	Blue (top 26-bits of 32-bit IEEE)	Fog 25:20	Green (top 26-bits of 32-bit IEEE)	Fog 19:14	Red (top 26-bits of 32-bit IEEE)	Fog 13:8

Figure 15: SX: Export Buffer Data Formats

The final portion of the above figure shows the format used to store an RGBA color that also includes a fog factor. The RGBA values come from the vector pipe and the fog factor comes from the scalar pipe. The SP drops the low order 6-bits of the RGBA components and the low order 8-bits of the scalar result, which is the fog factor. It then packs the 24-bit fog factor into the holes in the RGBA values. The RB rounds these truncated values to produce RGBA and a fog factor in its internal format.

4. RBT: Tile Logic Block

The Tile Logic Block processes 8x8 tiles from the Scan Converter. It can start a new 8x8 tile every four clocks. It loads the Tile cache with per-tile data, including hierarchical Z/stencil data and the depth and color compression modes. It uses this data and data from the Scan Converter to decide when to kill quads and tiles that fail the hierarchical tests. It also buffers data for use by the Depth Test block and allows the Depth Test and Color Blend blocks to update the data cached in the Tile Logic block. The Tile Logic Block ensures that tile data remains in the cache until the Depth Test and Color Blend blocks are done with it, then it writes out the tile data if it was modified.

The figure below illustrates the internal structure of the Tile Logic Block. The Tile Logic Block processes 8x8 tiles in two stages. Each stage steps through the list of 8x8 tiles provided by the Scan Converter. Stage one probes the Tile data cache for each tile and issues a read if that tile's data is not in the cache and does not have a read pending. It then steps to the next 8x8 tile without waiting for the read data to come back. Stage two waits until data is available in the Tile cache for each tile. Then it computes whether the 8x8 tile can be culled and passes that result to the Rasterizer. If the tile is not culled, stage 2 also passes the tile address to the Depth Test block, so that it can read the depth data.

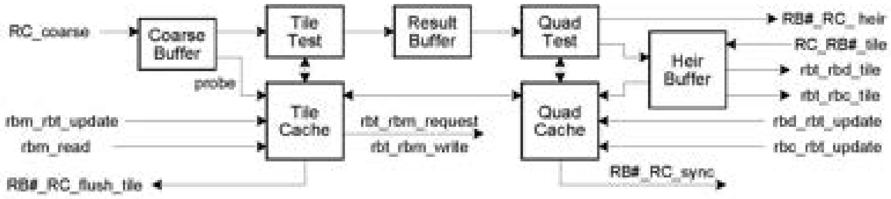


Figure 16: RBT: Tile Logic Block Diagram

The Tile Logic block also processes shadow depth buffer and multi-sample filter requests from the Texture Unit. Each such request to the Tile Logic consists of an 8x8 tile request that specifies which 8x8 tile of which shadow depth buffer to test or which 8x8 tile of which color buffer to filter. Shadow depth buffer and multi-sample filter requests use their own buffer memory so that they do not mix with 8x8 tile requests from the Scan Converter. They have priority for processing by the Tile Probe and Tile Cull stages.

Finally, the Tile Logic block also processes 8x8 tiles for multi-pass pixel processing. These requests come from the RB Parameter Buffer block. The Rasterizer does not send 8x8 tiles during multi-pass pixel processing, so these requests use the same queues that the Scan Converter uses. {**Note**: check whether this works.}

{NOTE: define in-flight counts and explain when they are incremented and decremented.}

4.1 Input Queues

The Tile Logic Block uses two physical input queues, which are each divided into two logical input queues. Each is implemented using a single port memory that operates on a four-clock cycle. In each queue, two of the four clocks are dedicated to writing data into the queues. The other two clocks are used to read data for use in one of the three Tile Logic stages. {**Note**: we could make them two-port memories to allow doubling the rate at which they can process 8x8 tiles. This is not necessary in the 4-pipeline system. Is it useful enough to justify the cost in the 2-pipeline system?}

The SC computes a pair of 8x8 tiles every other clock. These two 8x8 tiles are horizontally adjacent and are therefore always processed in separate pipelines. Therefore, the RB only needs to accept one 8x8 tile every other clock, at maximum. The fifo input logic gathers the dual block data, which is sent over two clocks, and selects which of the two blocks, if any, to write into the fifos. If the second block is selected, the fifo input logic also decodes the compressed dual-block format to find the second block. This requires nothing more than incrementing or decrementing the values specified for the first block.

The following lists describe the signals in the address queue and the depth queue. Each of the queues is used in two stages. The Tile Logic block reads the address queue during stages 1 and 2 and also reads the depth queue during stage 2. The Depth Test block reads the depth queue during stage 3.

Address Queue (stages 1 and 2, 24-bits):

Xaddr (10-bit): Specifies the X position of this 8x8 tile (1&2)
Yaddr (10-bit): Specifies the Y position of this 8x8 tile (1&2)
Context (3-bit): specifies which set of context parameters to use (1&2)
Covered (1-bit): specifies that the 8x8 tile is fully covered {note: may not be implemented} (2)

The address queue stores the (X, Y) address of the 8x8 tile. It also stores a 3-bit code that specifies the set of contect registers to use to process this 8x8 tile. Finally, it may also store a bit that specifies that the entire 8x8 tile is covered by a single triangle. {**Note**: this optimization probably won't be supported. It would allow the RB to skip reading the depth buffer if all depth tests pass, for example.}

Depth Queue (stages 2 and 3, 128-bits)
Zplane (88-bit): specifies the Zplane for this triangle in this 8x8 tile (2?, 3)
Context (3-bit): specifies which set of context parameters to use (3)
Direction (1-bit): specifies that the primitive is front or back facing (2)
PminDepth (10-bit): upper 10 bits of the minimum Z computed at the primitive vertices (2)
PmaxDepth (10-bit): upper 10 bits of the maximum Z computed at the primitive vertices (2)
CoarseMask (16-bit): specifies the 2x2 stamps that may contain valid samples (2 & 3?)

The depth queue stores information used in the Tile Logic Block in stage 2 and in the Depth Test block in stage 3. The Zplane allows the Depth Test logic to compute a depth value per sample. {**Note**: it could also be used in stage 2 to optimize the tile kill, if this proves worthwhile}. Zmin and Zmax specify the minimum and maximum Z values in the triangle, expressed as the upper 10-bits of the values in frame buffer format. Context is the same as the corresponding field in the address queue, and is repeated here to be available to stage 3. Direction distinguishes front-facing from back-facing primitives. Some operations differ depending on the triangle direction. Finally, CoarseMask provides a bit for each 2x2 quad in the 8x8 tile. If the bit is one, the corresponding quad may be required to process the tile. The Depth Test logic may use this to optimize reads for uncompressed depth and color data. {**Note**: the Tile Logic block could also use CoarseMask to optimize till killing.}

4.2 Stage 1: Issue Tile data Read

The first stage processes the 8x8 tiles passed by the SC. This stage probes the Tile Data cache and issues a read to load the cache, if necessary. Each line in the Tile Data cache stores information for sixteen 8x8 tiles. Each tile uses a 32-bit tile data word as illustrated in the figure below.

31 12	11 8	7 6 4	3 2 0
Zrange	Smask	Zmask	Cmask

Figure 17: RBT: Tile Logic Data Format

Cmask and Zmask specify the compression format for the color and depth buffers for each 8x8 tile. Zrange is used for hierarchical depth testing. Smask stores 3-bits of hierarchical stencil information and a bit for stencil compression. The Memory Format Specification describes the color, depth and stencil compression formats. Later sections of this document describe their formats as stored in the RB caches. The Depth Blend and Color Blend blocks update their fields when they modify color and depth/stencil data in their caches.

StencilCode, TminDepth and TmaxDepth describe the range of stencil and depth values in the 8x8 tile. These are used for hierarchical culling of 8x8 tiles. The following subsections describe the details of how these fields are used. These fields need not be read if hierarchical culling is disabled. They need not be written if depth and stencil updates are disabled. The Color Blend block updates these fields when it writes an 8x8 tile to the depth buffer.

Each Tile Data cache line also has a dirty bit and sixteen Pending counts. Stage 1 increments the Pending count for each 8x8 tile that it processes. The Color Blend block decrements the Pending count as processing finishes for each 8x8 block. The Tile Data cache cannot flush a line until all of its Pending counts are zero. This ensures that the Tile Data remains in the cache until it is no longer required by the Depth Test and Color Blend blocks. Before flushing a Tile Data cache line, the 16 tiles that it describes must also be flushed, since writing them to memory can alter the values in the Tile Data cache.

{**Note**: Mention the quad data cache, which provides finer granularity for hierarchical tests without requiring more memory bandwidth. The values would all be the same when the Tile Data cache line is first loaded, but could be different after receiving updates from the Depth Blend logic.}

4.3 Stage 2: Perform Z Kill and Probe Depth Cache

The second stage reads through the queue of 8x8 tiles in parallel with stage 1, though it cannot pass stage 1. For each tile, stage 2 waits until data is available in the Tile data cache. Then it performs three actions. First, it compares the StencilCode for the 8x8 tile with the stencil operation specified on this 8x8 tile, to see if the stencil operation passes, fails, or both. Second, it compares the PminDepth or Zmax for the 8x8 tile against the Zcull limit to determine if this 8x8 tile is guaranteed to fail the depth test. As a result of these tests, it sends the Rasterizer a bit to indicate whether the whole 8x8 tile can be killed. If the 8x8 tile is not killed, it sends the tile address and coarse mask to the depth cache, so that the block can be read into the cache if it is not already present.

{**Open issue**: Is it worthwhile to use the Zplane for the tile in the Zcull test? Also, the tile logic could compute pass/kill on a 2x2 quad basis. It isn't clear that this is worthwhile.}

{**Note**: The tile logic could also perform a test to determine whether any data needs to be read from the depth buffer. The depth buffer does not need to be read if either the entire tile is covered by the Zplane and the depth test passes at all samples, or if the tile is stored Uncompressed and hierarchical Z can determine whether every sample either passes or fails. At present, this doesn't seem worth computing.}

4.3.1 *Stage 2a: Tile Stencil Test*

The tile data contains a 4-bit Smask for each 8x8 tile that consists of a high order compression bit and a 3-bit Stencil code. The code specifies the range of stencil values in the 8x8 tile, relative to a Compare value in the range [1..254].

The stencil code is updated by the depth logic as stencil tests are performed and the stencil buffer changes. They are also modified by the hierarchical stencil logic is HierStencil is enabled. The stencil test involves a reference value, a comparison operation, three update actions, and 8-bit masks that select the bits to compare and the bits to update in the frame buffer. Hierarchical stencil testing must be disabled unless both masks are equal to 0xFF.

The following table describes the eight values for the 3-bit stencil code. The Stencil code specifies whether any stencil values in the 8x8 tile are equal to, less than, or greater than the Compare stencil value. Clear (000) is a special choice that indicates that the stencil values all have a Clear value, e.g. as the result of a fast-clear operation. When an 8x8 tile that has this code must be read from the frame buffer, the stencil cache is filled with the Clear value and the stencil code is set to Equal, Less, or Greater, depending whether Clear is =, <, or > the Compare value.

><=	Name	Description
000	Clear	Replace all stencil values with Clear value when reading the tile from memory
001	Equal	Compare value is equal to all stencil values (all equal, none less or greater)
010	Less	Compare value is less than all stencil values (all less, none equal or greater)
011	Lequal	Compare value is greater than no stencil values (some less or equal, none greater)
100	Greater	Compare value is greater than all stencil values (all greater, none less or equal)
101	Notequal	Compare value is equal to no stencil values (some less orgreater, none equal)
110	Gequal	Compare value is less than no stencil values (some less or greater, none equal)
111	Unknown	Nothing is known about how the Compare value compares to the stencil values

Table 1: Stencil Codes for 8x8 Tiles

The stencil test compares a Reference value to a stencil value, first ANDing both values with an 8-bit mask. The mask will be ignored here since it must equal 0xFF for hierarchical stencil testing to be enabled. The stencil code may be used to determine whether the stencil test passes, fails, or may do both over a tile of quad. This result depends on the stencil function and on whether the Reference value is equal to, less than, or greater than the compare value. Hierarchical stencil checking is most effective when the reference value equals the compare value.

The following tables show the result of the eight stencil tests for the seven stencil codes, depending on the relationship between the Reference value and the Compare value. "Pass" and "Fail" mean that all stencil comparisons in the tile pass or fail, whereas "both" means that both pass and fail results may occur in the 8x8 tile or 2x2 quad. Note that the Clear code is not listed since a tile with the Clear code was converted to either Equal, Less, or Greater.

Stencil	Stencil Function: reference stencil value OP tile stencil values, Reference==Compare							
Code	Never	Always	Less	Lequal	Equal	Gequal	Greater	Notequal
Equal	Fail	Pass	Fail	Pass	Pass	Pass	Fail	Fail
Less	Fail	Pass	Pass	Pass	Fail	Fail	Fail	Pass
Lequal	Fail	Pass	Both	Pass	Both	Both	Fail	Both
Greater	Fail	Pass	Fail	Fail	Fail	Pass	Pass	Pass
Notequal	Fail	Pass	Both	Both	Fail	Both	Both	Pass
Gequal	Fail	Pass	Fail	Both	Both	Pass	Both	Both
Unknown	Fail	Pass	Both	Both	Both	Both	Both	Both

Table 2: Stencil Function Pass/Fail for Reference==Compare

Stencil	Stencil Function: reference stencil value OP tile stencil values, Reference < Compare							
Code	Never	Always	Less	Lequal	Equal	Gequal	Greater	Notequal
Equal	Fail	Pass	Fail	Fail	Fail	Pass	Pass	Pass
Less	Fail	Pass	Both	Both	Both	Both	Both	Both
Lequal	Fail	Pass	Both	Both	Both	Both	Both	Both
Greater	Fail	Pass	Fail	Fail	Fail	Pass	Pass	Pass
Notequal	Fail	Pass	Both	Both	Both	Both	Both	Both
Gequal	Fail	Pass	Fail	Fail	Fail	Pass	Pass	Pass
Unknown	Fail	Pass	Both	Both	Both	Both	Both	Both

Table 3: Stencil Function Pass/Fail for Reference < Compare

Stencil	Stencil Function: reference stencil value OP tile stencil values, Reference > Compare							
Code	Never	Always	Less	Lequal	Equal	Gequal	Greater	Notequal
Equal	Fail	Pass	Pass	Pass	Fail	Fail	Fail	Pass
Less	Fail	Pass	Pass	Pass	Fail	Fail	Fail	Pass
Lequal	Fail	Pass	Pass	Pass	Fail	Fail	Fail	Pass
Greater	Fail	Pass	Both	Both	Both	Both	Both	Both
Notequal	Fail	Pass	Both	Both	Both	Both	Both	Both
Gequal	Fail	Pass	Both	Both	Both	Both	Both	Both
Unknown	Fail	Pass	Both	Both	Both	Both	Both	Both

Table 4: Stencil Function Pass/Fail for Reference > Compare

The stencil function selected to update a stencil value depends on the result of the depth test, as well as the result of the stencil test. Therefore, the updated 3-bit stencil code cannot be computed until after performing the depth test.

4.3.2 *Stage 2b: Tile Depth Test*

If the stencil test is guaranteed to fail, then the depth test need not be performed, since a failing stencil test prevents color and depth from being updated. If the stencil test is either guaranteed to pass or may either pass or fail, then the hierarchical depth test must be performed if HierDepth is enabled.

The tile data contains a 20-bit Zrange value for each 8x8 tile that decodes to two 14-bit depth values that are called TminDepth and TmaxDepth. Each is specified as a 0.14 fixed point value, with TminDepth implicitly zero extended to increase its precision and TmaxDepth implicitly extended with ones to increase its precision. Therefore the range of depth values represented is the half-open interval [TminDepth .. TmaxDepth+2^-14). Depth values are clamped to the half-open interval [0..1) for purposes of depth testing.

The depth test compares a depth value from the triangle against the depth value at each sample point. The SC sends RB two depth values, called PminDepth and PmaxDepth, which are the upper 10 bits of the smallest and largest vertex depth values for the primitive being rendered. The following table shows whether the depth test passes, fails, or may both pass and fail on the samples within the 8x8 tile. If all depth tests fail, the 8x8 tile can be killed. If all depth tests pass, then the RB does not need to perform depth comparisons for the 2x2 quads in this 8x8 tile.

Depth	Depth Function: primitive depth values OP tile depth values							
Comparison	Never	Always	Less	Lequal	Equal	Gequal	Greater	Notequal
Pmax < Tmin	Fail *Tmin:Tmax*	Pass Pmin:*Tmax*	Pass Pmin:*Tmax*	Pass Pmin:*Tmax*	Fail *Tmin:Tmax*	Fail *Tmin:Tmax*	Fail *Tmin:Tmax*	Pass Pmin:*Tmax*
Pmax ≥ Tmin Pmin < Tmin Pmax ≤ Tmax	Fail *Tmin:Tmax*	Pass Pmin:*Tmax*	Both Pmin:*Tmax*	Both Pmin:*Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both Pmin:*Tmax*
Pmin < Tmin Pmax > Tmax	Fail *Tmin:Tmax*	Pass Pmin:Pmax	Both Pmin:*Tmax*	Both Pmin:*Tmax*	Both *Tmin:Tmax*	Both *Tmin*:Pmax	Both *Tmin*:Pmax	Both Pmin:Pmax
Pmin > Tmin Pmax < Tmax	Fail *Tmin:Tmax*	Pass *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*
Pmin ≤ Tmax Pmin > Tmin Pmax > Tmax	Fail *Tmin:Tmax*	Pass *Tmin*:Pmax	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin*:Pmax	Both *Tmin*:Pmax	Both *Tmin*:Pmax
Pmin > Tmax	Fail *Tmin:Tmax*	Pass *Tmin*:Pmax	Fail *Tmin:Tmax*	Fail *Tmin:Tmax*	Fail *Tmin:Tmax*	Pass *Tmin*:Pmax	Pass *Tmin*:Pmax	Pass *Tmin*:Pmax

Table 5: Depth Test Pass/Fail and Tmin/Tmax Update

TminDepth and TmaxDepth may need to be updated as a result of the depth test. If the depth test does not fail and depth updates are enabled, then either TminDepth must be set to min(TminDepth,PminDepth) or TmaxDepth must be set to max(TmaxDepth,PmaxDepth), or both, depending on the comparison mode. The table above shows when to replace TminDepth and TmaxDepth with PminDepth and PmaxDepth. This over-estimates the effect of the depth test on the 8x8 tile, but is guaranteed not to kill an 8x8 tile that might pass the depth test. It also allows changes in the depth comparison mode without flushing the pipe. Accurate values for TminDepth and TmaxDepth are computed by the RB Write block, when the 8x8 depth tile is written to memory.

More accurate values for TminDepth and TmaxDepth would result if the RB waits until performing the per-quad depth test to update TminDepth and TmaxDepth with min(TminDepth,QminDepth) and max(ZaxTile,QmaxDepth). This method does not generate false kills, provided that the shader pipe is flushed when the depth comparison function changes. But it also does not improve the number of kills that can be generated

4.3.3 *Stage 2c: Tile Stencil Update*

Finally, the stencil value is modified using one of three stencil operations, depending whether the stencil test fails (Sfail), the stencil test passes and the depth test fails (Zfail) or both pass (Zpass). The following equations determine which of the three operations may apply within the tile or quad:

```
May_Sfail   = stencil_fail | stencil_both
May_Zfail   = (stencil_pass | stencil_both) & (depth_fail | depth_both)
May_Zpass = (stencil_pass | stencil_both) & (depth_pass | depth_both)
```

The following table shows how to modify the stencil code for the 8x8 tile, based on applying a single stencil op. Italics indicate cases where the stencil code and stencil values are nnot modified. If the stencil needs to be modified, then the stencil data must be loaded into the depth cache. If the stencil does not need to be modified, then the stencil data does not need to be loaded. Note that Compare is in the range [1..254], so it is never equal to either 0 or 255. Also note that the Invert stencil operation depends on whether the high order bit of Compare is 1 or 0.

Stencil Code	**Stencil Operation for Sfail, Zfail, or Zpass**								
	Keep	**Zero**	**Replace**	**Incr Clamp**	**Decr Clamp**	**Incr Wrap**	**Decr Wrap**	**Invert cmp<7>==1**	**Invert cmp<7>==0**
Equal	*Equal*	Less	*Equal*	Greater	Less	Greater	Less	Less	Greater
Less	*Less*	Less	Equal	Lequal	Less	Unknown	Unknown	Unknown	Greater
Lequal	*Lequal*	Less	Equal	Unknown	Less	Unknown	Unknown	Unknown	Greater
Greater	*Greater*	Less	Equal	Greater	Gequal	Unknown	Unknown	Less	Unknown
Notequal	*Notequal*	Less	Equal	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown
Gequal	*Gequal*	Less	Equal	Greater	Unknown	Unknown	Unknown	Less	Unknown
Unknown	*Unknown*	Less	Equal	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown

Table 6: Stencil Op Results for 8x8 Tiles

If a single one of May_Sfail, May_Zfail, and May_Zpass is true, apply the corresponding stencil op and update the 3-bit stencil code according to the table above. If more than one of May_Sfail, May_Zfail, and May_Zpass is true, compute an updated stencil code for each valid case, then OR the results together. For example, if May_Zfail and May_Zpass are both true, and the stencil op for one produces Equal and the stencil op for the other produces Greater, then the result would be Gequal, since resulting stencil values may be either equal to or greater than the compare value..

5. RBD: Depth Test Block

The Depth Logic processes 2x2 stamps within an 8x8 tile. In the normal mode of operation, the Zplane for the 2x2 stamps comes from slope that is stored in the 8x8 tile. The Depth Logic tests each 2x2 against the Zmin/Zmax to see if depth comparisons even need to be performed. It also updates Zmin and Zmax based on the samples that pass each 2x2 Z test. The second stage also issues depth and color reads for any blocks that pass the Zcull test and are not currently in the depth and color caches. For uncompressed 8x8 tiles, this stage only reads the 2x2 stamps specified by the Coarse field. This field specifies the 2x2 blocks that the SC determines might contain a valid sample. This allows the RB and the RS to process only the potentially useful portions of the 8x8 tile. For compressed tiles it is usually necessary to read or write the entire tile. A register will allow a threshold for the amount of compression that must be possible before the RB writes out tiles in a compressed format.

{**open issue**: R300 does w buffering: why & do we need it? If so, the shader could compute it. Define a 24-bit floating depth format.}

There is a shader mode that computes a Z value for each pixel. In that case, four Z values come from the shader pipe. They can be treated as individual Z values that each apply to all samples of their pixel. Alternately, the Depth Logic can produce a Zplane from the four Z values. This allows individual samples to have separate Z values that approximate the shape determined by the the Z values computed at the center of each pixel.

Perform stencil test on 2x2 stamp, if necessary
Perform Zpass test on 2x2 stamp
Perform depth comparison on 2x2 stamp, if necessary
Update Zmin and Zmax for 8x8 tile, if necessary
Update 8x8 tile depths, if necessary
Pass sample update and empty masks to the Color Logic

Z values are also specified per-pixel for shadow buffer tests. The Depth Logic converts the four Z values into a Zplane that it uses to test the samples within the 8x8.

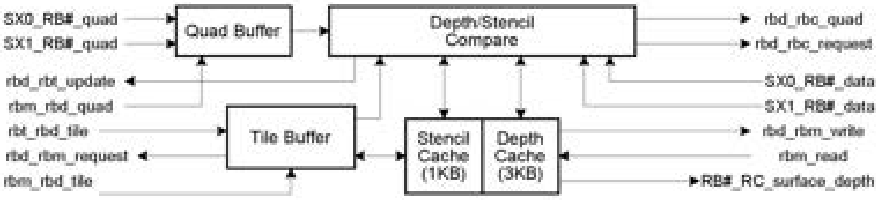


Figure 18: Depth Test Block Diagram

5.1 Input FIFOs

The Depth Logic Block uses two input fifos. Each is implemented using a single port memory that operates on a four-clock cycle. In each of these fifos, two of the four clocks are dedicated to writing data from either this pipeline's Rasterizer (RS) or from the Texture Unit (TU). The other two clocks are used to read data for use in one of the three Tile Logic stages.

The RS receives 8x8 tiles from the SC and culls them in two ways. It culls out tiles that are guaranteed to fail the Z test. This test is actually performed in the RB, which passes Zcull pass/fail bits back to the RS. The RS also culls out tiles that have no valid samples. {The RS needs to produce a valid tile every four clocks; I assume here that it can pass a tile to the RB every two clocks. There is a straightforward solution if the RS needs to pass RB a tile on every clock.} The following list describes the fifo that is written by the RS.

Stamp Queue (stages 2 and 3, written by RS)

16-bit Stamp mask: specifies the 2x2 stamps that will be processed by the shader pipeline
1-bit Covered: specifies whether the entire 8x8 tile is covered by this triangle (**is this needed?**)
7-bit (or less) Index: specifies the corresponding entry in the Zplane and Tile data queues

(Note: the above is replaced by a queue of individual 2x2 stamps. Each has an Index, the address of the 2x2 stamp within the 8x8 block, and a 32-bit mask of the covered samples in this 2x2 stamp.)

There is also a fifo that is written by the Texture Unit (TU). The TU sends a single bus to all four RBs, which can write each of them on one clock out of four. This leaves three clocks for the RB to read from the queue.

Ztexture Queue (stages 1, 2 and 3, written by TU):

20-bit Address: specifies XY position of the 8x8 tile
3-bit Context: specifies which set of context parameters to use
96-bit Zdata: Either four 24-bit Z values or a 56-bit Zplane, 16-bit stamp mask
16-bit Stamp mask: one bit per 2x2 stamp specifies whether it needs to be processed

(**Open issue**: When the shader pipe computes per-pixel Z values, how should we pass them to the RB? Under this scheme, when the shader computes per-pixel Z values, they are passed to the TU. The TU either passes these four values to the RB or changes them to

5.2 Stage 3: Depth/Stencil Tests

The third stage of the RB performs depth and stencil tests on 2x2 stamps. The RB determines the stamps to process based on 8x8 tile commands from the Rasterizer. The rasterizer only sends 8x8 tiles that contain at least one valid sample and that pass the Zcull test. Each such command contains a pointer to Zplane data that was passed by the SC. Each 2x2 operation stalls until

(Talk about per-pixel depth values. In this case, the stage stalls until depth data is available from the shader pipe, via a second fifo. Also talk about shadow buffer tests.)

6. RBC: Color Blend Block

The Color Logic inputs a 2x2 stamp of color data from the shader pipe. It also inputs a mask of the samples in that stamp that pass the depth test from the Depth Logic. The Color Logic performs fog blending and alpha blending on the dest colors.

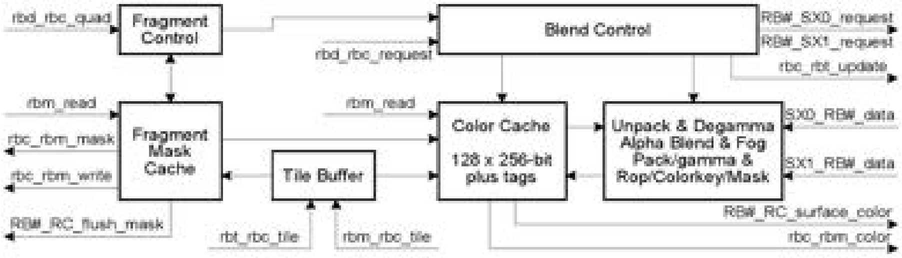


Figure 19: RBC: Color Blend Block Diagram

The Color Logic also supports filtering anti-aliased pixels. This occurs on requests from the Texture Unit.

Finally, the Color Logic supports two bypass modes. These output 2x2 stamps into a list of 512-bit entries, for use in multi-pass rendering. This is described in the following section.

Output four 32-bit floats per pixel in vertex or shader bypass modes, Else:
Compress 32-bit floats to 20-bit floats for RB processing
Output four 20-bit floats per pixel if RB bypass threshold met, Else:
Read 2x2 stamp from Color cache, if necessary
Perform fog and alpha blend multiplies, if necessary
Convert 20-bit floats to frame buffer pixel format
Perform rasterop, if necessary, and write result to the Color cache

6.1 Input FIFOs

The Color Logic Block uses three input fifos. The first is the fifo of 8x8 tile data from the Rasterizer, described above. The second fifo buffers results from the Depth Logic Block. The third fifo buffers pixels that are computed by the shader pipe.

{List the data passed from the Depth Logic. This includes a Pass bit per sample (this sample passed the depth test) and an Empty bit per sample (no depth value is stored for this sample). The Color Logic must drop the color at a sample if the Depth Logic has dropped the depth value for a sample and therefore cannot determine if it would pass the depth test. This avoids multi-pass bugs due to dropping samples.

{Talk about filtering multi-sampled pixels and passing them to the texture unit.}

6.2 Stage 4: Color Processing

{}

6.3 Fragment Processing

The first step in processing a quad in the RB Color Blend block involves determining what fragment operations to perform. A fragment is a color together with a mask that specifies which samples it is visible at within a pixel. With single-sample pixels, there is exactly one fragment per pixel, so each pixel is either processed or not, depending on whether the source pixel is valid. With multi-sample pixels, the process is more complex, since the destination pixel can contain multiple fragments and the source pixel may overlap one or more of them.

`The first step for fragment processing is to determine how the source pixel's sample mask intersects the sample mask of each fragment in the pixel. The following list defines the alternatives, where Ms and Md are the source pixel's sample mask and the sample mask of a particular fragment of the dest pixel.

Empty	$Ms \cap Md = \phi$	The two masks do not intersect
Cover	$Ms \supseteq Md$	The dest mask is covered by the source mask
Overlap	$Md \neq Md - Ms \neq \phi$	The source mask covers part of the dest mask

Table 7: Fragment Intersection Types

The next step determines what blend operations to perform. There are two different sets of rules for fragment processing, depending on whether the blend operation is write-only or read-write. A write-only operation depends only on the source color at each sample, whereas a read-write operation produces a result that depends on both the source color and the dest color at each sample.

The lists below specify the write-only and read-write operations to perform on each fragment. In the write-only case, the operation produces one new fragment and removes zero or more existing fragments. In the read-write case, the operation does not remove any fragments and can potentially double the number of fragments, up to the total number of samples per pixel. The fragment processing logic updates the fragment data and passes the specified blend operations to the alpha blender.

Empty	Ignore this dest fragment
Cover	Delete this dest fragment
Overlap	Replace the dest fragment mask with Md – Ms
(also)	Create a fragment with mask Ms, blended using the source color

Table 8: Fragment Rules for Write-Only Blending

Empty Ignore this dest fragment
Cover Blend the source color into the dest color at this fragment
Overlap Replace the dest fragment mask with Md – Ms
and create a new fragment with mask Ms – Md by blending the source and dest colors

Table 9: Fragment Rules for Read- Write Blending

There is one additional complication involved in allocating fragments. The blend logic uses two clocks to compute the four pixels of a quad, so it actually computes two pixels in parallel. This implies that fragment allocation must occur for pairs of pixels instead of for single pixels. When creating a new fragment for a pair of pixels, the fragment logic must distinguish three cases, based on the sample masks Md0 and Md1 for the pair of pixels being written

Both	Md0 ≠ ϕ & Md1 ≠ ϕ	Allocate this pair of pixels to an empty fragment-pair
Left	Md0 ≠ ϕ & Md1 = ϕ	Search for a fragment-pair that has an empty mask for the lefthand pixel. If found, modify the lefthand pixel and leave the righthand pixel unchanged. Else, allocate this pair of pixels to an empty fragment-pair.
Right	Md0 = ϕ & Md1 ≠ ϕ	Search for a fragment-pair that has an empty mask for the righthand pixel. If found, modify the righthand pixel and leave the lefthand pixel unchanged. Else, allocate this pair of pixels to an empty fragment-pair.

If the above steps are not performed, it is possible for a tile with S samples per pixel to contain up to 2S fragments, since some pixel-pair could have up to S fragments that touch a sample of the lefthand pixel but leave the righthand pixel empty, followed by another S fragments that touch a sample of the righthand pixel but leave the lefthand pixel empty. There is only room in the cache and in memory for up to S samples, so the algorithm must prevent this case from occurring.

7. RC: Read Data Block

8. RBW: Write Data Block

9. External Interfaces

Name	Direction	Bits	Description
RTRn	MC → RB	1	Ready to receive a new RB data request
DataSelect	RB → MC	8	Specifies a 256-bit entry or specifies that this request should be ignored

Table 10: RB DataRequest Interface

{}

9.1 Scan Converter Interface

{}

96-bit Zplane
16-bit Coarse and Covered masks
10-bit Zmin, Zmax
10-bit Xtile, Ytile
3-bit Context
1-bit Background (saves comparing Zplane to background)
1-bit Direction (front or back facing

9.2 Rasterizer Interface

{}

16-bit Coarse and Covered masks to the RE for HiZ elimination
16-bit Coarse and Covered masks from the RE
32-bit sample mask from the RE

9.3 Texture Unit Interface

{}

9.4 Shader Pipe Interface

{}

9.5 Memory Controller Interface

{}

10. Register Interface

The RB supports multiple contexts so that the CP can issue commands that use a different context without first flushing the pipe. The context is identified by a 3-bit index that is associated with each 8x8 tile issued to the RB by the Scan Converter (SC). {How many contexts?} {I assume here that the RB uses a single sub-context. The RB context could be split into multiple pieces if that is necessary, but this must be defined before the RB interfaces, since that would require multiple context indices to be associated with each tile.}

10.1 Surface Descriptors

There are {16 to 32} surface descriptor registers, which are similar to the base address registers in the Memory Hub. These registers store all of the surfaces in use by all of the active contexts. Context-dependent registers store indices into the surface descriptor array. Each surface in the array must be different, so that when multiple contexts use the same surface, they all use the same index. This allows the RB to keep its cache coherent by tagging each cache line with the index and an (x,y) position.

{Describe the surface descriptors, which include the following fields.}
base address
pixel size and format
endian control: no swap, word swap, dword swap {there is a fourth one}
dimensionality
width
height
low order Z bits if this is a 2D slice from a 3D array.
Address format: tiled in local memory, tiled in system memory, or linear in system memory.
Samples per pixel (1, 4, or 8)
number of bytes an 8x8 tile occupies in memory

{Is the color format an itemized list, or a more generic description of component sizes and positions, within certain limits? I prefer the latter.}

10.2 Depth Test Registers

Enable HiZ (possibly both 8x8 and 2x2)
Enable HiS (possibly both 8x8 and 2x2)

Enable shadow buffering
Enable per-pixel depth from the shader pipe
Stencil reference, base, and background values
Depth background value
Depth and control fields from OpenGL
(Do we need to select round or truncate for producing FB depth values?)
Depth (and maybe stencil) surface indices
MC address queue for each depth/stencil surface
(Should we support a second depth buffer?)

10.3 Color Blend Registers

Alpha test threshold
Alpha test control fields
Blend constant register
Fog factor enable (uses the blend constant)
Plane mask (pattern should repeat on pixel size, for good performance enable/disable whole bytes)
Enable multiple color buffers
Four color surface indices
Dither mode: truncate (add 0), round (add 0.5), dither (error diffusion?) Dither offsets?
Rop mode (do we have to directly support Rop3?)
MC address queue for each color surface
Enable tile filtering

10.4 Other Registers

Stipple pattern (?)
Multisample mode
Vertex buffer index (more than one?) and mode
Something about cache format, flushing, etc.?
Status bits (clean caches, RB idle, etc.)
Compression mode enables: force writes to be uncompressed or to not use certain modes
Enables for various optimizations (to let us turn them off)
Performance counters

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	21 March, 2002	17 September, 2004	GEN-CXXXXX-REVA	1 of 31

Author: Larry Seiler

Issue To: **Copy No:**

R400 Render Logic Architectural Specification

version 0.5a

Overview: This is an architectural specification for the R400 render logic blocks, including the Render Backend blocks (RB), the Shader Export blocks (SX), and the Render Control block (RC). It provides an overview of the requied capabilities and expected uses of the block. It also describes the block interfaces, internal subblocks, and provides internal state diagrams.

AUTOMATICALLY UPDATED FIELDS:
Document Location: D:\r400\doc_lib\design\blocks\rb\R400_RenderBackend.doc
Current Intranet Search Title : R400 Render Backend Architectural Specification

APPROVALS	
Name/Dept	Signature/Date

Remarks:

THIS DOCUMENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATI TECHNOLOGIES INC. THROUGH UNAUTHORIZED USE OR DISCLOSURE.

"Copyright 2001, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished work created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this unpublished work. The copyright notice is not an admission that publication has occurred. This work contains confidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or transmitted in any form or by any means without the prior written permission of ATI Technologies Inc."

Table Of Contents

Table Of Figures

Table Of Tables

Revision Changes:

Rev 0.1 (Larry Seiler) Date: April 6, 2001	First draft
Rev 0.2 (Larry Seiler) Date: June, 2001	Major updates on Parameter Buffer and Tile Logic blocks. Lesser changes to other blocks.
Rev 0.3 (Larry Seiler) Date: November 7, 2001 – December 20, 2001	Changes to reflect the new shader pipe design.
Rev 0.4 (Larry Seiler) Date: January 30, 2002	Major changes in the block overview descriptions, addition of a section on data flows, changes in the SX block description. Detailed RB block descriptions are temporarily removed, since they are out of date.
Rev 0.5 (Larry Seiler) Date: July 8, 2002	SX export buffer no longer stores depth compressed, various elaborations and extended explanations.
Rev 0.5a (Larry Seiler) September 17, 2004	Fixed a bug in the Stencil Function Pass/Fail table

Open Issues and Known Deficiencies:

The export buffer and SP->SX interface would be much simpler with a wider data bus.	
Work out details of the shadow buffer paths through the RB.	
Verify that the surface and context synchronization schemes meet our needs	
Work out details of Gaussian filter multi-sample resolve through the RB.	
How does vertex export address generation work – is the plan of record stable?	
Do we need to support sample dropping?	
Should we still support the fog blend?	
What changes need to be made in the SX for handling cached parameters?	
NOTE: bus names in the figures are not up to date in rev 0.4	

1. Overview

The R400 render logic blocks consist of the Render Control block (RC), the Shader Export blocks (SX), and the Render Backend blocks (RB). Together, these blocks accept depths, colors, and coverage masks in order to perform hierarchical tests, depth/stencil tests and destination color blend operations. They input 8x8 tiles and 2x2 quads from the Scan Converter (SC) and Shader Pipe (SP) blocks, read data from the Memory Controllers (MC), and write out the results. The RB blocks also compute shadow buffer coverage and filter anti-aliased pixels in response to commands from the texture pipe (TP). Finally, they pass through other types of export data computed in the Shader Pipes and write it to linear arrays in memory.

The figure below illustrates the render logic blocks and how they relate to other R400 blocks. In the upper half of the figure, the Render Central block (RC) distributes render tile commands to the other blocks, collects hierarchical cull results for the SC, and collects synchronization signals for the CP. The two Shader Export blocks (SX) each buffer the data that is output from a pair of Shader Pipes, as well as performing operations such as alpha test.

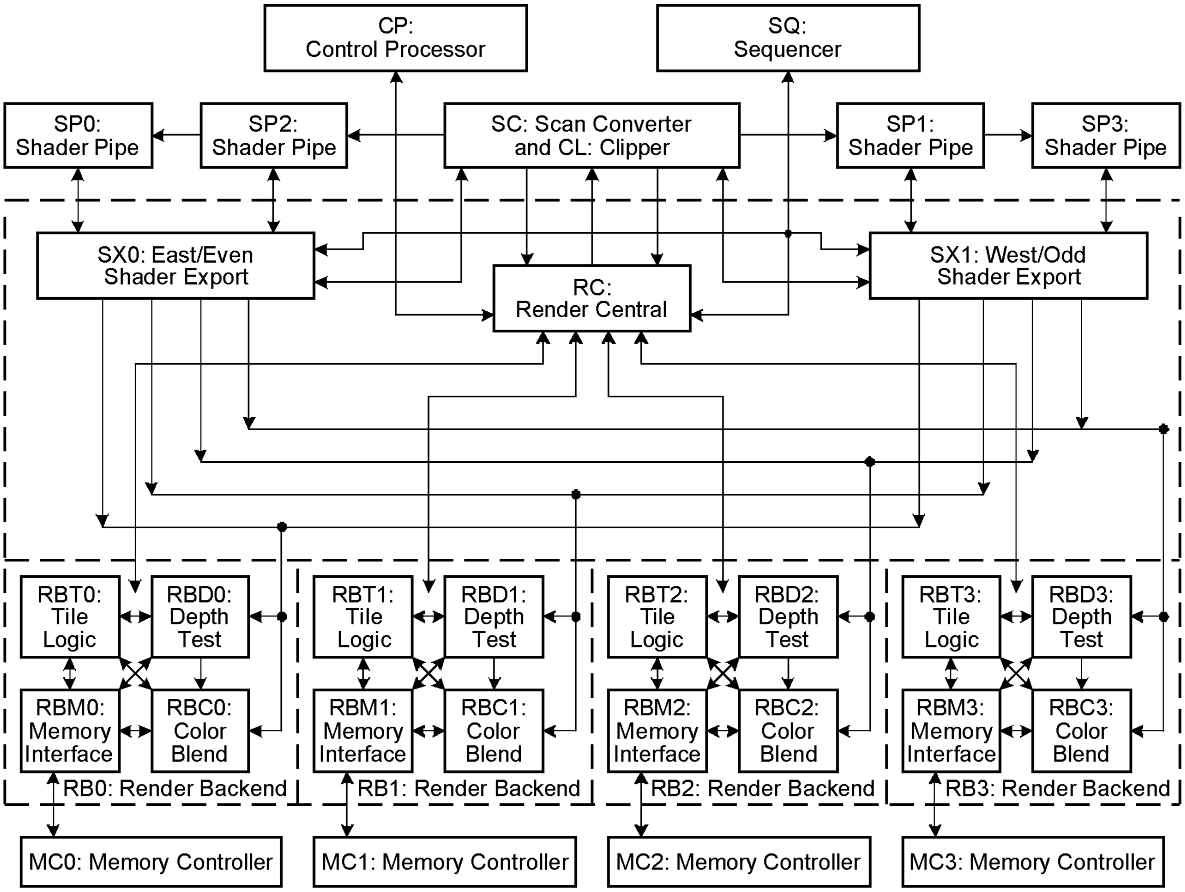


Figure 1: RB Top Level Block Diagram

The lower half of the figure shows the four Render Backend blocks (RB) and their associated Memory Controller blocks (MC). Dashed lines enclose the four sub-blocks that compose each RB. Each RB owns ¼ of the 8x8 tiles in the frame buffer and processes all render commands directed at their portion of the frame buffer. The RC and each SX block have separate busses that route to each of the four RB blocks, since each Shader Pipe (SP) can process pixels that belong to any of the RBs. These are long routing tracks in the physical layout, since the upper three blocks are physically placed near the PA block and the SP blocks, whereas the RB blocks are physically placed near the corresponding Memory Controllers.

The RB processes pixels in three sub-blocks, each of which may require memory accesses, which are managed by the fourth sub-block. The Tile Logic block (RBT) performs hierarchical depth and stencil tests in order to kill quads before they are sent to the Shader Pipe. This block manages a tile data cache that stores hierarchical and compression information about each 8x8 tile. The Depth Test block (RBD) performs depth and stencil tests, as well as shadow buffer comparisons. The Color Blend block (RBC) performs all back end color processing, including multi-sample fragment processing. It manages a fragment mask cache and a color cache. The color cache also stores non-color data that is exported to linear arrays. Finally, the Memory Interface block (RBM) performs data compression/decompression, manages the interface to the Memory Controller (MC) and converts between the data format in memory and the data format in the internal caches, when they differ.

1.1 RC: Render Central Block

The Render Central block (RC) distributes render tile commands from the Scan Converter (SC) to the RB blocks and collects synchronization signals for the CP. This block isolates the SC and CP from the details of how many RB blocks there are and how they divide up the rendering task. The R400 is able to run with one, two, or all four RBs enabled, in order to support smaller memory interfaces. The diagram below illustrates the basic elements of the RC.

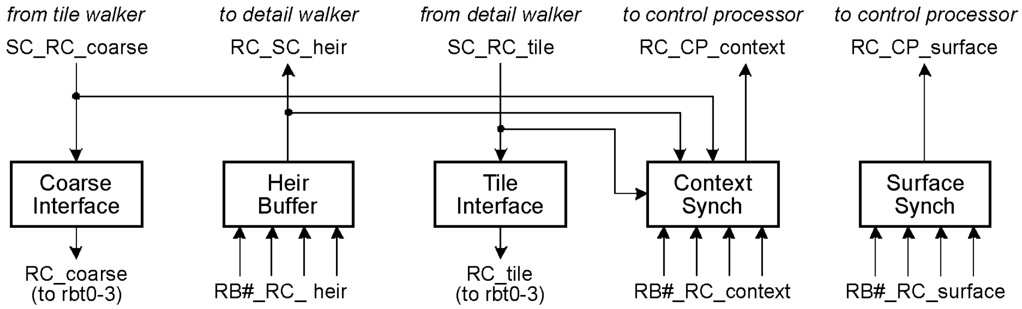


Figure 2: RC: Render Central Block Diagram

First, the SC's coarse walker finds tiles that may need to be rendered and sends them to the RC. The RC forwards this data to the four RB tile logic blocks (RBT), which each perform hierarchical tests on the tiles that they own. The RC then gathers the results of these hierarchical tests, puts them back in the original order and returns the results to the SC for use in the detail walker. The detail walker computes the valid samples inside each quad that passes the hierarchical tests, dropping a tile if none of its quads passed the test. If a quad contains any valid samples, the SC sends a quad command to the SX and SP blocks. Additionally the SC sends the RC a 16-bit mask of which quads are being processed out of each tile that passed the hierarchical tests, even if no quads contain valid samples in that tile. The RC forwards this data to the RB that controls that tile. This allows the RBs to read pixel data while the quads are being processed in the SP. The RBs need to keep track of what quads are in the shader pipe, so they also use this mask to determine which quads were killed by the SC detail walker.

The Context Synch block tells the CP when the render logic is finished processing each render context. Each tile has associated with it a state ID that specifies which render context the render logic should use to process that tile. The CP is free to reuse a context after the render is finished with it, since the render logic cannot finish a context until the SC, SP, and TP are also finished with it. Those blocks do not report when they have completed each context. A context is active in the RB when the SC issues a tile that uses that context. The context is finished when the SC issues a tile with a different context and all of the individual tiles marked with the previous context are finished. A tile is finished when it fails the hierarchical test, when the detail walker finds no valid samples in it, or when an RB reports to the Context Synch block that it has completed that tile. Therefore, the Context Synch block contains a count of the number of outstanding tiles per context. When a count goes to zero, the Context Synch block reports to the CP that the specified context is now free.

Finally, Surface Synch block supports synchronizing accesses to surfaces in frame buffer memory. Each RB receives requests directly from the register interface to synchronize on a specific range of FB addresses. The RB blocks respond to such a request by flushing their caches of all data within that surface. Each RB sub-block reports to the RC when it has finished flushing the surface data. The RC gathers these results and reports to the CP when the surface has been removed from the RB caches.

{NOTE: Add an interface to report pixels that pass the depth test, for the RB_ZPASS_DATA register.}

1.2 SX: Shader Export Block

The Shader Export Blocks (SX0 and SX1) store and process results exported from the shader pipes. There are three broad classes of exports: vertex parameters and positions, which are produced by vertex shaders; colors and depths, which are produced by pixel shaders, and raw data to be written to a linear array in memory. Raw data is produced by multi-pass shaders and by debug exports. The figure below illustrates the busses and basic internal blocks for each SX. SX0 is associated with SP0 and SP2. SX1 is associated with SP1 and SP3. They also communicate with each other and both of them communicate with the SC, SQ and all four RBs.

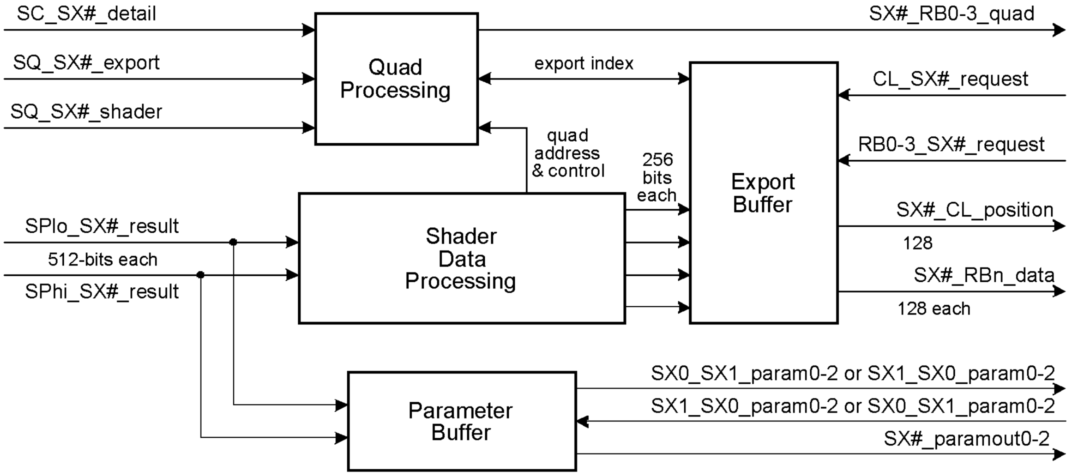


Figure 3: SX: Shader Export Block Diagram

The Parameter Selection sub-block handles vertex parameters. This includes texture coordinates, colors, and all other data specified per-vertex, except for the vertex position. Under control of the SQ, this sub-block reads three parameters from a cache for the three vertices of the current triangle. The three parameters could come from any of the shader pipes and are used by all of them, so the two SX blocks need to exchange data. The Parameter Selection sub-block then pre-processes the parameters and forwards the results to the Shader Pipe interpolators. All four interpolators receive the same parameter data. Accordingly, the two SX blocks replicate some of the multiplexing and pre-processing.

The Shader Data Processing sub-block processes all export data from the SP, except for the vertex parameters. It performs the alpha test and other tests on color data before writing them to the export buffer. It passes most other exports straight through to the Export Buffer.

The Export Buffer stores export data, other than parameters. The CL (Clipper) reads position data from a reserved portion of the buffer. The RBs read all other export data. Data is stored in the Export Buffer in a swizzled form that allows the four RBs to stream out 128-bit data without conflicts. The Export Buffer also reports the space available to the SQ, which waits to schedule a clause until there is room in the Export Buffer.

Finally, the Quad Processing sub-block sends quad commands to the RBs that correspond to the data in the Export Buffer. This sub-block receives quad commands from the SC's detail walker and buffers them until the corresponding color data emerges from the Shader Pipe. It then modifies each pixel's sample mask based on the result of the alpha test and SP pixel kill instructions and appends the location of the pixel data in the export buffer. This sub-block also generates quad commands for export data other than colors and positions. These quad commands include the memory address at which the RB must write the data, as well as the location of the non-pixel data in the export buffer. The memory address replaces the sample mask that is contained in a pixel quad command.

1.3 RBT: Tile Logic Block

The Tile Logic block performs hierarchical depth/stencil tests. It stores Zmask, Smask and Cmask compression codes for all tiles currently active in the RB. It also stores an in-flight count for each quad and tile currently being rendered. The in-flight count tracks the number of operations currently in the pipeline for a quad or tile. The in-flight counts are incremented when the RBT first receives a tile from the Scan Converter (SC) and are decremented when the tile or quad is removed from the RB, either because it is killed or because the RB finishes processing the tile or quad. The figure below illustrates the internal structure of the Tile Logic block.

The upper part of the figure shows the path used for hierarchical tests. The RC_coarse bus sends per-tile information to the Tile Logic block, including a mask of the quads in the tile that may be covered (the coarse mask). The Coarse buffer stores this information for tiles that belong to this particular Tile Logic block. The Tile Test block performs an hierarchical depth and stencil test on the tile, using per-tile hierarchical data in the Tile Cache. The results are buffered and then the Quad Test block performs a 2x2 hierarchical test on any quads that passed the 8x8 tile test, using per-quad hierarchical data in the Quad Cache. Finally, the RB#_RC_hier bus passes a mask of the result per quad back to the Render Central block.

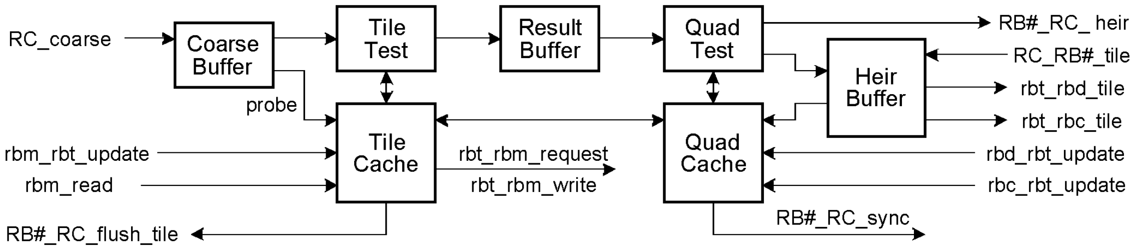


Figure 4: RBT: Tile Logic Block Diagram

The Tile Cache performs several actions in addition to providing per-tile hierarchical data for the Tile Test. First, the Coarse Buffer probes the Tile Cache to see whether each tile is present in the cache. This occurs in parallel with and in advance of the Tile Test, so that the Tile Cache can send a read request to the RB Memory Interface block early enough to cover the read latency. The Tile Cache reads 256-bit values that store data for 8 tiles. It may also read data for adjacent sets of tiles, to increase the chance that tiles are already in the cache when they arrive from the RC. These extra reads don't cost much memory bandwidth, since the Memory Controller requires multiple reads on the same page for efficiency. Finally, the Tile Cache sends tiles that pass the tile HiZ test to the RB Memory Interface block so that depth can be prefetched.

Next, the Tile Cache probes the Quad Cache for each tile that passes the hierarchical tile test. If the Quad Cache doesn't contain an entry for that tile, it allocates one, initializing each quad with the hierarchical data for the tile. This way, the Quad Cache does not need to read data from memory. The Depth Test (RBD) block sends revised stencil and depth ranges to the Quad Cache as it processes quads, so that the quad cache becomes more precise over time. The Quad Cache in turn updates the Tile Cache with more accurate hieararchical data. A quad or a tile can only be fully updated if no operations are pending for that quad or tile. Therefore, in-flight counts must be tested to ensure that the quad cache and tile cache account for quads in-flight that have not yet been processed by the RBD. The rbd_rbt_update bus reports these changes in the depth range, as well as reporting changes in the Zmask and Smask fields for each tile. The rbc_rbt_update bus reports when the Color Blend block is done with each quad, as well as reporting changes in the Cmask for each tile.

Finally, the Heir Buffer receives tile data from the SC detail walker (via the RC) and passes tile data to the RBD and RBC blocks. The data from the SC updates the in-flight counts of the quads in each tile, to account for quads killed by the detail walker. The RBC and RBD receive a mask of the quads being processed in the Shader Pipe. The RBD also receives Zplanes and hierarchical test results for individual quads.

1.4 RBD: Depth Test Block

The Depth Logic block processes 2x2 quads. In the normal mode of operation, the Depth Logic uses a Zplane to specify the depths at each sample in the quad. Alternately, the pixel shader program may compute an explicit depth per pixel. The Depth Logic block performs the depth and stencil tests, modifies the quad sample mask accordingly, updates the depth and stencil cache, and forwards the results to the RB Color Blend block. The figure below summarizes the processing in the Depth Test block.

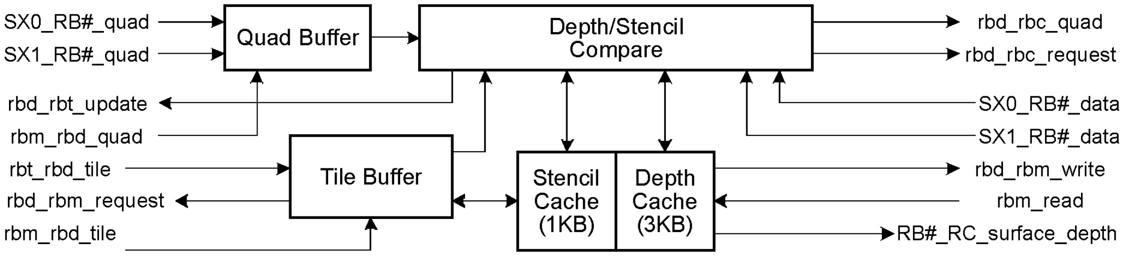


Figure 5: RBD: Depth Test Block Diagram

The Depth Test logic also computes the minimum and maximum Z within each quad that it processes and determines whether the quad contains stencil values that are equal to, greater than, or less than a specified compare value for the surface. The Depth Test logic returns this information to the Tile logic in order to update the hierarchical data for the quad.

Finally, the Depth Test logic performs shadow buffer operations. The RB Memory Interface accepts shadow buffer requests from the Memory Hub/Texture Central and converts them into tile and quad commands. The RBD stores these in separate logical queues so that shadow buffer requests can be completed ahead of regular rendering commands that are already in the queue. This is necessary to avoid deadlocks, since some quad commands currently in the queue may not be able to complete until after the shadow buffer is computed.

1.5 RBC: Color Blend Block

The Color Blend block processes quads of color data. It performs fog interpolation, alpha blending, number format conversions, gamma/degamma conversion, raster-ops, and color keying. It also processes multi-sample pixels that are defined by multiple fragments, where a fragment is a color together with a mask of the samples within a pixel where that color is visible. The Color Blend block also passes along non-blendable data that is being exported to memory.

The figure below shows the basic logic of the Color Blend block. The block receives a quad command from the Depth Test block, which specifies a quad address and a mask of the valid samples. The block looks this up in the Fragment Cache, which stores the mapping of samples to fragments for the currently active pixels. The Fragment Control block produces a sequence of blend commands based on the intersection of the quad sample mask and the fragment masks for those pixels and passes those commands to the Blend Control.

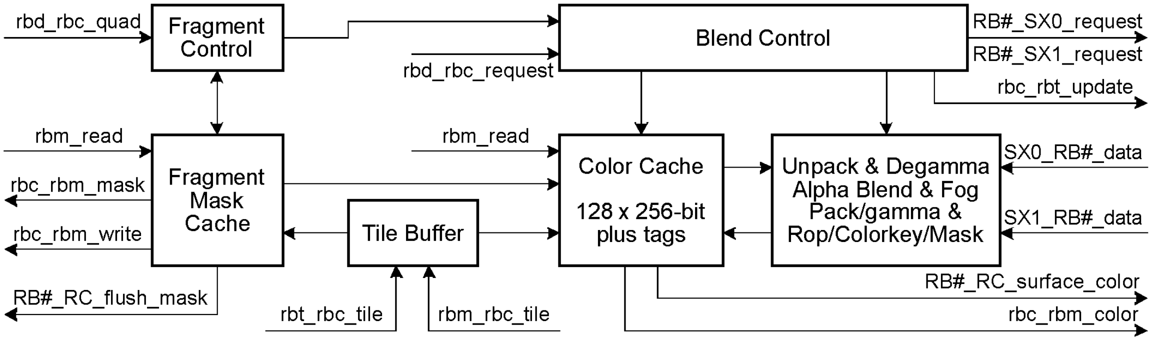


Figure 6: RBC: Color Blend Block Diagram

The Blend Control block reads the color data from the SX export buffer and controls the traditional backend blending functions on it, as well as gamma/degamma conversions and filtering anti-aliased pixels to a single color per pixel. All frame buffer pixels are converted to an internal floating point format for alpha blending, then are converted back. The Blend Control block also synchronizes the quad processing so that a quad command does not read pixel data until any previous write to the same pixel has completed.

The Color Cache contains 64 512-bit lines and has two 256-bit ports that alternate between serving the internal blend logic and the RBM block. This allows the blend logic to process a quad every other clock and allows the RBM to saturate both the memory read bus and memory write bus. Non-blendable data passes through the Blend Control block and into the color cache, where it is written out to memory in the usual way. This allows the RBM to gather this data into multi-bursts for memory efficiency, as it does for ordinary color data. The Color cache also supports flushing cache entries for surface synchronization and for auto-flush.

1.6 RBM: Memory Interface Block

The RB Memory Interface block transfers data between the other three RB sub-blocks and the Memory Controller (MC). The MC uses short access request queues with just-in-time transfers of new read requests and write requests. When the MC processes an access request, the RBM must quickly move a new request into the queue. If possible, the RBM provides data that is likely to be on the same page as the previous requests, so that the MC can perform an efficient series of burst accesses.

{Issues: some data needs to be compressed/decompressed. The RBM also reads commands from the Texture Unit to perform shadow buffer comparisons or to filter anti-aliased pixels. We need to avoid deadlocks, which can arise if the RBM commits a read that cannot complete until some other read completes. E.g., if it commits a depth read, then it must be possible to load that read data into the cache. If the queue fills up, so that the cache won't have room until a color read completes, we could have deadlock. This didn't occur when depth and color reads used separate queues.}

{The following figure needs to be vastly modified.}

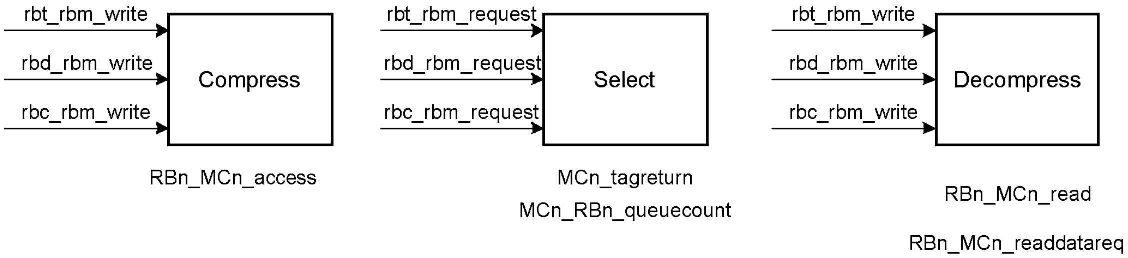


Figure 7: RBM: Memory Interface Block Diagram

The RB Memory Interface block compresses and decompresses tile, depth and color data, if necessary, and issues memory read and write requests to the MC. It also writes shadow buffer comparison results and filtered anti-aliased pixels to the Texture Unit.

2. Render Logic Data Flows

This section describes the sequence of data transfers between the render logic blocks and other R400 blocks. There are three basic types of transfers: pixel render transfers, pass through transfers, and texture unit transfers.

2.1 Pixel Render Transfers

The basic operation of the RB is to render pixels to the frame buffer. The following list describes the data transfers involved in rendering pixels.

1. SC->RC->RBT: The SC tile walker sends the RC an 8x8 tile, together with its address, a Zplane, and a coarse mask of the quads within the tile that might need to be rendered. The RC forwards each tile to all four RB tile logic blocks, each of which buffers just the tiles it owns.
2. RBT<->MC: The RB tile logic blocks read and write per-tile data as necessary in order to compute hierarchical tests on their tiles.
3. RBT->RC->SC: The four RB tile logic blocks return their hierarchical test results to the RC. The RC assembles the hierarchical test results into their original order and forwards them to the SC. It also keeps track of whether an entire context is culled by the hierarchical test (see final step).
4. The following sequence of tile operations occurs, roughly sequentially but with some lookahead:
   a. SC->RC->RBT: The SC sends each tile that was not culled by the hierarchical test to the RC, which forwards it to the RB tile logic. The tile logic buffers this as the list of upcoming tiles to render.
   b. RBT<->MC: The tile logic issues color mask reads and writes to load the mask cache with multi-sample fragment data for the color cache. Reads and writes can occur in parallel due to the MC buffer design.
   c. RBD<->MC: The tile logic issues depth data reads and writes to load the depth cache.
   d. RBC<->MC: The tile logic issues color data reads and writes to load the color cache. For multi-sample pixels, this operation uses the data in the mask cache and so much wait for the mask cache to be valid.
   e. RB->MC: The RB must sometimes clear cache entries in response to synchronization requests or autoflush timeouts.
5. The following sequence of quad operations occurs, in parallel with the above tile operations:
   a. SX->SC: The SX tells the SC how much space is available in the export buffers. A clause cannot begin until there is room to buffer all of the data that will be generated by the clause.
   b. SC->SX: The SC sends quad addresses and sample masks to the SX blocks. The SX buffers them until the shader pipe emits the pixel shader data corresponding to each quad.
   c. SP->SX: The SP sends color and/or depth data to the SX, which stores them in the export buffer. The SX also performs the alpha test and modifies the corresponding quad mask based on that test.
   d. SX->RBD: The SX forwards quads to the RBD so that it can performs the depth test.
   e. RBD<-SX or RBT: The RBD gets depth values for a quad from one of two places. If the shader program computes depths, it reads them from the SX export buffer. Otherwise, it reads a Zplane from the RBT, which received the Zplane from the SC in step 4a.
   f. RBD->RBC: The RB depth test logic forwards the quad to the RB color blend logic.
   g. RBC<-SX: The RB color blend logic reads the pixel shader color data from the SX export buffer, performs the backend color blend operations, and writes the result into the color cache.
6. RBC->RC->CP: When all processing is complete on the last quad of a context, the RB informs the RC. When a context is complete in all four RBs, the RC informs the CP. Note that a context may end early if its quad commands were culled by the hierarchical test.

2.2 Pass Through Transfers

The render logic also supports three types of pass through transfer, in which it transfers data generated by a shader program to its destination without modification. The most common pass through transfer moves position data from the vertex shader to the clipper (CL). The list below describes the steps involved.

1. SP->SX: Each SP vertex shader exports a vector of 16 4-vectors to its SX. Each 4-vector is either an (x,y,z,w) position or a sprite width/height with edge masks. The SX puts this in a reserved portion of the export buffer.
2. CL<-SX: The Clipper requests one position at a time from each SX.
3. SX->SQ: The SX tells the Sequencer when there is room in the buffer for another vector of positions.

The other two pass through transfers move data from a pixel or vertex shader program to memory. The list below describes the steps involved to transfer data from the pixel shader program. As for pixel rendering, each transfer groups together results from the four vertex units that make up one of the pixel shaders.

1. SP->SX: Each SP pixel shader exports a vector of four quads to its SX. Each quad consists of four 4-vectors, e.g. four RGBA colors. The SX puts this data into the export buffer.
2. SX->RB: The SX computes a frame buffer address for each of the four quads and generates four quad commands that contains those addresses. The SX sends each quad command to the appropriate RB.
3. RB<-SX: After processing all previous quad commands, the RB reads the quad from the SX export buffer.
4. RB->MC: The RB writes the quad to memory.

Finally, the following list describes vertex shader program exports. They differ from pixel shader program exports because vertex programs compute vectors of 64 independent vertices, unlike pixel shader programs, which compute vectors of 16 2x2 quads. Therefore, the SX groups together sets of four exports from the same vector unit, instead of grouping together four pixels from adjacent vector units.

1. SP->SX: Each SP vertex shader exports a vector of 16 4-vectors to its SX. The SX gathers together groups of four exports from the same vertex shader clause.
2. SX->RB: The SX computes a frame buffer address for each group of four exports and generates quad commands that contain those addresses. The SX sends each quad command to the appropriate RB.
3. RB<-SX: After processing all previous quad commands, the RB reads the data from the SX export buffer.
4. RB->MC: The RB writes the four 4-vectors to memory.

{Add in detail about caching and transfers within the RB sub-blocks?}

2.3 Texture Unit Transfers

Finally, the RB performs several types of commands for Texel Central (TC). In effect, the RB generates textures on the fly for the TC, based on data in existing color buffers or depth buffers. This section describes the data flows for each of these operations.

The following list describes the steps necessary to filter a multi-sample color buffer and return the data to TC as a texture.

1. TC->MH->MC->RBM: The TC determines that it needs to read a tile of multi-sample color data. The TC sends a command to the RBM that specifies the color surface and the block of pixels within that surface. The TC sends its command over existing datapaths through the MH to the MC that hols the multi-sample color data. The MC routes the request to its assocated RB's Memory Interface block.
2. RBM->RBC: The RBM sends the RBC a tile request. The RBC buffers this in a logically separate queue from the tile data that it receives from the RBT, though it could be a reserved portion of the same physical queue. A separate logical queue is necessary so that the RBC can process the filter request without waiting for normal render requests that are currently in the queue to complete.
3. RBC<->MC: The RBC reads the multi-sample color data and stores it in the color cache.
4. RBM->RBD: In parallel with the preceeding step, the RBM creates a sequence of quad commands that specify filtering the desired pixels. The RBM puts these into a logically separate queue from the quad commands that

the RBD receives from the SX blocks, so that the RB can process filter requests ahead of pending render commands.

5. RBD->RBC: The RBD passes quad filter commands to the RBC. The RBM sends them via the RBD so that they are processed in sequence with any shadow buffer commands.
6. RBC->RBM: The RBC computes the filtered pixel values and passes the results directly to the RBM. This data does not go into the color cache since there is no need to either save the data for later or group together multiple writes into an efficient memory access.
7. RBM->MC->MH->TC: The RBM forwards the filtered pixels back to the TC, via its existing datapath to the MC and the existing datapath from the MC to the MH.

A shadow buffer operation compares a Zplane against an existing depth buffer. For each of a group of pixel positions in the depth buffer, it computes what fraction of the Zplane is visible. The TC filters these results to determine whether pixels on a triangle are shadowed. See the R400 Shadows specification for a detailed explanation of the shadow buffer algorithm.

1. TC->MH->MC->RBM: The TC determines that it needs more shadow coverage values. The TC sends a command to the RBM that specifies the Zplane, the depth surface and the block of pixels within that surface. The TC sends its command over existing datapaths through the MH to the MC that hols the multi-sample color data. The MC routes the request to its assocated RB's Memory Interface block.
2. RBM->RBD: The RBM sends the RBD a tile request. The RBD buffers this in a logically separate queue from the tile data that it receives from the RBT, though it could be a reserved portion of the same physical queue. A separate logical queue is necessary so that the RBD can process the filter request without waiting for normal render requests that are currently in the queue to complete.
3. RBD<->MC: The RBD reads the depth data and stores it in the depth cache.
4. RBM->RBD: In parallel with the preceeding step, the RBM creates a sequence of quad commands that specify filtering the desired pixels. The RBM puts these into a logically separate queue from the quad commands that the RBD receives from the SX blocks, so that the RB can process filter requests ahead of pending render commands. This is the same queue that is used for all other texture requests to the RB.
5. RBD->RBM: The RBD computes the shadow coverage values and passes the results directly to the RBM. This data does not go into the depth cache.
6. RBM->MC->MH->TC: The RBM forwards the filtered pixels back to the TC, via its existing datapath to the MC and the existing datapath from the MC to the MH.

Shadow buffering requires computing and storing the shadow buffer at multiple levels of detail, so that the TC can read from a shadow buffer that has approximately the correct pixel spacing. Unlike ordinary texture maps, a lower resolution depth buffer is simply a decimated version of the original depth buffer. The following sequence may be used to compute a decimated depth buffer. Results of this operation are written to the depth buffer that is specified in the RB depth surface descriptor.

1. TC->MH->MC->RBM: The TC receives a command to read the depth surface in decimate mode. It sends a command to the RBM that specifies the depth surface to be decimated. The TC sends its command over existing datapaths through the MH to the MC that hols the multi-sample color data. The MC routes the request to its assocated RB's Memory Interface block.
2. RBM->RBD: The RBM sends the RBD a tile request. The RBD buffers this in a logically separate queue from the tile data that it receives from the RBT, though it could be a reserved portion of the same physical queue. A separate logical queue is necessary so that the RBD can process the filter request without waiting for normal render requests that are currently in the queue to complete.
3. RBD->RBM->MC: The RBD issues a read request for the depth source data, specifying that it is for a decimation command. It also issues a read request for the corresponding tile of the destination depth buffer.
4. MC->RBM: The MC returns the depth data to be decimated, but it is not written into the depth cache. Instead, the depth tile is processed by the RBM.
5. RBM->RBD: The RBM uses the source depth tile to create a series of quad and tile commands that it sends to the RBD quad queue as for the shadow buffer command. If the source depth tile is stored in Zplane format, the RBD sends the Zplanes to he RBD tile queue and sends the RBD quad queue commands to render those Zplanes at a decimated set of sample positions. If the source depth tile is stored as individual depth values, the RMB sends a a decimated subset of the depth values to the RBD tile queue and sends commands to render them to the RBD quad queue.

6. RBD->RBM: The RBD processes the decimation quad commands, which cause it to update the destination depth buffer in the depth cache. Eventually the RBD writes the depth data out of its cache into memory.

{Note: the decimation command could use tile read commands produced by the standard rendering process. It would still produce quad commands from the source depth data, which would replace the quad commands produced by the shader pipe.}

3. SX: Shader Export Block

The Shader Export Blocks (SX0 and SX1) store and process results exported from the shader pipe. There are six classes of exports that the SX processes:

Parameters: produced by vertex shader programs. Three per clock are sent to the shader pipes (SP).
Positions: produced by vertex shader programs and passed serially to the Clipper block (CL).
Colors: produced by pixel shader programs. Up to four different RGBA color buffers are used in the RBC.
Depths: optionally produced by pixel shader programs. If used, this replaces the Zplane in the RBD.
Multipass data: produced by either vertex or pixel shader programs. The RBC stores these to a linear list.
Addresses: each vertex or pixel shader clause can specify an address for the multipass data it outputs.

The figure below illustrates the busses and basic internal blocks for each SX. SX processing can be divided into four broad groups: parameter selection, quad processing, shader data processing, and the export buffer. SX0 inputs data from and outputs data to SP0, SP2, and SX1. SX1 inputs data from and outputs data to SP1 and SP3, and SX0. Both of them communicate with the SC, SQ and all four RBs. In this figure, and through the remainder of this section, SP0 and SP1 are called SPlo and SP2 and SP3 are called SPhi. SX# indicates the specific SX block.

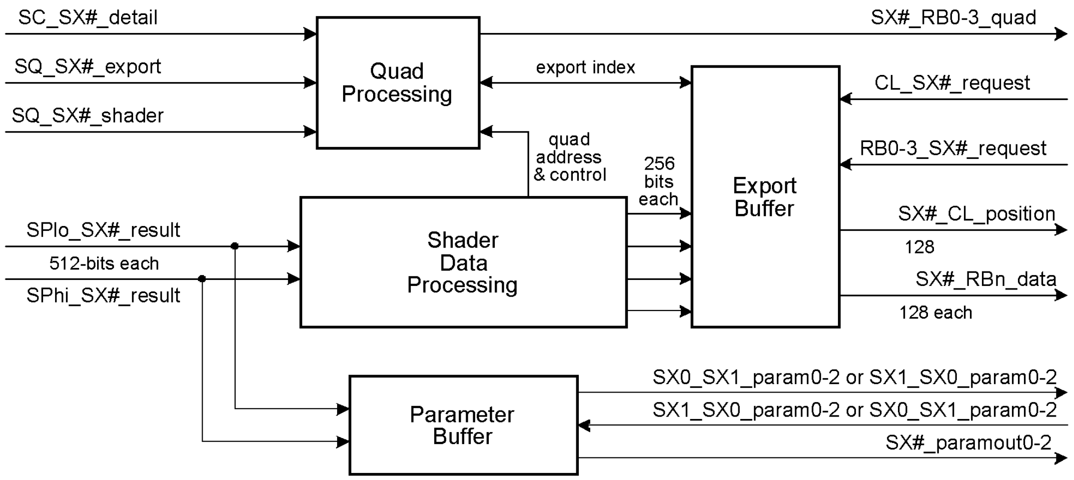


Figure 8: SX: Shader Export Block Diagram

The Shader Export block buffers the remaining types of export data, except for addresses. Buffered positions are passed to the Clipper (CL), buffered depths are passed to one of the four depth test blocks (RBD), and buffered colors and multi-pass data are passed to one of the color blend blocks (RBC). Multipass data is simply routed through the RB to be stored in memory without interpretation, at offsets from a specified address. This data may be used to store results from a high order surface program, a multi-pass vertex program, or a multi-pass pixel program. Each shader pipe must export an address per clause to specify where the RB stores multipass data.

3.1 SX Parameter Buffer Block

The SX parameter selection logic processes vertex parameter data, which the SX receives from its pair of shader pipes on the data busses. The SX caches recently computed parameters from vertex shader programs. In order to execute a pixel shader, each shader pipe needs to select three vertices from the caches, based on the current triangle, and interpolate their parameters. All four SP's receive the same parameters, since they all work on the same triangle at once.

Together, the two SX blocks contain 16 separate parameter caches, each of which stores 128 128-bit parameters. Each vertex can have up to 16 parameters, so this cache stores parameters for 128 or more vertices, depending on the number of parameters per vertex. A single vertex shader program computes parameters for 64 vertices, so the parameter cache can store the results of two or more vertex shader clauses. This cache seems generous in size, but there is a problem. The three vertices that form a triangle may have been computed in any of the vector units and therefore may be in any of the parameter caches. The SPs require three parameters on every clock, so the Sequencer (SQ) must ensure that the three vertices are stored in different caches. The SQ requests a value from three of the 16 caches on each clock. The SX Parameter Block performs the necessary multiplexing.

The figure below shows the internal structure of the SX Parameter Buffer Block. The SQ sends control bits to the SX to indicate when to store the SP result into the vertex buffer. Each SX then reads a parameter from each of its eight parameter caches and selects three of these values to send to the other SX block. Alternately, the SX reads three realtime parameters, which are loaded into a local memory over the register bus. The Final Mux block selects three parameters out of the three from this SX and the three from the other SX. Both SX blocks choose the same three parameters.

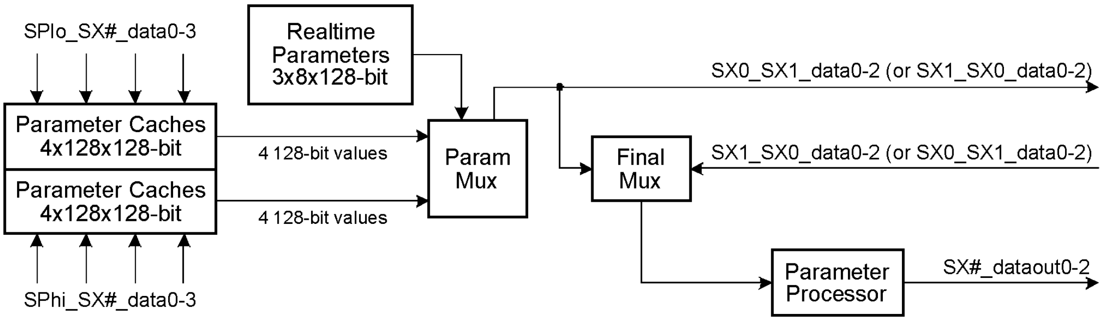

SPlo_SX#_data0-3
Realtime Parameters 3x8x128-bit
SX0_SX1_data0-2 (or SX1_SX0_data0-2)
Parameter Caches 4x128x128-bit
4 128-bit values
Param Mux
Final Mux
SX1_SX0_data0-2 (or SX0_SX1_data0-2)
Parameter Caches 4x128x128-bit
4 128-bit values
Parameter Processor
SX#_dataout0-2
SPhi_SX#_data0-3

Figure 9: SX: Parameter Selection Logic

The Parameter Processor performs three operations on the parameters before forwarding them to the shader pipes. First, it selects the provoking vertex parameter value for flat shading (performs parameter selection). Second, it calculates the delta differences between the parameter values at vertices (Parameter Difference). These deltas are then used as input values into the interpolator units. Finally, it performs a cylindrical wrap adjustment on the delta parameters (Cylindrical Wrap). All the functions are implemented on per channel basis. {The control/state signals are part of the "SQ_SP: Interpolator Bus" interface.}

{Note: We could insert a global block between the two SX blocks that does the final mux and forwards the results to the four shader blocks. This doesn't increase the number of global wires, though it may increase the wire length. It eliminates one of the two copies of the Final Mux block, but much more importantly, it provides a single location for inserting the realtime parameters and for including logib to perform several operations on the three parameters that otherwise would have to be performed in all four shader pipes.}

3.2 SX Shader Data Processing Block

On each clock, the Shader Data Processing block inputs four 128-bit RGBA values from each of two Shader Pipes. If they are export values (other than parameter exports), this sub-block processes the data and passes it on to be stored in the Export Buffer sub-block. It also extracts information from the export data and sends it to the Quad Control sub-block. The operation performed depends on the type of export data. The figure below shows the internal structure of the Shader Data Processing Block. The following text describes the processing that occurs for each type of export, except for parameter exports, which are described above.

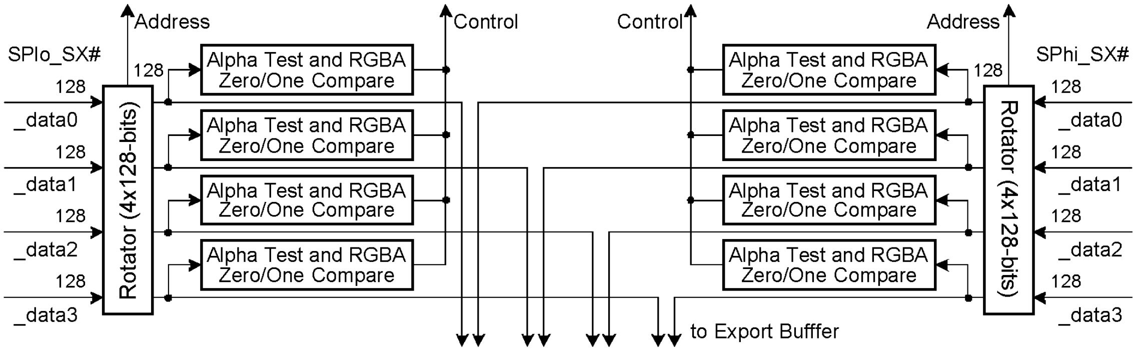

Address
Control
Control
Address
SPlo_SX#
128
_data0
128
_data1
128
_data2
128
_data3
128
Rotator (4x128-bits)
Alpha Test and RGBA Zero/One Compare
Alpha Test and RGBA Zero/One Compare
Alpha Test and RGBA Zero/One Compare
Alpha Test and RGBA Zero/One Compare
Alpha Test and RGBA Zero/One Compare
Alpha Test and RGBA Zero/One Compare
Alpha Test and RGBA Zero/One Compare
Alpha Test and RGBA Zero/One Compare
128
Rotator (4x128-bits)
SPhi_SX#
128
_data0
128
_data1
128
_data2
128
_data3
to Export Bufffer

Figure 10: SX: Shader Data Logic

A pixel shader may export one or more RGBA color vectors for rendering to the destination color buffer(s). The Alpha Test and Zero/One Compare blocks operate on this RGBA data. These blocks compare all four components to zero and one and also compare alpha to the Alpha Test value. The Quad Control sub-block uses the results of these comparisons to cull quads. Alpha test culling is defined in OpenGL and DirectX. The zero/one comparisons allow optimizations for blend modes that become no-ops if color or alpha components are equal to zero or one. {cite the register fields that control how this data is used.} The four 128-bit vectors that come from each SP are part of a single 2x2 quad. Four susbsequent clocks send four different quads.

There are two kinds of export data that pass through the Shader Data Block unchanged. The first type is depth data that may be exported from a pixel shader for use in rendering the pixels. As for rendered colors, the four vectors that come from a single SP on a single clock are part of the same 2x2 quad. The second type is position data, which vertex shaders output to specify the (X, Y, Z, W) position of a vertex, plus optionally a sprite width, sprite height, and edge flags.

Multi-pass vertex shaders may export vectors to memory, instead of to the parameter cache. In that case, the vertex shader must specify the address at which to write the vectors. This block forwards these address exports directly to the Quad Processing block, where they select the RB that writes the vectors to memory.

For all remaining export data, the data output from the four vector units of a shader are not related to each other and may be processed by different RBs. Therefore the SX block associates groups of four exports from the same vector unit for these exports. The rotator block rotates the vectors so that the export buffer can group them by vector unit. The following section on the Export Buffer Data Format describes this in detail.

Finally, note that the Alpha Test/Compare blocks only operate on unrotated vectors. Therefore, the Rotator can be placed either before or after the Alpha Test/Compare blocks, depending on implementation constraints.

{**Note**: The Shader Pipe can send up to four color values per pixel. Under DX9, alpha test only applies to the primary color. We may need to support something more general for DX10.}

3.3 SX Quad Processing Block

The SX Quad Processing Block is illustrated in the figure below. The Detail Buffer stores quad commands from the SC until the pixel shader program produces the corresponding color data. The size of the Detail Buffer (over both SX blocks) determines the maximum number of quads that can be in flight in the SP at the same time. The Quad Control and Quad Buffer blocks are described below.

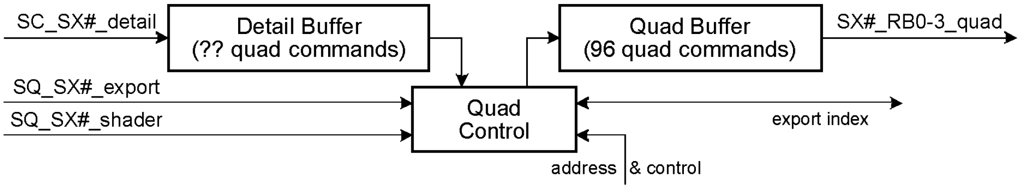


Figure 11: SX: Quad Processing Logic

The Quad Control block allocates data in the export buffer, tells the Sequencer (SQ) how much space is available in the export buffer, and puts quad commands into the Quad Buffer. A quad command specifies a quad for one of the RBs to process, including the sample mask, the quad's position within its tile, and the location of associated data in the export buffer. The Quad Control block operates on three types of data: positions, pixels, and pass-through data.

When the Quad Control block receives position data, it stores it in a reserved portion of the export buffer. Position data consists of either one or two 128-bit words per vertex, depending on the context. In addition to the (x,y,z,w) location of the vertex, the position can include a sprite width, sprite height, and edge flags. The export buffer can store four single-size or two double-size position data results, so that the export buffer can shift out one set of positions while a second set are being computed. A vertex shader that exports position data cannot start until there is room in the export buffer.

When the Sequencer tells the Quad Control block to start a pixel export clause, the Quad Control block allocates space in the Export Buffer for one to four colors per pixel and an optional depth value, depending on the context. It also allocates a space in the Quad Buffer. The Quad Control block then computes an address in the export buffer for each export, depending on the export instruction. The Quad Control block also removes the next quad command from the Detail Buffer, modifies the sample mask based on the comparisons performed by the Shader Data block, and writes the result into the Quad Buffer, together with the location of the pixel data in the export buffer.

The Quad Control block handles clauses that export pass-through data somewhat differently. As before, it allocates space for the export data in the export buffer, but this data does not have corresponding quad commands in the Detail Buffer. Instead, the Quad Control block allocates space in the Quad Buffer and generates quad commands for the pass-through data. Each such command specifies up to four exports from a single vertex and contains a device address in place of the sample mask. The Quad Control block uses a base address context register to generate these device addresses. For vertex exports, it generates the device address by adding the base address to a special address export, which is described in the preceeding section. For all other exports, it generates the device address by incrementing from the base address.

Finally, the Quad Buffer outputs the quad commands to the four RBs. Each RB gets one quad command every four clocks from each SX block, so the Quad Buffer only needs a single read port. {**NOTE:** There are two basic ways to distribute the quad commands. First, all quad commands could be sent to all RBs, with each RB taking the ones that it owns. This requires buffering for at least 32 quad commands per RB, to avoid stalling when one RB gets two full tiles before another RB gets any. Second, the Quad Buffer could send a separate stream of quad commands to each RB. This requires tracking the next quad command for each RB. Possibly this can tie into the means of figuring out how much free space exists in the export buffer.}

3.4 SX Export Buffer Logic

Finally, the Export Buffer block stores all export data, except for parameters, in eight 80x128-bit buffer memories, which are organized into four buffer pairs. Each buffer stores 16 words of position data and 64 words of other exports. Across both SX blocks, this is sufficient to buffer two vectors of 64 position data (XYZW and sprite size per position) plus 16 8x8 tiles of pixels (with one RGBA color per pixel).

Four of the eight buffers may be read per clock to send results to the RB blocks or the CL block (clipper), using the multiplexers in the figure below. The data must be swizzled so that the five output multiplexers each get one 128-bit word from each of four buffers. The precise storage formats are described in the following subsection. Each output multiplexor reads from buffer pair zero to three sequentially. The four RB output multiplexers are offset so that each occupies a non-competing time slice. The Clipper (CL) output multiplexor steals a time slice from one of the RBs whenever the Clipper needs more position data. The Buffer Read Control block accepts read requests from the CL and the four RBs and schedules whether the CL or an RB gets to use each time slice.

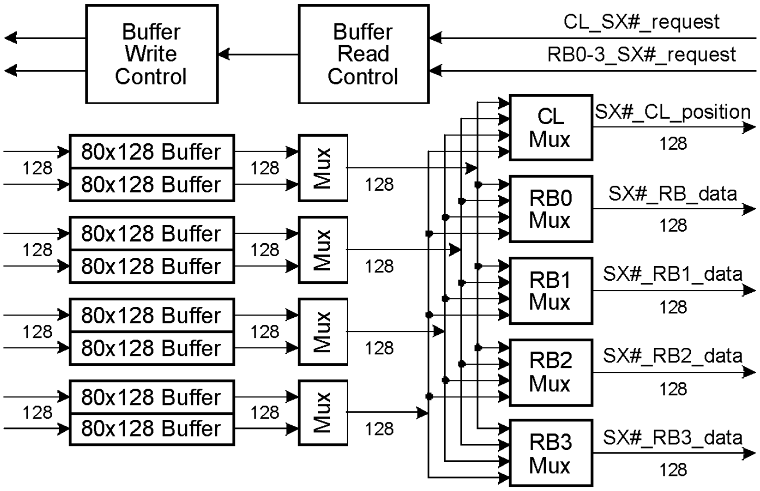


Figure 12: SX: Export Buffer Logic

The SX keeps track of free space in the buffers by storing a bit per access unit, which it sets when it loads data and clears when the RB accesses that data. This requires 64*2 bits per SX, since the RBs read four 128-bit words at a time and each SX has 64*8 128-bit words for non-position exports. The RB is able to skip reading data that it doesn't need by setting a Discard bit when it sends the SX a buffer address.

{**Note**: at one time the export data needed to be maskable by 32-bit words, since the shader instructions could specify which of the four components to write on an export command. This feature has been removed, so that each shader always exports four 32-bit components to the SX. However, the RB may be programmed to ignore some of the components.}

3.5 Export Buffer Data Format

The Export Buffer stores data as vectors of four 32-bit IEEE floating point values, with one exception that is described below. The data format in the export buffer is designed to allow writing shader export data into it at full speed and to allow the four RBs to read data out without interfering with each other. The exact format depends on whether the data is a pixel shader export or a vertex/debug export.

The figure below shows how pixel export data is mapped to the four pairs of export buffer memories. Each SP outputs four 128-bit vectors in parallel on each of four sequential clocks. A single 256-bit wide export buffer memory stores vectors from the two SPs that are associated with the SX block. SX0 receives data from SP0 and SP2 (the even SPs) and SX1 receives data from SP1 and SP3 (the odd SPs). All of the export formats store the corresponding data from the two SPs in the same buffer memory. These figures mark the shaders as SPlo and Sphi, which are SP0 and SP2 for SX0 or SP1 and SP3 for SX1. A single quad can store up to 5 contiguous vectors for one to four colors, followed by an optional vector containing a depth value.

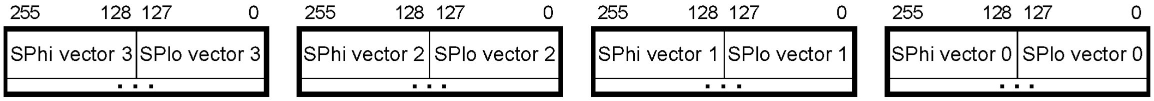


Figure 13: SX: Export Buffer Data Order

{Include another figure that shows how multiple colors and depth are stored. All of the exports for a given quad are stored at sequential locations, before the exports for the next quad. For R400, depth occupies a full 128-bits per pixel, even though only one component is used. If depth is exported, it occupies the first location, followed by 1-4 color buffers.}

The figure below shows how vertex and debug exports map to the four memory buffers. Vertex and debug exports are rotated depending on the export address. Given export address N, (N mod 4) determines the rotation. This is necessary since the four vectors that a single SP outputs in parallel are not related to each other for vertex exports, unlike pixel exports where they are the four pixels of a 2x2 quad. So for vertex exports, the SX groups together four exports from a single vector unit. The figure shows that each row consists of four values with the same vector number, at different values of N mod 4. This allows the RBs to read out either SPhi or SPlo from a single row to get 512-bits of contiguous data.

255 128	127 0	255 128	127 0	255 128	127 0	255 128	127 0
SPhi vector 0 N mod 4 = 3	SPlo vector 0 N mod 4 = 3	SPhi vector 0 N mod 4 = 2	SPlo vector 0 N mod 4 = 2	SPhi vector 0 N mod 4 = 1	SPlo vector 0 N mod 4 = 1	SPhi vector 0 N mod 4 = 0	SPlo vector 0 N mod 4 = 0
SPhi vector 1 N mod 4 = 0	SPlo vector 1 N mod 4 = 0	SPhi vector 1 N mod 4 = 3	SPlo vector 1 N mod 4 = 3	SPhi vector 1 N mod 4 = 2	SPlo vector 1 N mod 4 = 2	SPhi vector 1 N mod 4 = 1	SPlo vector 1 N mod 4 = 1
SPhi vector 2 N mod 4 = 1	SPlo vector 2 N mod 4 = 1	SPhi vector 2 N mod 4 = 0	SPlo vector 2 N mod 4 = 0	SPhi vector 2 N mod 4 = 3	SPlo vector 2 N mod 4 = 3	SPhi vector 2 N mod 4 = 2	SPlo vector 2 N mod 4 = 2
SPhi vector 3 N mod 4 = 2	SPlo vector 3 N mod 4 = 2	SPhi vector 3 N mod 4 = 1	SPlo vector 3 N mod 4 = 1	SPhi vector 3 N mod 4 = 0	SPlo vector 3 N mod 4 = 0	SPhi vector 3 N mod 4 = 3	SPlo vector 3 N mod 4 = 3
• • •		• • •		• • •		• • •	

Figure 14: SX: Export Buffer Data Order

Vertex shaders that export 1-4 vectors use the above format. Vertex shaders that export 5-8 vectors use two copies of the above format, one immediately following the other. Vertex shaders that export 9-12 vectors use three copies of the above format. Therefore, each vertex shader clause requires export buffer space for 0, 4, 8, or 12 exports. {**Note**: what about unaligned exports?}

The figure below shows the three data formats used in the export buffer. Normally, each 128-bit vector represents four 32-bit IEEE floating point values, which the RB may pack to a smaller format. The components are specified in little-endian order, with X or Red in the low order position and W or Alpha in the high order position. The SP may perform multiple exports to the same export address, using the word select to write different words.

127 96	95 64	63 32	31 0
W or Alpha (32-bit IEEE)	Z or Blue (32-bit IEEE)	Y or Green (32-bit IEEE)	X or Red (32-bit IEEE)

127 96	95 64	63 32	31 0
Depth (32-bit IEEE)	*unused*	*unused*	*unused*

127 102	101 96	95 6	5 64	63 38	37 32	31 6	5 0
Alpha (top 26-bits of 32-bit IEEE)	Fog 31:26	Blue (top 26-bits of 32-bit IEEE)	Fog 25:20	Green (top 26-bits of 32-bit IEEE)	Fog 19:14	Red (top 26-bits of 32-bit IEEE)	Fog 13:8

Figure 15: SX: Export Buffer Data Formats

The final portion of the above figure shows the format used to store an RGBA color that also includes a fog factor. The RGBA values come from the vector pipe and the fog factor comes from the scalar pipe. The SP drops the low order 6-bits of the RGBA components and the low order 8-bits of the scalar result, which is the fog factor. It then packs the 24-bit fog factor into the holes in the RGBA values. The RB rounds these truncated values to produce RGBA and a fog factor in its internal format.

4. RBT: Tile Logic Block

The Tile Logic Block processes 8x8 tiles from the Scan Converter. It can start a new 8x8 tile every four clocks. It loads the Tile cache with per-tile data, including hierarchical Z/stencil data and the depth and color compression modes. It uses this data and data from the Scan Converter to decide when to kill quads and tiles that fail the hierarchical tests. It also buffers data for use by the Depth Test block and allows the Depth Test and Color Blend blocks to update the data cached in the Tile Logic block. The Tile Logic Block ensures that tile data remains in the cache until the Depth Test and Color Blend blocks are done with it, then it writes out the tile data if it was modified.

The figure below illustrates the internal structure of the Tile Logic Block. The Tile Logic Block processes 8x8 tiles in two stages. Each stage steps through the list of 8x8 tiles provided by the Scan Converter. Stage one probes the Tile data cache for each tile and issues a read if that tile's data is not in the cache and does not have a read pending. It then steps to the next 8x8 tile without waiting for the read data to come back. Stage two waits until data is available in the Tile cache for each tile. Then it computes whether the 8x8 tile can be culled and passes that result to the Rasterizer. If the tile is not culled, stage 2 also passes the tile address to the Depth Test block, so that it can read the depth data.

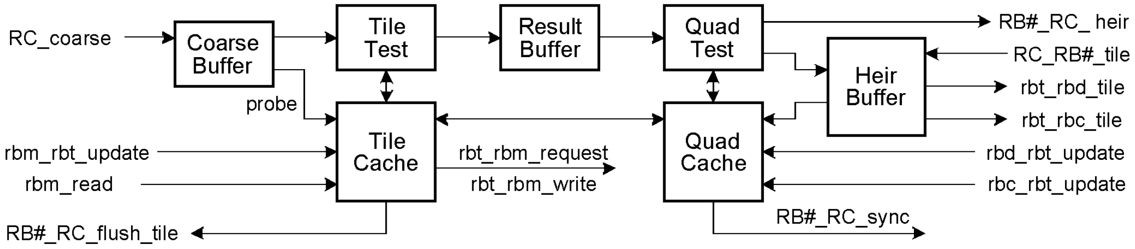


Figure 16: RBT: Tile Logic Block Diagram

The Tile Logic block also processes shadow depth buffer and multi-sample filter requests from the Texture Unit. Each such request to the Tile Logic consists of an 8x8 tile request that specifies which 8x8 tile of which shadow depth buffer to test or which 8x8 tile of which color buffer to filter. Shadow depth buffer and multi-sample filter requests use their own buffer memory so that they do not mix with 8x8 tile requests from the Scan Converter. They have priority for processing by the Tile Probe and Tile Cull stages.

Finally, the Tile Logic block also processes 8x8 tiles for multi-pass pixel processing. These requests come from the RB Parameter Buffer block. The Rasterizer does not send 8x8 tiles during multi-pass pixel processing, so these requests use the same queues that the Scan Converter uses. {**Note**: check whether this works.}

{NOTE: define in-flight counts and explain when they are incremented and decremented.}

4.1 Input Queues

The Tile Logic Block uses two physical input queues, which are each divided into two logical input queues. Each is implemented using a single port memory that operates on a four-clock cycle. In each queue, two of the four clocks are dedicated to writing data into the queues. The other two clocks are used to read data for use in one of the three Tile Logic stages. {**Note**: we could make them two-port memories to allow doubling the rate at which they can process 8x8 tiles. This is not necessary in the 4-pipeline system. Is it useful enough to justify the cost in the 2-pipeline system?}

The SC computes a pair of 8x8 tiles every other clock. These two 8x8 tiles are horizontally adjacent and are therefore always processed in separate pipelines. Therefore, the RB only needs to accept one 8x8 tile every other clock, at maximum. The fifo input logic gathers the dual block data, which is sent over two clocks, and selects which of the two blocks, if any, to write into the fifos. If the second block is selected, the fifo input logic also decodes the compressed dual-block format to find the second block. This requires nothing more than incrementing or decrementing the values specified for the first block.

The following lists describe the signals in the address queue and the depth queue. Each of the queues is used in two stages. The Tile Logic block reads the address queue during stages 1 and 2 and also reads the depth queue during stage 2. The Depth Test block reads the depth queue during stage 3.

Address Queue (stages 1 and 2, 24-bits):
Xaddr (10-bit): Specifies the X position of this 8x8 tile (1&2)
Yaddr (10-bit): Specifies the Y position of this 8x8 tile (1&2)
Context (3-bit): specifies which set of context parameters to use (1&2)
Covered (1-bit): specifies that the 8x8 tile is fully covered {**note**: may not be implemented} (2)

The address queue stores the (X, Y) address of the 8x8 tile. It also stores a 3-bit code that specifies the set of contect registers to use to process this 8x8 tile. Finally, it may also store a bit that specifies that the entire 8x8 tile is covered by a single triangle. {**Note**: this optimization probably won't be supported. It would allow the RB to skip reading the depth buffer if all depth tests pass, for example.}

Depth Queue (stages 2 and 3, 128-bits)
Zplane (88-bit): specifies the Zplane for this triangle in this 8x8 tile (2?, 3)
Context (3-bit): specifies which set of context parameters to use (3)
Direction (1-bit): specifies that the primitive is front or back facing (2)
PminDepth (10-bit): upper 10 bits of the minimum Z computed at the primitive vertices (2)
PmaxDepth (10-bit): upper 10 bits of the maximum Z computed at the primitive vertices (2)
CoarseMask (16-bit): specifies the 2x2 stamps that may contain valid samples (2 & 3?)

The depth queue stores information used in the Tile Logic Block in stage 2 and in the Depth Test block in stage 3. The Zplane allows the Depth Test logic to compute a depth value per sample. {**Note**: it could also be used in stage 2 to optimize the tile kill, if this proves worthwhile}. Zmin and Zmax specify the minimum and maximum Z values in the triangle, expressed as the upper 10-bits of the values in frame buffer format. Context is the same as the corresponding field in the address queue, and is repeated here to be available to stage 3. Direction distinguishes front-facing from back-facing primitives. Some operations differ depending on the triangle direction. Finally, CoarseMask provides a bit for each 2x2 quad in the 8x8 tile. If the bit is one, the corresponding quad may be required to process the tile. The Depth Test logic may use this to optimize reads for uncompressed depth and color data. {**Note**: the Tile Logic block could also use CoarseMask to optimize till killing.}

4.2 Stage 1: Issue Tile data Read

The first stage processes the 8x8 tiles passed by the SC. This stage probes the Tile Data cache and issues a read to load the cache, if necessary. Each line in the Tile Data cache stores information for sixteen 8x8 tiles. Each tile uses a 32-bit tile data word as illustrated in the figure below.

31 12	11 8	7 6 4	3 2 0
Zrange	Smask	Zmask	Cmask

Figure 17: RBT: Tile Logic Data Format

Cmask and Zmask specify the compression format for the color and depth buffers for each 8x8 tile. Zrange is used for hierarchical depth testing. Smask stores 3-bits of hierarchical stencil information and a bit for stencil compression. The Memory Format Specification describes the color, depth and stencil compression formats. Later sections of this document describe their formats as stored in the RB caches. The Depth Blend and Color Blend blocks update their fields when they modify color and depth/stencil data in their caches.

StencilCode, TminDepth and TmaxDepth describe the range of stencil and depth values in the 8x8 tile. These are used for hierarchical culling of 8x8 tiles. The following subsections describe the details of how these fields are used. These fields need not be read if hierarchical culling is disabled. They need not be written if depth and stencil updates are disabled. The Color Blend block updates these fields when it writes an 8x8 tile to the depth buffer.

Each Tile Data cache line also has a dirty bit and sixteen Pending counts. Stage 1 increments the Pending count for each 8x8 tile that it processes. The Color Blend block decrements the Pending count as processing finishes for each 8x8 block. The Tile Data cache cannot flush a line until all of its Pending counts are zero. This ensures that the Tile Data remains in the cache until it is no longer required by the Depth Test and Color Blend blocks. Before flushing a Tile Data cache line, the 16 tiles that it describes must also be flushed, since writing them to memory can alter the values in the Tile Data cache.

{**Note**: Mention the quad data cache, which provides finer granularity for hierarchical tests without requiring more memory bandwidth. The values would all be the same when the Tile Data cache line is first loaded, but could be different after receiving updates from the Depth Blend logic.}

4.3 Stage 2: Perform Z Kill and Probe Depth Cache

The second stage reads through the queue of 8x8 tiles in parallel with stage 1, though it cannot pass stage 1. For each tile, stage 2 waits until data is available in the Tile data cache. Then it performs three actions. First, it compares the StencilCode for the 8x8 tile with the stencil operation specified on this 8x8 tile, to see if the stencil operation passes, fails, or both. Second, it compares the PminDepth or Zmax for the 8x8 tile against the Zcull limit to determine if this 8x8 tile is guaranteed to fail the depth test. As a result of these tests, it sends the Rasterizer a bit to indicate whether the whole 8x8 tile can be killed. If the 8x8 tile is not killed, it sends the tile address and coarse mask to the depth cache, so that the block can be read into the cache if it is not already present.

{**Open issue**: Is it worthwhile to use the Zplane for the tile in the Zcull test? Also, the tile logic could compute pass/kill on a 2x2 quad basis. It isn't clear that this is worthwhile.}

{**Note**: The tile logic could also perform a test to determine whether any data needs to be read from the depth buffer. The depth buffer does not need to be read if either the entire tile is covered by the Zplane and the depth test passes at all samples, or if the tile is stored Uncompressed and hierarchical Z can determine whether every sample either passes or fails. At present, this doesn't seem worth computing.}

4.3.1 *Stage 2a: Tile Stencil Test*

The tile data contains a 4-bit Smask for each 8x8 tile that consists of a high order compression bit and a 3-bit Stencil code. The code specifies the range of stencil values in the 8x8 tile, relative to a Compare value in the range [1..254]. The stencil code is updated by the depth logic as stencil tests are performed and the stencil buffer changes. They are also modified by the hierarchical stencil logic is HierStencil is enabled. The stencil test involves a reference value, a comparison operation, three update actions, and 8-bit masks that select the bits to compare and the bits to update in the frame buffer. Hierarchical stencil testing must be disabled unless both masks are equal to 0xFF.

The following table describes the eight values for the 3-bit stencil code. The Stencil code specifies whether any stencil values in the 8x8 tile are equal to, less than, or greater than the Compare stencil value. Clear (000) is a special choice that indicates that the stencil values all have a Clear value, e.g. as the result of a fast-clear operation. When an 8x8 tile that has this code must be read from the frame buffer, the stencil cache is filled with the Clear value and the stencil code is set to Equal, Less, or Greater, depending whether Clear is =, <, or > the Compare value.

><=	Name	Description
000	Clear	Replace all stencil values with Clear value when reading the tile from memory
001	Equal	Compare value is equal to all stencil values (all equal, none less or greater)
010	Less	Compare value is less than all stencil values (all less, none equal or greater)
011	Lequal	Compare value is greater than no stencil values (some less or equal, none greater)
100	Greater	Compare value is greater than all stencil values (all greater, none less or equal)
101	Notequal	Compare value is equal to no stencil values (some less orgreater, none equal)
110	Gequal	Compare value is less than no stencil values (some less or greater, none equal)
111	Unknown	Nothing is known about how the Compare value compares to the stencil values

Table 1: Stencil Codes for 8x8 Tiles

The stencil test compares a Reference value to a stencil value, first ANDing both values with an 8-bit mask. The mask will be ignored here since it must equal 0xFF for hierarchical stencil testing to be enabled. The stencil code may be used to determine whether the stencil test passes, fails, or may do both over a tile of quad. This result depends on the stencil function and on whether the Reference value is equal to, less than, or greater than the compare value. Hierarchical stencil checking is most effective when the reference value equals the compare value.

The following tables show the result of the eight stencil tests for the seven stencil codes, depending on the relationship between the Reference value and the Compare value. “Pass” and “Fail” mean that all stencil comparisons in the tile pass or fail, whereas “both” means that both pass and fail results may occur in the 8x8 tile or 2x2 quad. Note that the Clear code is not listed since a tile with the Clear code was converted to either Equal, Less, or Greater.

Stencil Code	Stencil Function: reference stencil value OP tile stencil values, Reference==Compare							
	Never	Always	Less	Lequal	Equal	Gequal	Greater	Notequal
Equal	Fail	Pass	Fail	Pass	Pass	Pass	Fail	Fail
Less	Fail	Pass	Pass	Pass	Fail	Fail	Fail	Pass
Lequal	Fail	Pass	Both	Pass	Both	Both	Fail	Both
Greater	Fail	Pass	Fail	Fail	Fail	Pass	Pass	Pass
Notequal	Fail	Pass	Both	Both	Fail	Both	Both	Pass
Gequal	Fail	Pass	Fail	Both	Both	Pass	Both	Both
Unknown	Fail	Pass	Both	Both	Both	Both	Both	Both

Table 2: Stencil Function Pass/Fail for Reference==Compare

Stencil Code	Stencil Function: reference stencil value OP tile stencil values, Reference < Compare							
	Never	Always	Less	Lequal	Equal	Gequal	Greater	Notequal
Equal	Fail	Pass	Fail	Fail	Fail	Pass	Pass	Pass
Less	Fail	Pass	Both	Both	Both	Both	Both	Both
Lequal	Fail	Pass	Both	Both	Both	Both	Both	Both
Greater	Fail	Pass	Fail	Fail	Fail	Pass	Pass	Pass
Notequal	Fail	Pass	Both	Both	Both	Both	Both	Both
Gequal	Fail	Pass	Fail	Fail	Fail	Pass	Pass	Pass
Unknown	Fail	Pass	Both	Both	Both	Both	Both	Both

Table 3: Stencil Function Pass/Fail for Reference < Compare

Stencil Code	Stencil Function: reference stencil value OP tile stencil values, Reference > Compare							
	Never	Always	Less	Lequal	Equal	Gequal	Greater	Notequal
Equal	Fail	Pass	Pass	Pass	Fail	Fail	Fail	Pass
Less	Fail	Pass	Pass	Pass	Fail	Fail	Fail	Pass
Lequal	Fail	Pass	Pass	Pass	Fail	Fail	Fail	Pass
Greater	Fail	Pass	Both	Both	Both	Both	Both	Both
Notequal	Fail	Pass	Both	Both	Both	Both	Both	Both
Gequal	Fail	Pass	Both	Both	Both	Both	Both	Both
Unknown	Fail	Pass	Both	Both	Both	Both	Both	Both

Table 4: Stencil Function Pass/Fail for Reference > Compare

The stencil function selected to update a stencil value depends on the result of the depth test, as well as the result of the stencil test. Therefore, the updated 3-bit stencil code cannot be computed until after performing the depth test.

4.3.2 *Stage 2b: Tile Depth Test*

If the stencil test is guaranteed to fail, then the depth test need not be performed, since a failing stencil test prevents color and depth from being updated. If the stencil test is either guaranteed to pass or may either pass or fail, then the hierarchical depth test must be performed if HierDepth is enabled.

The tile data contains a 20-bit Zrange value for each 8x8 tile that decodes to two 14-bit depth values that are called TminDepth and TmaxDepth. Each is specified as a 0.14 fixed point value, with TminDepth implicitly zero extended to increase its precision and TmaxDepth implicitly extended with ones to increase its precision. Therefore the range of depth values represented is the half-open interval [TminDepth .. TmaxDepth+2^-14). Depth values are clamped to the half-open interval [0..1) for purposes of depth testing.

The depth test compares a depth value from the triangle against the depth value at each sample point. The SC sends RB two depth values, called PminDepth and PmaxDepth, which are the upper 10 bits of the smallest and largest vertex depth values for the primitive being rendered. The following table shows whether the depth test passes, fails, or may both pass and fail on the samples within the 8x8 tile. If all depth tests fail, the 8x8 tile can be killed. If all depth tests pass, then the RB does not need to perform depth comparisons for the 2x2 quads in this 8x8 tile.

Depth	**Depth Function: primitive depth values OP tile depth values**							
Comparison	**Never**	**Always**	**Less**	**Lequal**	**Equal**	**Gequal**	**Greater**	**Notequal**
Pmax < Tmin	Fail *Tmin:Tmax*	Pass Pmin:*Tmax*	Pass Pmin:*Tmax*	Pass Pmin:*Tmax*	Fail *Tmin:Tmax*	Fail *Tmin:Tmax*	Fail *Tmin:Tmax*	Pass Pmin:*Tmax*
Pmax ≥ Tmin Pmin < Tmin Pmax ≤ Tmax	Fail *Tmin:Tmax*	Pass Pmin:*Tmax*	Both Pmin:*Tmax*	Both Pmin:*Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both Pmin:*Tmax*
Pmin < Tmin Pmax > Tmax	Fail *Tmin:Tmax*	Pass Pmin:Pmax	Both Pmin:*Tmax*	Both Pmin:*Tmax*	Both *Tmin:Tmax*	Both *Tmin*:Pmax	Both *Tmin*:Pmax	Both Pmin:Pmax
Pmin > Tmin Pmax < Tmax	Fail *Tmin:Tmax*	Pass *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*
Pmin ≤ Tmax Pmin > Tmin Pmax > Tmax	Fail *Tmin:Tmax*	Pass *Tmin*:Pmax	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin:Tmax*	Both *Tmin*:Pmax	Both *Tmin*:Pmax	Both *Tmin*:Pmax
Pmin > Tmax	Fail *Tmin:Tmax*	Pass *Tmin*:Pmax	Fail *Tmin:Tmax*	Fail *Tmin:Tmax*	Fail *Tmin:Tmax*	Pass *Tmin*:Pmax	Pass *Tmin*:Pmax	Pass *Tmin*:Pmax

Table 5: Depth Test Pass/Fail and Tmin/Tmax Update

TminDepth and TmaxDepth may need to be updated as a result of the depth test. If the depth test does not fail and depth updates are enabled, then either TminDepth must be set to min(TminDepth,PminDepth) or TmaxDepth must be set to max(TmaxDepth,PmaxDepth), or both, depending on the comparison mode. The table above shows when to replace TminDepth and TmaxDepth with PminDepth and PmaxDepth. This over-estimates the effect of the depth test on the 8x8 tile, but is guaranteed not to kill an 8x8 tile that might pass the depth test. It also allows changes in the depth comparison mode without flushing the pipe. Accurate values for TminDepth and TmaxDepth are computed by the RB Write block, when the 8x8 depth tile is written to memory.

More accurate values for TminDepth and TmaxDepth would result if the RB waits until performing the per-quad depth test to update TminDepth and TmaxDepth with min(TminDepth,QminDepth) and max(ZaxTile,QmaxDepth). This method does not generate false kills, provided that the shader pipe is flushed when the depth comparison function changes. But it also does not improve the number of kills that can be generated

4.3.3 *Stage 2c: Tile Stencil Update*

Finally, the stencil value is modified using one of three stencil operations, depending whether the stencil test fails (Sfail), the stencil test passes and the depth test fails (Zfail) or both pass (Zpass). The following equations determine which of the three operations may apply within the tile or quad:

```
May_Sfail   = stencil_fail | stencil_both
May_Zfail   = (stencil_pass | stencil_both) & (depth_fail | depth_both)
May_Zpass = (stencil_pass | stencil_both) & (depth_pass | depth_both)
```

The following table shows how to modify the stencil code for the 8x8 tile, based on applying a single stencil op. Italics indicate cases where the stencil code and stencil values are nnot modified. If the stencil needs to be modified, then the stencil data must be loaded into the depth cache. If the stencil does not need to be modified, then the stencil data does not need to be loaded. Note that Compare is in the range [1..254], so it is never equal to either 0 or 255. Also note that the Invert stencil operation depends on whether the high order bit of Compare is 1 or 0.

Stencil Code	Stencil Operation for Sfail, Zfail, or Zpass								
	Keep	Zero	Replace	Incr Clamp	Decr Clamp	Incr Wrap	Decr Wrap	Invert cmp<7>==1	Invert cmp<7>==0
Equal	*Equal*	Less	*Equal*	Greater	Less	Greater	Less	Less	Greater
Less	*Less*	Less	Equal	Lequal	Less	Unknown	Unknown	Unknown	Greater
Lequal	*Lequal*	Less	Equal	Unknown	Less	Unknown	Unknown	Unknown	Greater
Greater	*Greater*	Less	Equal	Greater	Gequal	Unknown	Unknown	Less	Unknown
Notequal	*Notequal*	Less	Equal	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown
Gequal	*Gequal*	Less	Equal	Greater	Unknown	Unknown	Unknown	Less	Unknown
Unknown	*Unknown*	Less	Equal	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown

Table 6: Stencil Op Results for 8x8 Tiles

If a single one of May_Sfail, May_Zfail, and May_Zpass is true, apply the corresponding stencil op and update the 3-bit stencil code according to the table above. If more than one of May_Sfail, May_Zfail, and May_Zpass is true, compute an updated stencil code for each valid case, then OR the results together. For example, if May_Zfail and May_Zpass are both true, and the stencil op for one produces Equal and the stencil op for the other produces Greater, then the result would be Gequal, since resulting stencil values may be either equal to or greater than the compare value..

5. RBD: Depth Test Block

The Depth Logic processes 2x2 stamps within an 8x8 tile. In the normal mode of operation, the Zplane for the 2x2 stamps comes from slope that is stored in the 8x8 tile. The Depth Logic tests each 2x2 against the Zmin/Zmax to see if depth comparisons even need to be performed. It also updates Zmin and Zmax based on the samples that pass each 2x2 Z test. The second stage also issues depth and color reads for any blocks that pass the Zcull test and are not currently in the depth and color caches. For uncompressed 8x8 tiles, this stage only reads the 2x2 stamps specified by the Coarse field. This field specifies the 2x2 blocks that the SC determines might contain a valid sample. This allows the RB and the RS to process only the potentially useful portions of the 8x8 tile. For compressed tiles it is usually necessary to read or write the entire tile. A register will allow a threshold for the amount of compression that must be possible before the RB writes out tiles in a compressed format.

{**open issue**: R300 does w buffering: why & do we need it? If so, the shader could compute it. Define a 24-bit floating depth format.}

There is a shader mode that computes a Z value for each pixel. In that case, four Z values come from the shader pipe. They can be treated as individual Z values that each apply to all samples of their pixel. Alternately, the Depth Logic can produce a Zplane from the four Z values. This allows individual samples to have separate Z values that approximate the shape determined by the the Z values computed at the center of each pixel.

Perform stencil test on 2x2 stamp, if necessary
Perform Zpass test on 2x2 stamp
Perform depth comparison on 2x2 stamp, if necessary
Update Zmin and Zmax for 8x8 tile, if necessary
Update 8x8 tile depths, if necessary
Pass sample update and empty masks to the Color Logic

Z values are also specified per-pixel for shadow buffer tests. The Depth Logic converts the four Z values into a Zplane that it uses to test the samples within the 8x8.

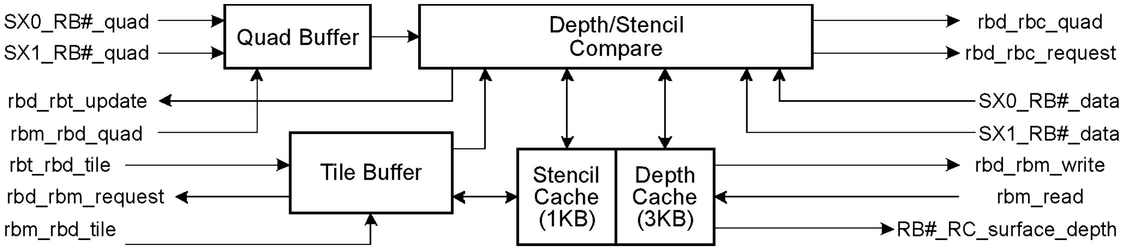


Figure 18: Depth Test Block Diagram

5.1 Input FIFOs

The Depth Logic Block uses two input fifos. Each is implemented using a single port memory that operates on a four-clock cycle. In each of these fifos, two of the four clocks are dedicated to writing data from either this pipeline's Rasterizer (RS) or from the Texture Unit (TU). The other two clocks are used to read data for use in one of the three Tile Logic stages.

The RS receives 8x8 tiles from the SC and culls them in two ways. It culls out tiles that are guaranteed to fail the Z test. This test is actually performed in the RB, which passes Zcull pass/fail bits back to the RS. The RS also culls out tiles that have no valid samples. {The RS needs to produce a valid tile every four clocks; I assume here that it can pass a tile to the RB every two clocks. There is a straightforward solution if the RS needs to pass RB a tile on every clock.} The following list describes the fifo that is written by the RS.

Stamp Queue (stages 2 and 3, written by RS)
16-bit Stamp mask: specifies the 2x2 stamps that will be processed by the shader pipeline
1-bit Covered: specifies whether the entire 8x8 tile is covered by this triangle {**is this needed?**}
7-bit {or less} Index: specifies the corresponding entry in the Zplane and Tile data queues

{Note: the above is replaced by a queue of individual 2x2 stamps. Each has an Index, the address of the 2x2 stamp within the 8x8 block, and a 32-bit mask of the covered samples in this 2x2 stamp.}

There is also a fifo that is written by the Texture Unit (TU). The TU sends a single bus to all four RBs, which can write each of them on one clock out of four. This leaves three clocks for the RB to read from the queue.

Ztexture Queue (stages 1, 2 and 3, written by TU):
20-bit Address: specifies XY position of the 8x8 tile
3-bit Context: specifies which set of context parameters to use
96-bit Zdata: Either four 24-bit Z values or a 56-bit Zplane, 16-bit stamp mask
16-bit Stamp mask: one bit per 2x2 stamp specifies whether it needs to be processed

{**Open issue**: When the shader pipe computes per-pixel Z values, how should we pass them to the RB? Under this scheme, when the shader computes per-pixel Z values, they are passed to the TU. The TU either passes these four values to the RB or changes them to

5.2 Stage 3: Depth/Stencil Tests

The third stage of the RB performs depth and stencil tests on 2x2 stamps. The RB determines the stamps to process based on 8x8 tile commands from the Rasterizer. The rasterizer only sends 8x8 tiles that contain at least one valid sample and that pass the Zcull test. Each such command contains a pointer to Zplane data that was passed by the SC. Each 2x2 operation stalls until

{Talk about per-pixel depth values. In this case, the stage stalls until depth data is available from the shader pipe, via a second fifo. Also talk about shadow buffer tests.}

6. RBC: Color Blend Block

The Color Logic inputs a 2x2 stamp of color data from the shader pipe. It also inputs a mask of the samples in that stamp that pass the depth test from the Depth Logic. The Color Logic performs fog blending and alpha blending on the dest colors.

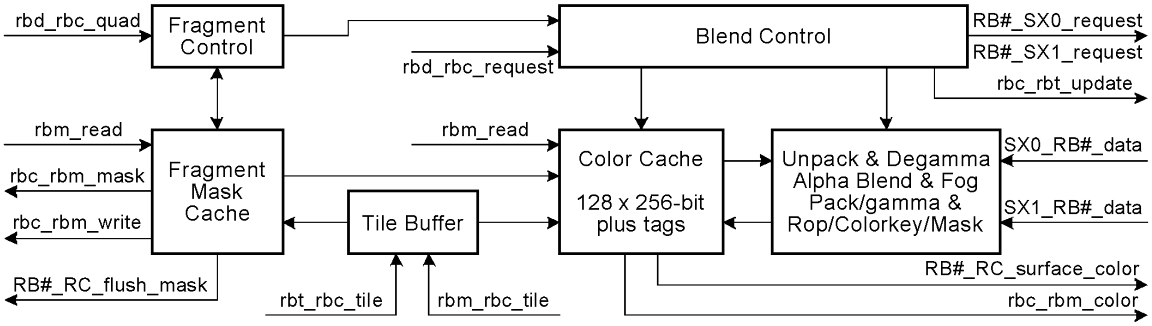


Figure 19: RBC: Color Blend Block Diagram

The Color Logic also supports filtering anti-aliased pixels. This occurs on requests from the Texture Unit.

Finally, the Color Logic supports two bypass modes. These output 2x2 stamps into a list of 512-bit entries, for use in multi-pass rendering. This is described in the following section.

Output four 32-bit floats per pixel in vertex or shader bypass modes, Else:
Compress 32-bit floats to 20-bit floats for RB processing
Output four 20-bit floats per pixel if RB bypass threshold met, Else:
Read 2x2 stamp from Color cache, if necessary
Perform fog and alpha blend multiplies, if necessary
Convert 20-bit floats to frame buffer pixel format
Perform rasterop, if necessary, and write result to the Color cache

6.1 Input FIFOs

The Color Logic Block uses three input fifos. The first is the fifo of 8x8 tile data from the Rasterizer, described above. The second fifo buffers results from the Depth Logic Block. The third fifo buffers pixels that are computed by the shader pipe.

{List the data passed from the Depth Logic. This includes a Pass bit per sample (this sample passed the depth test) and an Empty bit per sample (no depth value is stored for this sample). The Color Logic must drop the color at a sample if the Depth Logic has dropped the depth value for a sample and therefore cannot determine if it would pass the depth test. This avoids multi-pass bugs due to dropping samples.

{Talk about filtering multi-sampled pixels and passing them to the texture unit.}

6.2 Stage 4: Color Processing

{}

6.3 Fragment Processing

The first step in processing a quad in the RB Color Blend block involves determining what fragment operations to perform. A fragment is a color together with a mask that specifies which samples it is visible at within a pixel. With single-sample pixels, there is exactly one fragment per pixel, so each pixel is either processed or not, depending on whether the source pixel is valid. With multi-sample pixels, the process is more complex, since the destination pixel can contain multiple fragments and the source pixel may overlap one or more of them.

`The first step for fragment processing is to determine how the source pixel's sample mask intersects the sample mask of each fragment in the pixel. The following list defines the alternatives, where Ms and Md are the source pixel's sample mask and the sample mask of a particular fragment of the dest pixel.

Empty	$Ms \cap Md = \phi$	The two masks do not intersect
Cover	$Ms \supseteq Md$	The dest mask is covered by the source mask
Overlap	$Md \neq Md - Ms \neq \phi$	The source mask covers part of the dest mask

Table 7: Fragment Intersection Types

The next step determines what blend operations to perform. There are two different sets of rules for fragment processing, depending on whether the blend operation is write-only or read-write. A write-only operation depends only on the source color at each sample, whereas a read-write operation produces a result that depends on both the source color and the dest color at each sample.

The lists below specify the write-only and read-write operations to perform on each fragment. In the write-only case, the operation produces one new fragment and removes zero or more existing fragments. In the read-write case, the operation does not remove any fragments and can potentially double the number of fragments, up to the total number of samples per pixel. The fragment processing logic updates the fragment data and passes the specified blend operations to the alpha blender.

Empty	Ignore this dest fragment
Cover	Delete this dest fragment
Overlap	Replace the dest fragment mask with Md – Ms
(also)	Create a fragment with mask Ms, blended using the source color

Table 8: Fragment Rules for Write-Only Blending

Empty	Ignore this dest fragment
Cover	Blend the source color into the dest color at this fragment
Overlap	Replace the dest fragment mask with Md – Ms and create a new fragment with mask Ms – Md by blending the source and dest colors

Table 9: Fragment Rules for Read- Write Blending

There is one additional complication involved in allocating fragments. The blend logic uses two clocks to compute the four pixels of a quad, so it actually computes two pixels in parallel. This implies that fragment allocation must occur for pairs of pixels instead of for single pixels. When creating a new fragment for a pair of pixels, the fragment logic must distinguish three cases, based on the sample masks Md0 and Md1 for the pair of pixels being written

Both	$Md0 \neq \phi$	& $Md1 \neq \phi$	Allocate this pair of pixels to an empty fragment-pair
Left pixel.	$Md0 \neq \phi$	& $Md1 = \phi$	Search for a fragment-pair that has an empty mask for the lefthand If found, modify the lefthand pixel and leave the righthand pixel unchanged. Else, allocate this pair of pixels to an empty fragment-pair.
Right pixel.	$Md0 = \phi$	& $Md1 \neq \phi$	Search for a fragment-pair that has an empty mask for the righthand If found, modify the righthand pixel and leave the lefthand pixel unchanged. Else, allocate this pair of pixels to an empty fragment-pair.

If the above steps are not performed, it is possible for a tile with S samples per pixel to contain up to 2S fragments, since some pixel-pair could have up to S fragments that touch a sample of the lefthand pixel but leave the righthand pixel empty, followed by another S fragments that touch a sample of the righthand pixel but leave the lefthand pixel empty. There is only room in the cache and in memory for up to S samples, so the algorithm must prevent this case from occurring.

7. RC: Read Data Block

8. RBW: Write Data Block

9. External Interfaces

Name	Direction	Bits	Description
RTRn	MC → RB	1	Ready to receive a new RB data request
DataSelect	RB → MC	8	Specifies a 256-bit entry or specifies that this request should be ignored

Table 10: RB DataRequest Interface

{}

9.1 Scan Converter Interface

{}

96-bit Zplane
16-bit Coarse and Covered masks
10-bit Zmin, Zmax
10-bit Xtile, Ytile
3-bit Context
1-bit Background (saves comparing Zplane to background)
1-bit Direction (front or back facing

9.2 Rasterizer Interface

{}

16-bit Coarse and Covered masks to the RE for HiZ elimination
16-bit Coarse and Covered masks from the RE
32-bit sample mask from the RE

9.3 Texture Unit Interface

{}

9.4 Shader Pipe Interface

{}

9.5 Memory Controller Interface

{}

10. Register Interface

The RB supports multiple contexts so that the CP can issue commands that use a different context without first flushing the pipe. The context is identified by a 3-bit index that is associated with each 8x8 tile issued to the RB by the Scan Converter (SC). {How many contexts?} {I assume here that the RB uses a single sub-context. The RB context could be split into multiple pieces if that is necessary, but this must be defined before the RB interfaces, since that would require multiple context indices to be associated with each tile.}

10.1 Surface Descriptors

There are {16 to 32} surface descriptor registers, which are similar to the base address registers in the Memory Hub. These registers store all of the surfaces in use by all of the active contexts. Context-dependent registers store indices into the surface descriptor array. Each surface in the array must be different, so that when multiple contexts use the same surface, they all use the same index. This allows the RB to keep its cache coherent by tagging each cache line with the index and an (x,y) position.

{Describe the surface descriptors, which include the following fields.}
base address
pixel size and format
endian control: no swap, word swap, dword swap {there is a fourth one}

dimensionality
width
height
low order Z bits if this is a 2D slice from a 3D array.
Address format: tiled in local memory, tiled in system memory, or linear in system memory.
Samples per pixel (1, 4, or 8)
number of bytes an 8x8 tile occupies in memory

{Is the color format an itemized list, or a more generic description of component sizes and positions, within certain limits? I prefer the latter.}

10.2 Depth Test Registers

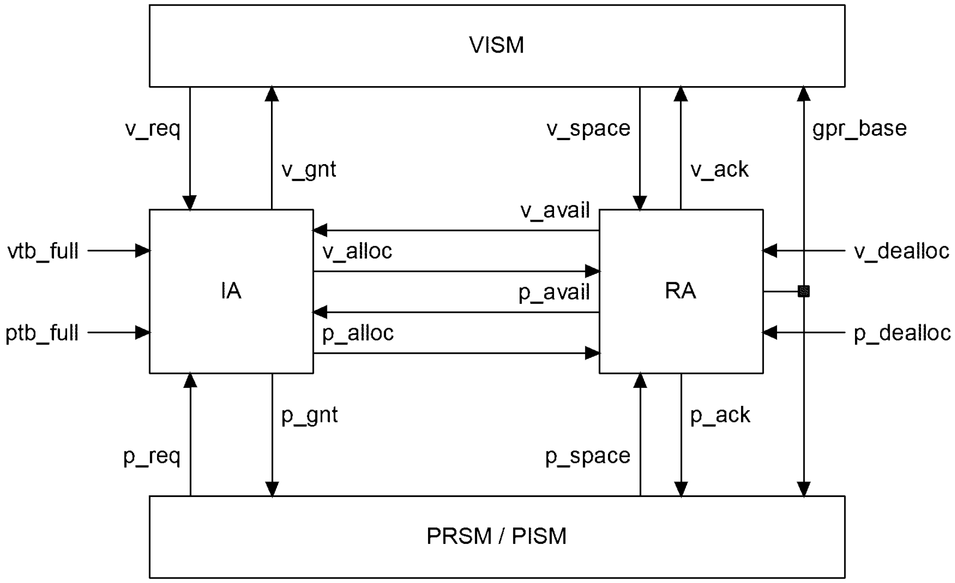
Enable HiZ (possibly both 8x8 and 2x2)
Enable HiS (possibly both 8x8 and 2x2)
Enable shadow buffering
Enable per-pixel depth from the shader pipe
Stencil reference, base, and background values
Depth background value
Depth and control fields from OpenGL
{Do we need to select round or truncate for producing FB depth values?}
Depth {and maybe stencil} surface indices
MC address queue for each depth/stencil surface
{Should we support a second depth buffer?}

10.3 Color Blend Registers

Alpha test threshold
Alpha test control fields
Blend constant register
Fog factor enable (uses the blend constant)
Plane mask (pattern should repeat on pixel size, for good performance enable/disable whole bytes)
Enable multiple color buffers
Four color surface indices
Dither mode: truncate (add 0), round (add 0.5), dither {error diffusion?} Dither offsets?
Rop mode {do we have to directly support Rop3?}
MC address queue for each color surface
Enable tile filtering

10.4 Other Registers

Stipple pattern {?}
Multisample mode
Vertex buffer index {more than one?} and mode
Something about cache format, flushing, etc.?
Status bits (clean caches, RB idle, etc.)
Compression mode enables: force writes to be uncompressed or to not use certain modes
Enables for various optimizations (to let us turn them off)
Performance counters


VISM
v_req
v_gnt
v_space
v_ack
gpr_base
v_avail
vtb_full
v_alloc
v_dealloc
IA
p_avail
RA
ptb_full
p_alloc
p_dealloc
p_gnt
p_ack
p_req
p_space
PRSM / PISM

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	26 December, 2001	[date \@ "d MMMM,	GEN-CXXXXX-REVA	1 of 34

Author: Vic Romaker

Issue To:	**Copy No:**

SQ Hardware Specification

v 0.1

Overview: Details of r400 sequencer (SQ): block diagrams, interface timing diagrams, implementation.

AUTOMATICALLY UPDATED FIELDS:
Document Location: Document2
Current Intranet Search Title : Go to "File -> Properties -> Summary" to set title name

APPROVALS	
Name/Dept	Signature/Date

Remarks:

THIS DOCUMENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATI TECHNOLOGIES INC. THROUGH UNAUTHORIZED USE OR DISCLOSURE.

"Copyright 2000, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished work created in 2000. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this unpublished work. The copyright notice is not an admission that publication has occurred. This work contains confidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or transmitted in any form or by any means without the prior written permission of ATI Technologies Inc."

Table Of Contents

Revision Changes:

Rev 0.0 (Your Name)
Date: January 1, 2000
Initial revision.

1. Introduction

The shader system processes groups – known as vectors or threads - of vertices and pixels. The SQ receives vertex info from the VGT and pixel info from the SC, and controls their flow through the shader pipe.

The following list describes this flow of vertex and pixel vectors through the SQ and also touches on the high level the functions of various modules contained in the SQ.

1.1 Top Level Flow

1. The VGT sends vertex data (indices) to the SQ; the SQ buffers them, and after receiving a vectors worth, transfers them to the SP
   - the SQ's Vertex Input Contol module distributes the vertices to the 4 SPs by reading from the Vertex FIFO and writing to the SP's vertex staging registers (VSRs)
2. The SC sends pixel data to the SP and parameter data to the SQ
   - quads of ij data are sent directly to the SP interpolators (SPIs) from the SC
   - pointers to the parameter data associated with the ij's are sent from the SC to the SQ
   - the SQ's Pixel Input Control module coordinates the pixel interpolation by reading the parameters from the SX's parameter cache and ij data from the SPI's ij buffers and writing the result into the GPRs
3. The SQ writes vertex and pixel vectors into the SP GPRs
   - the SQ's Input Arbiter module arbitrates between pixel and vertex vectors since they both share the same write port into the GPRs
   - the SQ's GPR Alloc module must also allocate space for a vector in the GPRs before it's loaded
4. The Input Controller that sucessfully writes its vector to the GPRs then sends a control packet for the vector to the associated Thread Buffer module (a.k.a. Reservation Station)
   - the control packet contains the context ID and a GPR base pointer (along with other control info)
   - one control packet exists for each vector being processed by the shader pipes
5. The Vertex Thread Buffer can hold 16 vertex threads and the Pixel Thread Buffer can hold 64 pixel threads
   - the term thread really refers to both the vector's actual data, which is in the shader pipe, and the vector's control packet, which is in the SQ
   - so thread and vector generally mean the same thing, while the term control packet is pretty much limited to the chunk of info sent to the thread buffers by the input controllers
6. Threads enter and exit the thread buffer in a FIFO-like manner
   - a thread may be selected to run several times before it is actually removed from the buffer
7. Threads are picked for execution by the Thread Arbiter module
   - threads have an associated status that, in addition to the age of the thread, determine it's priority
8. In addition to the two types of threads, vertex and pixel, there are two important sub-types: texture and ALU; the thread arbiters work with these thread sub-types
   - so there is a Texture Thread Arbiter that looks at all the texture threads in both the Vertex and Pixel thread buffers, and an ALU Thread Arbiter that looks at all the ALU threads in both reservation stations
   - note that a shader program is being run on the threads, and that the shader program has both texture and alu instructions, so the thread sub-type is really a status bit that changes dynmically with the execution of the shader program
9. after picking a thread, the thread arbiter sends the thread's control info to the associated Control Flow Sequencer
   - the texture thread arbiter sends the chosen control packet to the texture control flow sequencer, and the alu thread arbiter send its chosen thread's control packet to the ALU control flow sequencer
   - from here on down, the processing of texture instructions is separated from the processing of alu instructions; they each have their own instrcution pipe that starts with a control flow sequencer

- also note that there are two ALU instruction pipes (per SIMD) in order to achieve on overall instruction rate of one every 4 cycles (it takes 8 cycles to execute an alu instruction so each AIS issues an instruction every 8 cycles, but the two AISs are offset by 4 clocks, so the final rate becomes one instruction every 4 clocks)

10. the control flow sequencer module (CFS) executes control flow instructions
    - part of the control packet is an instruction store address; the CFS uses it to fetch control flow instructions
    - the CFS also breaks the thread execution up into clauses
    - a clause is a sequence of identical instruction types – either texture or alu
    - a clause boundary is identified by the CFS when the instruction type changes
    - when it hits a clause boundary, the CFS is finished with this thread and it updates the thread's status in the reservation station (the thread is done when the last instruction of the shader is detected)

11. the CFS then passes a count of how many instructions there are in the clause and a starting instruction store pointer to the Instruction Fetch module

12. the instruction fethcer basically uses the starting pointer and fetches the given number of instructions, passing them to the Instruction Queue

13. the Instruction Queue hold the instructions on their way to the Instruction Sequencer

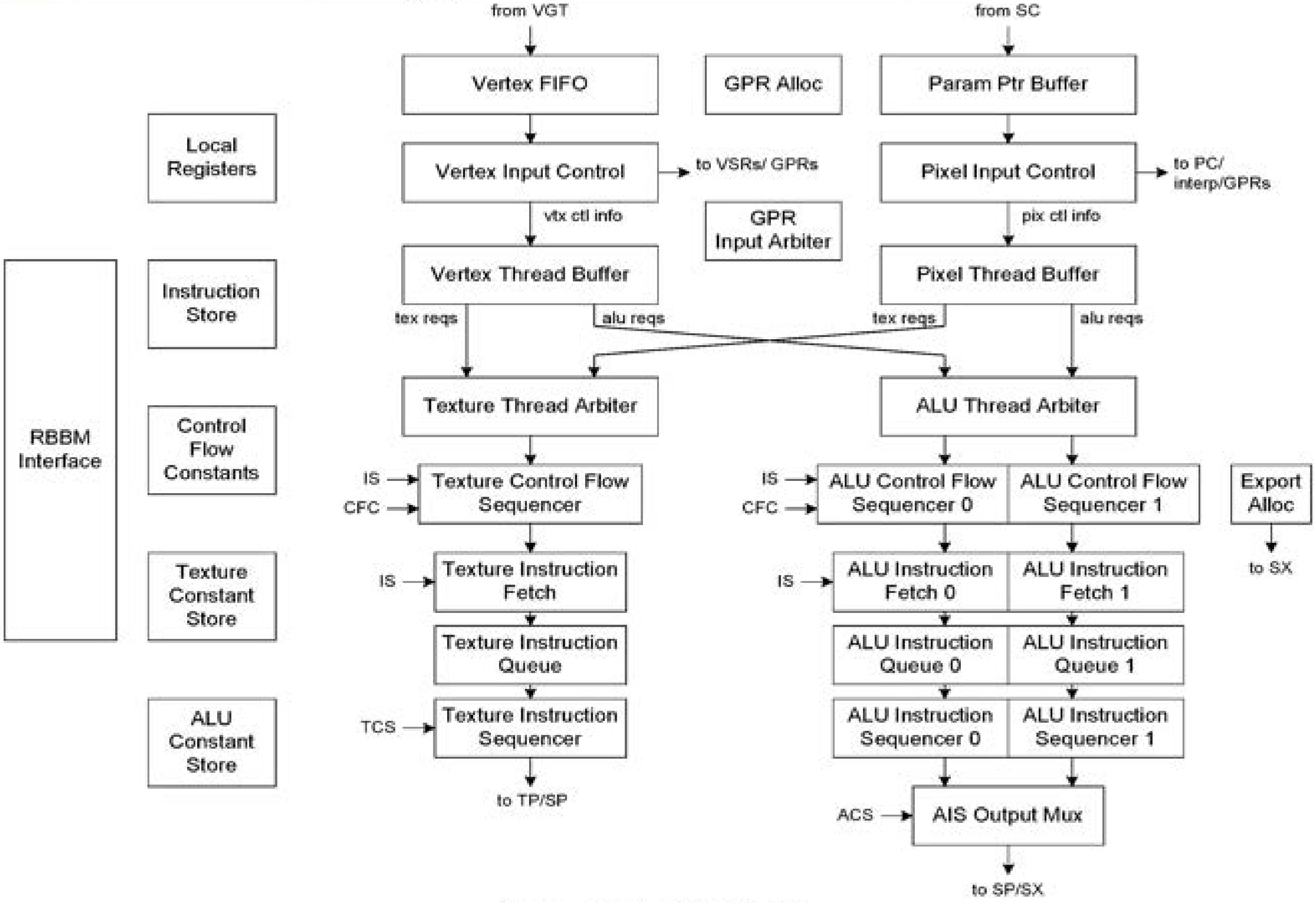
14. the Instruciton Sequencer then drives its instruction interfaces

15. the texture instruction sequencer sends texture instructions to the Texture Pipe (the SQ also provides the read GPR address of the texture coordinates to the SP for texture instructions), and the ALU instruction sequencer sends ALU instrctions to the SP (some control info also goes to the SX for ALU export instructions)

16. after all the instructions of a clause have been issued, the Instruction sequencer will update the status of the thread in the reservation station
    - a flag indicating the end of the shader program is also passed dowm with the control packet; the Instruction Sequencer uses this flag to tell the thread buffer that a thread is done

17. when the thread buffer sees that the last instruction of a thread has completed, the GPRs are deallocated, the thread is removed from the buffer, and the SQ's role in processing the vector is complete

1.2 SQ Block Diagram



Sequencer Top Level Block Diagram

2. GPR Access Synchronization

The Sequencer reads and writes the Shader Pipe GPRs. Vertex and pixel input data, texture fetch data, and SP ALU results are all sources of writes, and ALU operands and texture fetch addresses are the possible results of reads.

To manage the different write sources and read destinations, access to the GPRs is divided into four phases. The SQ uses a two bit GPR phase count to synchronize the different controllers that perform the above read and write accesses. The SQ also sends the GPR phase to the SP for use as a GPR write data mux select, and to the TPC so it knows the correct cycle to send fetch return data.

The following table summarizes the type of write and read accesses associated with the GPR phase. There is a fixed relationship between the operand read accesses and the result write accesses based on the timing of ALU instruction execution (i.e. the number of cycles it takes from reading the operands from the GPRs to writing the results back to the GPRs – see SQ_SP GPR timing diagram).

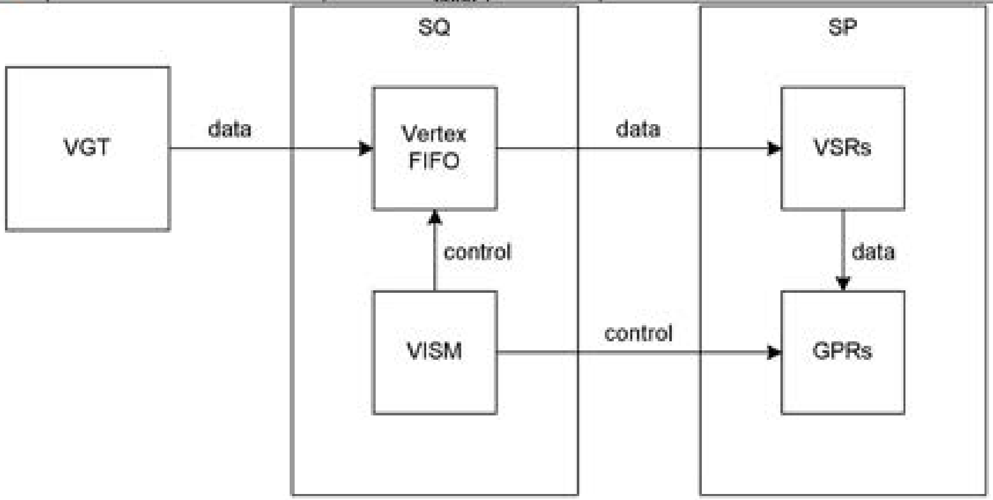
GPR phase	write cycle	read cycle
0	input data (ID)	source B
1	fetch data (FD)	source C
2	vector result (PV)	fetch address (FA)
3	scalar result (PS)	source A

3. Vertex Input Control

The Vertex Input Control logic consists mainly of the Vertex FIFO and the Vertex Input State Machine. Vertex data from the VGT is first loaded into the Vertex FIFO. It is then read from the FIFO and sent to the SP Vertex Staging Registers. From the VSRs the data is loaded into the GPRs by the Vertex Input State Machine (see figure).

3.1 Vertex FIFO

The Vertex FIFO is basically a skid buffer for the VGT to SQ vertex data interface. The VGT transfers one vertex index per cycle to the SQ.

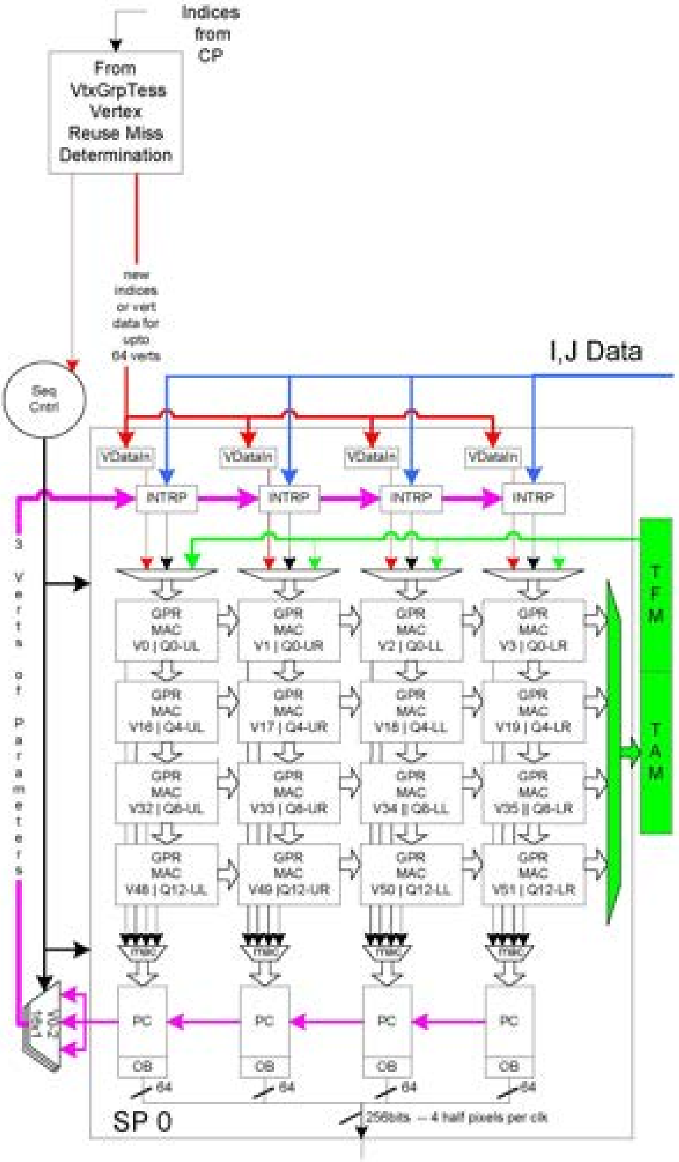


Vertex Input

3.2 Vertex Input State Machine

The following lists the functionality of the Vertex Input State Machine (VISM):

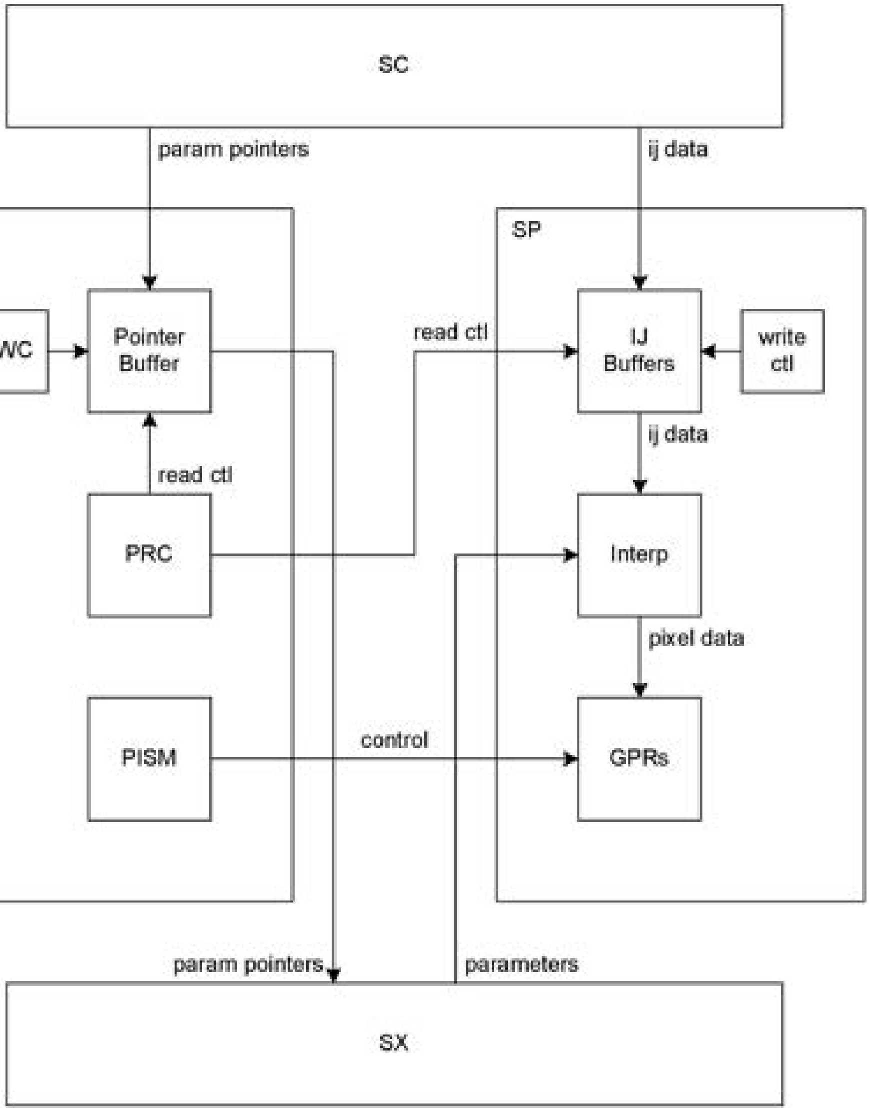
- reads 96 bits of vertex data per cycle from the FIFO
  - only the least significant 32 bits are used to represent a vertex index
  - HOS data can use 3 32-bit words or, when the double flag is asserted, 6 32-bit words
- sends them to the correct SP staging registers (VSR0 and 1)
  - all 64 verts (or groups of HOS data) are transferred into the individual SP staging buffers before actually being loaded into the GPRs
  - if the vertex vector is less than 64 in length, the SQ will only transfer the valid number of verts and set the valid bits appropriately
  - each SP has staging buffers organized as 4 8x100 rams (8 locations needed in double mode)
  - the assumption is that the verts will come in order from the VGT and the SQ will route them to the correct SP (i.e. verts {0..3} to SP0, {4..7} to SP1, {8..11} to SP2, {12..15} to SP3, {16..19} to SP0, etc.)
  - the SQ broadcasts the data to all the SPs and sends a separate valid bit to each SP
  - the SP must keep a counter and/or a small state machine to control the staging buffer ram write address and write enable (also must control read address and read enable when receiving a read request – see below)
- once the entire vector of verts has been sent to the staging buffers, the VISM requests GPR storage and a write slot
  - the write request goes to the Input Arbiter (IA) and the space requirement goes to GPR Allocater (RA)
  - the space requirement is in the VS_NUM_REG field of the SQ_PRGM_CNT_SHADING local register
  - the IA asserts a grant signal both to the RA, to allow the allocation to occur, and to the VISM, to allow it to proceed (note that the grant to the VISM is properly sync'd to the GPR phase counter)
  - the gpr base pointer is returned from the RA to the VISM where it is stored in a register/counter (it will be used as the source of the gpr write address)
- when granted, the VISM will transfer the vertex vector into GPRs
  - the VISM will drive the GPR write address (using the base it received from the RA) and the GPR write enables (which are all asserted)
  - the number of cycles required for the load can be 4 for single VSR, 8 for double VSR or single VSR plus auto count, or 12 for double VSR plus auto count
  - VSR 0 goes into GPR base, VSR 1 (if enabled) goes into base + 1, and count (if enabled) goes into base + 2
- finally the VISM loads a vertex control packet into the Vertex Shader Sequencer's first reservation station FIFO
  - the state info, valid bits, and GPR base are loaded into the RS FIFO on the last cycle of the gpr load



4. Pixel Input Control

The SC sends ij data directly to the four SPs, while it sends control info (parameter cache pointers, etc.) to the SQ. The SQ uses the pointers to read the SX parameter cache, with the resulting parameter data being routed into the SP interpolators. The SQ must also initiate the transfer of ij data from the SP ij buffers to the interpolators. The interpolators then use the ij data and the per-vertex parameters to create the per-pixel data that is loaded into the SP GPRs (see figure).

The Pixel Input Contol logic consists primarily of the Pointer Buffer (PB), the PB write logic (PWC), the Pointer Read State Machine (PRSM), and the Pixel Input State Machine (PISM). Note that the pointer read and the pixel input are separate state machines because they need to overlap due to interpolation latency.



Pixel Input

The following describes the pixel input control logic:

- 16 quads of ij data are sent to the 4 SPs (4 quads to each SP) for one vector of pixels; the corresponding parameter cache pointers are sent to the SQ (along with other control info such as the context)
- 1 quad of ij data (128 bits) can be sent to each SP over two cycles
  - a write mask is associated with each quad, so from 0 to 4 quads can be sent (to all SPs) per 2-cycle transfer
  - the control info is sent to the SQ at the same rate as the ij data is sent to the SPs (so the SQ can receive from 0 to 4 quads worth of control info over two cycles)
  - all quads sent to different SPs during one 2-cycle transfer must be from the same primitive

- each SP has an input buffer for storing the 4 quads of ij data; the SQ has 4 similar buffers for storing the corresponding 16 quads worth of parameter cache pointers
    - actually these are ping-pong buffers, so the total storage is 8 quads/SP; 32 pointers in the SQ
    - considering the 4 SPs all together, the 16 quads form a 4x4 2d array, with the rows going across the SPs and the columns within a SP – the pointer buffer in the SQ mirrors this structure
    - the SP buffers are written independently – it is up to the SC to route valid quads to the correct SP (so that they fill the 2d array sequntially from column 0 to column 3 and from row 0 to row 3)
    - this packing allows quads from different primitives to be on the same row (this also means that a 2-cycle write from the SC may go to adjacent rows of the buffer)
    - all control info other than pointers (state, valid bits, etc.) is stored in a separate ping-pong buffer
- the pointer and ij buffer control algorithm is basically:
    - write the locations of the active ping-pong side sequentially (based on the write enables sent with the data)
    - read the rows in order from 0 to 3, possibly multiple times (while any incoming writes go to the other side of the buffer)
    - swap sides of the ping-pong buffer when done reading

Because a row can contain quads from different primitives, multiple 4-cycle read passes may be required to send the ij data to the interpolators (since quads from different prims use different parameters). Also when there is more than one parameter to be interpolated, the same series of passes is made for each parameter.

If any special parameter 0 data (xy, gen_st, faceness) is associated with this pixel vector, it is transferred to the GPRs before the interpolated data. If xy is enabled, only one 4-cycle read pass through the xy buffer is required. If the vector count is enabled (see section on auto counts), it is also transferred before the interpolated data.

Space in the GPRs must be allocated before the interpolated data can be transferred to the GPRs. This is done by the GPR Allocation (RA) module. Also, there is arbitration between vertex and pixel vectors, so the pixel vector must be granted a turn to write into the GPRs before it can start.

Finally, a contol packet for the pixel vector is loaded into the pixel thread buffer.

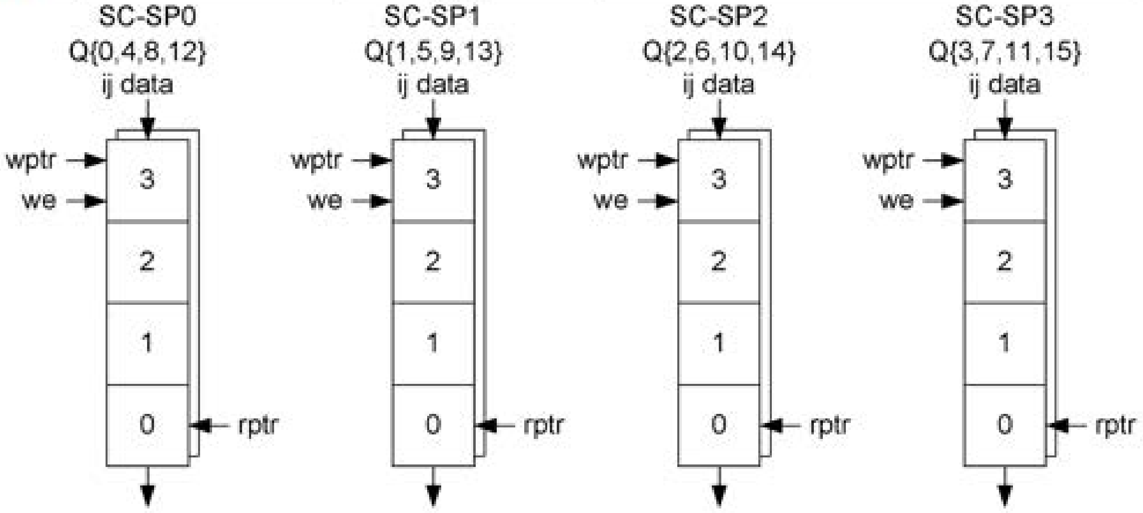
4.1 Pointer Buffer Write Control

The following describes the implementation of the Pointer Write Control (PWC):

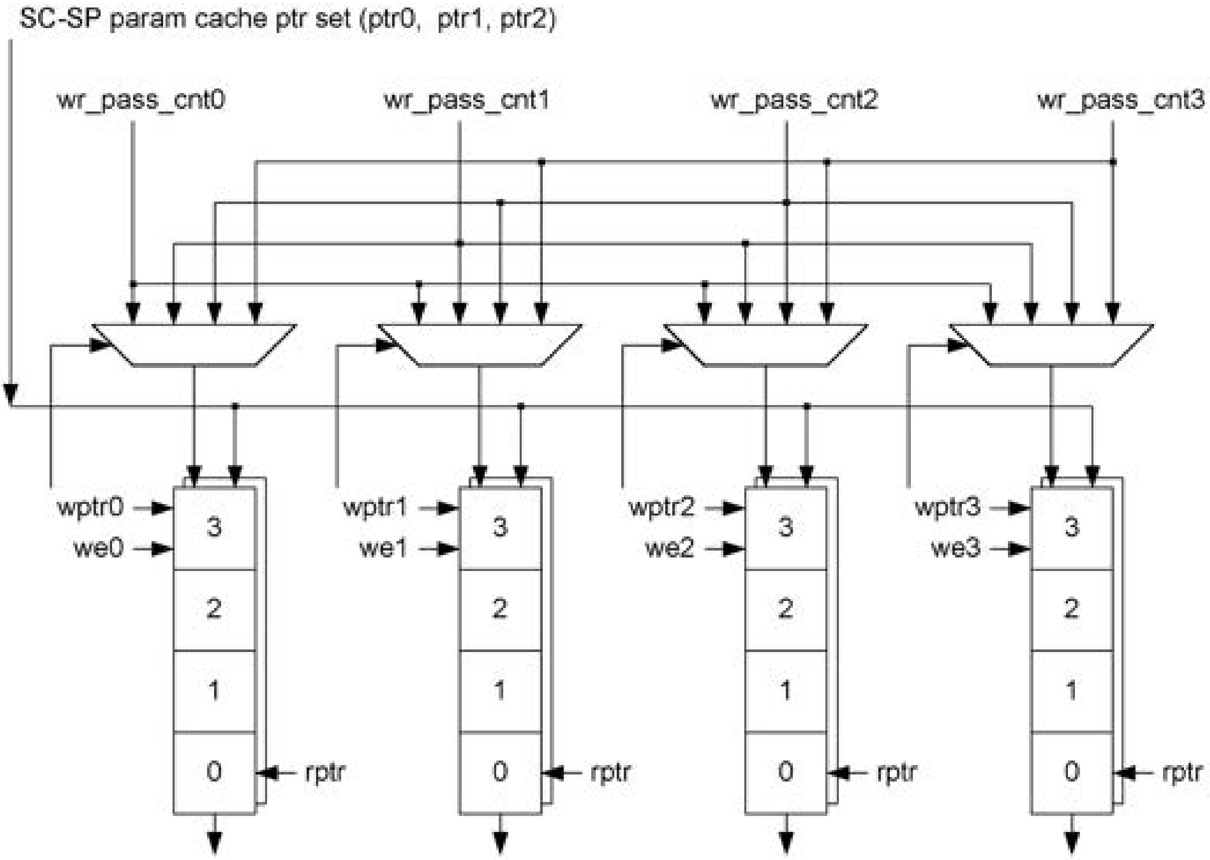
- pointer data is written to the buffers based on the write enable from the SC and buffer avaliability
    - buffer availability depends on whether both sides of the ping-pong buffer are full
        - when one side is in use, it is full
        - when the PRSM is done with all the read passes for a side, it will free the entire side back up for writes
- write pass counters (one per row) are used to track primitive changes within a row
    - a copy of the current write pass count is written into the buffer along with the pointer
    - write pass counts are incremented when the end_of_primitive signal is asserted
    - the per column write enables from the SC indicate the valid quads of a primitive, so sucessive prims in a row will have increasing pass counts
- when the current buffer side fills up, switch to the other side for writes (if it is not full)
    - if both sides are full, deassert the RTR back to the SC
- if the vector being transferred contains less than 16 quads (end_of_vector asserted early), just set the full bit for this side and switch to the other side as above

4.2 Pointer Buffer

The pointer buffer stores the 3 parameter cache pointers needed for interpolation (described in detail above), as well as other per primitive control that is needed for interpolation. Also included in this module is a ping-pong buffer that's used to store the per-vector state info sent with the pointers.



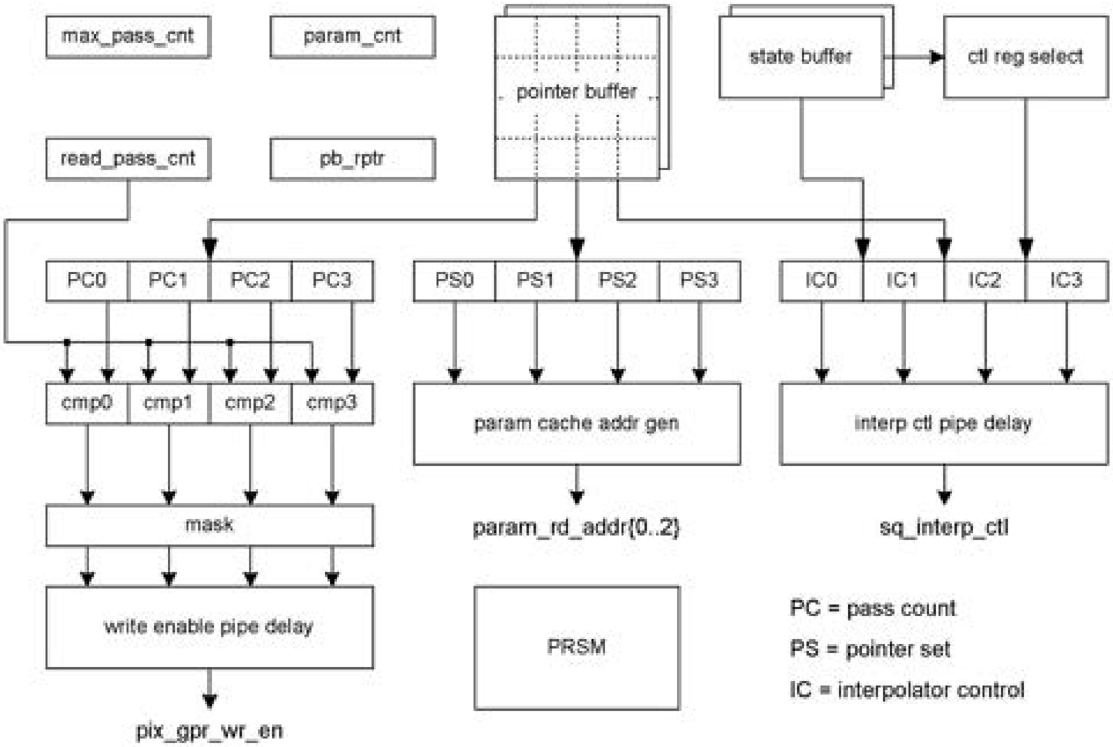
Shader Pipe IJ Data Buffers



Sequencer Parameter Cache Read Pointer Buffer

4.3 Pointer Read Control

The Pointer Read Control (PRC) module is responsible for reading the pointer buffer and sending the resulting pointers to the parameter cache read interface. The control is generated by the Pointer Read State Machine (PRSM). The PRC also drives the SQ_SP Interpolator interface and generates the correct GPR write enables for pixel writes.



Pointer Read Control

Because of the latency associated with getting pixel data through the interpolators (param cache access, latency through the SX, latency through the interpolator pipeline), the Pointer Read SM is separate from the Pixel Input SM. This will allow the two state machines to overlap; i.e. the pointer read for the next vector can start while the current vector is being loaded.

The following describes the implementation of the PRSM:

- when a pointer buffer side is full, request a GPR write slot and GPR storage allocation
  - the PRSM requests the write slot from the Input Arbiter (IA)
  - the space requirement is sent to the GPR Allocater (RA)
    - the state info needed to index the local registers comes from the vector state ping-pong buffer (in the pointer buffer module)
  - the IA asserts a grant signal both to the PRSM and the PISM to allow them to start

- the gpr base pointer is returned from the RA to the PISM where it is stored in a register/counter (it will be used as the source of the gpr write address, and the original base is copied to the shader sequencer)
    - in order to overlap pointer reads and pixel writes, the GPR base pointers (and the state info) will need to be stored in an input FIFO within the PISM
- when the PRSM sees the grant from the IA, it starts reading the pointer buffer
    - the IA has sync'd the grant to the correct GPR phase (this depends on the interpolation latency, which is a fixed value; once the read starts, there's no stalling the data, so the read has to be started on the exact cycle that will cause the data to show up at the gpr write port coincident with the address and write enables)
    - one row (each of the four pointer buffers) is read per cycle
    - one 4 cycle read pass causes four quads (per SP) to be sent thru the interpolators to the GPRs
- also start reading the SP ij buffers after a fixed delay
    - this delay is the time it takes to get the parameters into the top of the SP
    - this can be implemented as pipeline delays, or as another counter/state machine
- a read pass counter (initialized to 0 and incremented after each 4-cycle read pass) is compared to each of the pass counts as they are read out of the pointer buffer
    - if the counts match, then the corresponding GPR write enable for the interpolated data will be set (for those that don't match, the enables are not set since these pointers are for different primitives and must be interpolated on subsequent passes)
        - there's also a pixel valid bit that is considered when generating the GPR write enable
    - for any particular pass, pointers from the lowest matching column are sent to the parameter cache (all pointers from matching columns are the same)
        - there's a pointer mux on the buffer outputs which is controlled by the comparator matches (see figure)
- a param_done signal will indicate that all the passes for one parameter (up to 4) have been made
    - a row is done when the read pass count matches the pass count read out of the fourth (last) column in that row – a parameter is done when all 4 rows are done
    - this signal increments the param counter
- we're completely done with the pixel vector when the param counter equals the number of params being used by this state
- this state machine then returns to idle to look for the buffer to be full again (the other side could be ready to go right away – the min read time is 4 cycles, the min write time is 2 cycles)

4.4 Pixel Input State Machine

The Pixel Input State Machine (PISM) is responsible for supplying the write address with the correct timing to cause the interpolated data to be loaded into the GPRs. It also will drive the GPR input mux select to load generated param 0 data (if enabled) and auto count (if enabled) before interpolated data (gen'd p0 data and count are loaded during the initial interpolation latency).

In order to pipeline multiple pixel vector loads, the PISM contains an input FIFO that holds the state and base address info necessary to control the GPR input.

The following describes the implementation of the PISM:

- the PISM supplies the GPR storage requirement to the RA
    - the space requirement is is stored in the PS_NUM_REG field of the SQ_PRGM_CNT_SHADING register
    - the PISM get the correct state from the pixel input FIFO (which got it from the vector state ping-pong buffer)
    - the RA returns the gpr base pointer to the PISM where it is stored first in a FIFO, then in a register/counter (it will be used as the source of the gpr write address, and is coped to the thread buffer)
- when the pixel write request is granted by the IA, the PISM can be either idle, or busy transferring the previous pixel vector
    - when idle: the PISM needs to wait for the interpolation latency
        - if count and/or sprite is enabled, subtract 4 cycles for each one enabled from the latency since the PISM will overlap the transfer of these with the interpolation

  - when busy: once the PISM has started the last GPR load of the current vector, it will check if the next pixel vector has been granted; if so, it'll wait the number of cycles left on the latency timer (again less if count and/or sprite is enabled)
    - note this can be done by loading the current arbiter count into the PISM counter and waiting until it reaches the latency requirement (more qualification is needed in the case of multiple overlapping accesses)
- finally the PISM loads the control packet info into the first pixel shader sequencer reservation station FIFO
  - the control info for pixels is the same as that for verts, plus the LOD correction bits

5. Auto Counts

Included in each of the Input State Machines is a vector counter whose value is sent to the SP for loading into a specific GPR when the GEN_INDEX register bit is set. The count is reset whenever the ISM sees a state change, and is incremented for every new vector.

For vertices, the auto_count is always loaded into R2.x (gpr_base + 2). For pixels, the auto_count is loaded into R0.r (gpr_base) when point spites are not enabled (GEN_ST is clear) or into R1.r (gpr_base + 1) when point sprites are enabled (GEN_ST is set) (sprite data is loaded into R0).

Each ISM generates an input mux select that controls the source of the data being loaded into the GPRs via the Input Data (ID) port (either vertex, pixel, or auto_count data).

The Input Arbiter (IA) controls the mux that selects between the two ISM input mux selects.

6. GPR Allocation

The GPR Allocation (RA) logic manages the GPR space used by vectors of vertices and pixels. The space available for allocation can be configured as either static or dynamic, where static is a fixed division of the total space between verts and pixels, and dynamic allows the boundary between vertex space and pixel space to move based on demand.

Alocation requests are made by the Input Arbiter (IA) since it needs to know whether register space is available before granting a write request. The vertex and pixel space requirement is supplied by the corresponding Vector and Pixel Input state machines.

Deallocation requests are made by the shader sequencers as vectors leave the thread buffer.

The RA module is implemented as a group of pointer registers and a state machine that controls the loading of these registers. The following pointer registers are used (where v means vertex and p means pixel): v_head, v_tail, v_max, p_head, p_tail, p_max. Free space in the register file is calculated based on the pointers. Space allocation requests are granted when there is enough free space to satisfy the request; the pointers are then updated. Deallocation requests simply update the pointers.

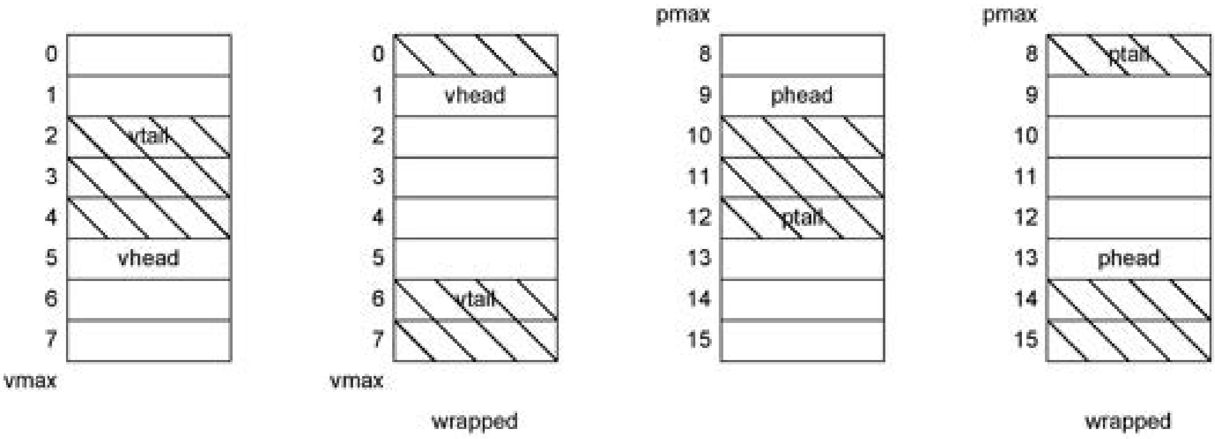
6.1 Dynamic Allocation

The following describes the functionality of the dynamic allocation logic:

- pixel space is allocated from the top (starting at location 0), and vertex space is allocated from the bottom (starting at the max location, which is 127 since there are 128 registers)

- a minimum amount of space is reserved for each type – this means the min of one type is the initial max of the other type
    - so if the mins are set at 25% of the space, each max is initially 75% resulting in a 50% overlap in the space available for allocation of each type
- as one type fills and passes its min, it reduces the max for the other space
- when the maxes of each type meet, a temporary fixed boundary is formed; this is then the wrap point for each type
- the boundary will disappear when free space becomes available on each side of the boundary (refered to as a bubble); at this time, the max of each type will snap back to the head of the opposite type

6.1.1 Dynamic Allocation Algorithm



Vertex Space:

```
vhead = vtail = 0;
vmax = 127 – PIXMIN;

// free space
if ( !vwrapped )
  vfree = vmax – (vhead – vtail);
else
  vfree = vtail – vhead;

// alloc – update head pointer
if ( space <= vfree)
  if ( vhead + space >= vmax )
    vwrapped = true;
    vhead = vhead + space - vmax;
  else
    vhead = vhead + space;

// dealloc – update tail pointer
if ( vtail + space >= vmax )
  vwrapped = false;
  vtail = vtail + space - vmax;
else
  vtail = vtail + space;
```

```
// update pmax
if ( vwrapped )
  pmax = vmax + 1;
else
  if ( vhead > pmax + 1 )
    pmax = vhead – 1;
```

The algorithm for pixels is essentially the same except that the phead and ptail start at 127, and space is allocated from the bottom of the register file.

Note that since the above was written, pixel and verts have been swapped such that now pixels start from the top and verts start from the bottom.

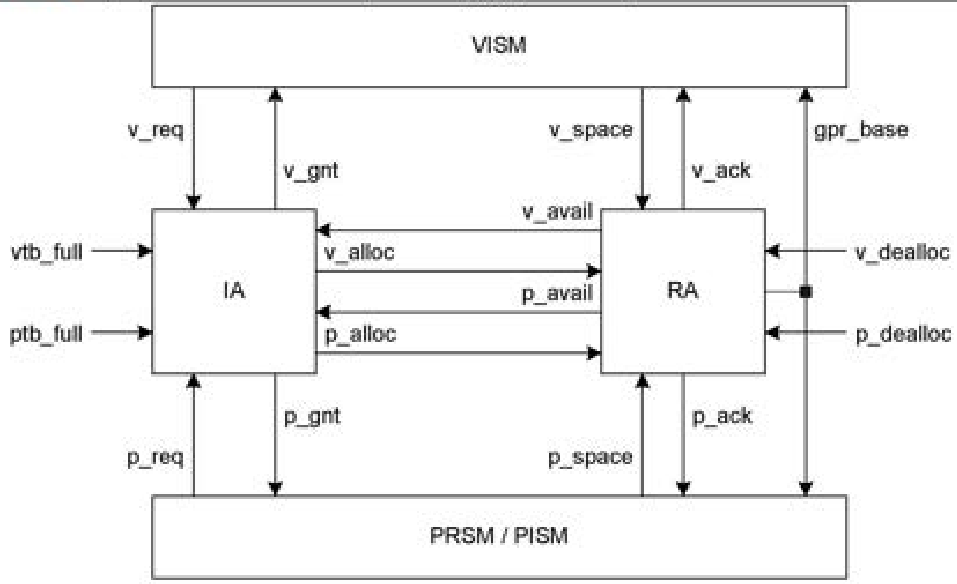
6.2 GPR Alloc State Machine

The following describes the GPR Alloc State Machine:

- the gpr_alloc state machine handles four types of requests – vertex alloc, pixel alloc, vertex dealloc and pixel dealloc
- the space required is presented with an alloc request
- for alloc requests, the SM will move to the ack state if there's enough free space to satisfy the request
  - the request will not be recognized until the space is available (actually the IA may hold off the alloc request – even if space is available – if there is a transfer in progress)
  - vertex requests (with available space) have priority over pixel requests (with available space)
- for dealloc requests, the SM will move to the ack state as soon as the request is recognized
  - alloc requests have priority over dealloc requests (deallocs will get thru when the allocs are stopped due to lack of space), and vertex deallocs have priority over pixel deallocs
- the new pointers are calculated during the idle state, and updated when moving to the ack state
- for alloc requests, new max values are updated when moving back to the idle state
- for dealloc requests, it is checked if the used space has gone to zero (i.e. empty), and if so, the pointers are reset to their initial values
- dealloc state flow then moves to the bubble check state, where if it is seen that a bubble has formed (a bubble is continuous space between the two head pointers), the max values are updated

7. Input Arbitration

The GPR Input Arbiter (IA) grants vertex and pixel GPR write requests based on 1) the availability of GPR space, 2) any currently active transfers, 3) on priority, and 4) whether the associated thread buffer is full. There is also a constant synchronization bit for each state that is used to enable vertex input grants (see the section on constant remapping table implementation). The following figure shows the major connections between the ISMs, the IA, and the RA:



The following describes the IA logic:

- the vertex request is made by the VISM and the pixel request is made by the PRSM
- priority is given to vertex requests, but this can be changed via the INPUT_ARB_POLICY register
- the RA tells the IA if space is available for a request, and the IA uses this info along with the priority setting to determine who gets granted when there are no transfers in progress
- the IA is basically a state machine, but it also contains a counter that is used to determine if incoming requests that occur while a transfer is in progress can be granted
- the IA grants the request in sync with the GPR phase
  - for verts, this is on phase 2
  - for pixels it gets a bit trickier: The pixel grant triggers the PRSM to start the whole interpolation process, which has a fixed latency based on the time to read the param cache, get the params back to the SP, and the time through the interpolation pipeline
- vertex requests will not overlap due to the nature of vertex transfer, but a pixel request can occur during a vertex tranfer, and a vert or pixel request can occur during a pixel transfer
- once it's recognized a vert request, the IA will not look at pixel requests until the vert request has been sync'd and transfer has begun
  - vertex transfers are so short that even in the worst case (double mode plus count), the interpolation latency will allow the pixel request to simply be sync'd and granted
- once it has recognized a pixel request, the IA will not look at vert or new pixel requests until the current pixel request has been sync'd and transfer has begun
  - once pixel transfer has started the IA will again look at new vert and pixel reqs
  - a new vert request will lock out any new pixel requests if priority is given to verts; although the vert transfer will happen right after the pixel transfer, any possible overlap of pixels is lost (to prevent even higher complexity)

The IA will also control output muxes that switch between VISM and PISM sources of GPR write addresses, GPR write enables, and input mux selects.

8. Vertex and Pixel Shader Thread Buffers

The vertex and pixel thread buffers store state and status information for the vertex and pixel threads currently being processed by the shader system. Each thread can request either texture or alu processing depending on the type of shader instructions that the thread wants to execute. All the texture requests from both thread buffers go to the texture thread arbiter, and all the alu requests from both thread buffers go to the alu thread arbiter. A thread arbiter returns the winning thread ID to the thread buffer, which then reads the winner's state and status info and sends it to the associated control flow sequencer.

A thread buffer receives its initial thread data from the associated input state machine. A thread buffer receives state and status updates from the control flow processor, the instuction sequencer, and from external sources such as the texture pipe (which returns texture fetch completed status). A state machine within a thread buffer sends a dealloc request to the GPR alloc module when a thread has completed.

Details of the state and status are in the Xenos Sequencer Specification section 6.3 (Xenos_SQ_Spec.doc).

9. Thread Arbitration

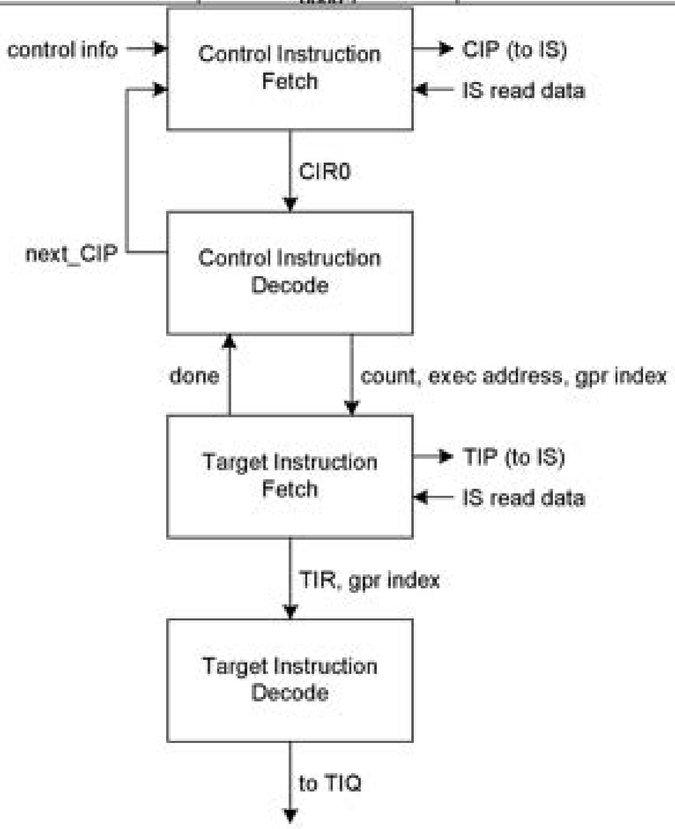
Multiple shaders are processed concurrently, but only one of each type of clause may execute at a time. Clause arbitration determines which clause to run.

10. Control Flow Sequencers

The Texture and the ALU Control Flow Sequencers execute control flow instructions and prefetch target instructions for the target instruction sequencer state machines. They receive control information from the reservation station selected by the corresponding clause arbiter.

The Control Flow Sequencer (CFS) functionality can be divided into the following groups:

- Control Instruction Fetch
- Control Instruciton Decode
- Control Instruction Execute
  - Target Instruction Fetch
  - Target Instruction Decode



Control Flow Sequencer

10.1 Control Instruction Fetch

The Control Instruction Fetcher (CIF) contains the following:

- Control Instruction Pointer (CIP)
- Control Instruction Registers 0 and 1 (CIR0, CIR1)
- Control Instruction Fetch state machine

The following lists the functionality of the Control Instruction Fetcher:

- gets the context, thread number, and the GPR base from thread buffer
- uses this info to select the CF base address from local registers
- calculates the initial Instruction Store address from the CF base and loads it into the CIP
- reads contol flow instructions from the IS using the CIP
  - there is also a CFS sub-phase to allow for the sharing of the IS CF instruction read phase
- waits for vaild control instruction data (based on the IS phase)
- two 48 bit CF instructions are received per IS read; they are loaded into CIR 0 and 1 (where 0 is the first instruction to be decoded)
- gets next instruction pointer from the Control Instruction Decode pipe and uses it to fetch the next instruction (unless the next instruction is already in CIR1, in which case it will just copy CIR1 to CIR0)

- move CIR1 to CIR0 when next_CIP is odd

10.2 Control Instruction Decode

The following lists the functionality of the Conrol Instruction Decoder (CID):

- a two stage decode pipe that gets its input from CIR1
- contains the CIP stack
- contains the 4 loop iterator counters
- selects the contol flow constant info (booleans, loop counts, etc) specified by the instruction
- calculates the next CIP and sends it back to the CIF
- calculates initial target instruction pointer (TIP)
- loads TIP and target instruction counter (TIC) for execute CF instructions
    - waits if target instruction execution is busy
- control flow instruction that do not execute target instructions can proceed concurrently with target instruction processing
    - may want a small queue between CID and TIF (2 entries)

The following RTL pseudo-code describes the functionality of each CF instruction:

execute:

```
TIC <- instr_count;
TIP <- absolute ? exec_addr : exec_addr + shader_base;
CIP <- CIP + 1;
```

conditional execute:

```
TIC <- (bool[BID] == condition) ? instr_count : 0;
TIP <- absolute ? exec_addr : exec_addr + shader_base;
CIP <- CIP + 1;
```

conditional execute predicate:

```
TIC <- (predicate_test[PID] == condition) ? instr_count : 0;
TIP <- absolute ? exec_addr : exec_addr + shader_base;
CIP <- CIP + 1;
```

conditional jump:

```
CIP <- [(bool[BID] == condition) & (fwd_test)] ? jump_addr : CIP + 1;
// fwd_test = ~fwd_only | (jump_addr > CIP);
```

conditional call:

```
CIP <- (bool[BID] == condition) ? jump_addr : CIP + 1;
IP_stack[SP] <- (bool[BID] == condition) ? CIP : hold;
SP <- (bool[BID] == condition) ? SP + 1 : hold;
```

return:

```
CIP <- IP_stack[SP – 1];
SP <- SP - 1;
```

loop start:

```
nest_lvl <- nest_lvl + 1;                                              // increment nest level when starting
CIP <- (iterator[nest_lvl] == loop_cnt[LID]) ? jump_addr : CIP + 1;    // jump to end + 1 to skip loop
gpr_index <- (iterator[nest_lvl] * loop_step[LID]) + loop_start[LID];  // load gpr_index initial value
```

loop end:

```
iterator[nest_lvl] <- iterator[nest_lvl] + 1;                              // increment the loop iterator
gpr_index <- (iterator[nest_lvl] * loop_step[LID]) + loop_start[LID];      // update the gpr_index
CIP <- (iterator[nest_lvl] == loop_cnt[LID]) ? CIP + 1: jump_addr;         // jump back to start + 1 to keep looping
iterator[nest_lvl] <- (iterator[nest_lvl] == loop_cnt[LID]) ? 0 : hold;    // reset iterator when done
nest_lvl <- (iterator[nest_lvl] == loop_cnt[LID]) ? nest_lvl – 1 : hold;   // decrement next level when done
```

clause end:

```
clause_done <- 1;
```

conditional clause end:

```
clause_done <- (bool[BID] == condition) ? 1 : 0;
```

10.3 Control Execution Unit

Contol flow instruction execution consists primarily of fetching and partially decoding target instuctions and loading the results in to the target instruction queue.

10.3.1 Target Instruction Fetch

The TIF contains the following:

- Target Instruction Pointer (TIP)
- Target Instruction Counter (TIC)
- Target Instruction Register (TIR)
- Target Instruction Fetch state machine

The following lists the functionality of the Target Instruction Fetcher (TIF):

- reads the target instruction using the TIP
- loads the TIR with data read from the IS
- increments TIP and decrements TIC for every instruction read
- sends a done signal back to the CID state machine when the TIC reaches zero to allow the next execute instruction to start

10.3.2 Target Instruction Decode

The following lists the general functionality of the target instruction decoders (the two different types of TIDs – texture and alu – have differing decode details which will be listed in the next two sections):

- calculates GPR read and write addresses
- puts the GPR addresses and the remaining instruction bits into the Target Instruction Queue

10.3.3 Texture Instruction Decode

The following lists the specific functionality of the Texture Instruction Decoder (TID):

- calculates the GPR read address for the texture fetch address (absolute or loop index)
- calculates the GPR write address for the texture fetch return data (absolute or loop index)
- puts the GPR read and write addresses, the texture instruction, the clause number, and an end-of-clause indicator into the Texture Instruction Queue

10.3.4 ALU Instruction Decode

The following lists the specific functionality of the ALU Instruction Decoder (AID):

- calculates the three GPR read addressess for srcA, srcB, and srcC (absolute or loop index)
- calculates the GPR write addresses for PV and PS when the register file is the destination (absolute or loop index)
- compares current source addresses with previous destination addresses and on a match, will change the source to PV or PS
  - if there is any write masking done, the PV/PS substitution is not made and a NOP is inserted into the instruction stream to account for writing the result into the GPRs
- calculate the parameter cache write address when the parameter cache is the destination
- puts the GPR addresses along with the remaining fields of the alu instruction into the ALU Instruction Queue

11. Target Instruction Queues

Each Control Flow Sequencer has an associated Target Instruction Queue. The Texture CFS places texture instructions into the TIQ, and the two ALU CFSs place their ALU instructions into their own AIQ. A target instruction sequencing state machine reads the contents of its instruction queue and drives its instruction interface.

12. Texture Instruction Sequencer

The following describes the Texture Instruction Sequencer state machine (TIS):

- reads the GPR addresses and the texture instruction from the TIQ
- reads the texture constant from the Texture Constant Store (TCS)
- reads the texture fetch address from the GPRs (the resulting fetch address goes directly from SPn to TPn)
- sends the instruction, the constant, the GPR write address, the clause number, and the end-of-clause signal to the TPC
- upon stalls from the TPC, the TIS will
  - still send the next instruction
  - continue to send the next instruction until the stall goes away
- note that timing of the transfers to and from the TP is determined by the GPR phase
  - fetch address reads are done on gpr phase 2
  - fetch data writes are done on gpr phase 1

13. ALU Instruction Sequencer

There are two ALU Instruction Sequencers - each reads from its associated AIQ.

The following describes the functionality of the AIS state machine:

- reads alu instructions, GPR read addresses, GPR/parameter cache write address from AIQ
- reads alu constants from the ACS
- calculates ALU Constant Store (ACS) read addresses for constant read indexing
  - each AIS contains 64 9-bit constant index registers (64*9 = 576)
  - the SP sends the constant indices via MOVA (36 bits/cycle over 4 cycles from each SP) (36*4*4 = 576)
  - the constant needs to be looked up – and the instruction re-issued – for every different constant index (up to 64 times)
  - the ACS physical memory address port (only 1) will be dedicated to reads while in this constant waterfalling mode
- transfers the instruction, constants, GPR read addresses, destination write address, etc. to the SP
- note that timing of the transfers to the SP is determined by the GPR phase
- for export instructions, also drives the SQ_SX export interface
- the AIS also contains the 4 64-bit predicate vectors
  - the AND/OR of all 64 bits of each predicate vector is sent back to the associated CFS for use with conditional execute predicate instructions

14. Register Bus Interface

The following lists the functionality of the RBI:

- register bus writes are loaded into an input FIFO
- the destinations are local registers, instruction store, texture and alu constants, and control flow constants
- a decoder looks at the FIFO output and sends a request to the appropriate destination module
- when ready, the destination module reads the data from the FIFO
- draw commands are also sent to the RBI

14.1 Local Registers

The local registers basically fall into two catagories – overall control and per-context control. The overall control registers are the normal type of control register that affects the functions of various pieces of logic in the SQ. The per-context control registers are like normal control registers except that there are 8 copies of each – one for each context (state) that can exist in the chip. Therefore, reads of the per-context control registers must include the state as a mux select to get the correct value.

See the architecture spec section on Registers for the full list (or the on-line register spec).

14.2 Control Flow Constant Store

Control Flow Constant store is entirely memory mapped (the CP will write all locations). CF constants consist of loop controls (start, count, and step) and booleans.

Refer to the arcchitecture spec for details on size.

14.3 Instruction Store

The Instruction Store is 4096 x 96 bits, and is also entirely memory mapped. The IS can be organized as either a single ring or a dual (split) ring, where the single ring has one overall shader base pointer and the dual ring has two overall shader base pointers (one for all vertex shaders and one for all pixel shaders).

In single ring mode, the vertex and pixel shaders are interleaved (grouped in VS-PS pairs), whereas in dual ring mode they are separated into their own section of memory (all the vertex shaders are grouped together and all the pixel shaders are grouped together). The wrap point for any VS or PS in single ring is the end of memory, and in dual ring mode, the VS will wrap at the PS base and the PS will wrap at the end of memory. Note that each individual vertex and pixel shader has its own base pointer that is used to fetch shader instructions.

Access to the IS is divided into four phases - 0: Control Flow instruction read, 1: Texture instruction read, 2: ALU instruction read, and 3: CP write (or read for debug). An instruction store phase counter will provide the phase info to the SQ.

14.4 Texture Constant Store

The following are the details of the texture constant store:

- the physical memory is 128 x 192
- a max of 32 constants can be used per context (shared between vertex and pixel shaders)
- each context uses a 32 location remapping table – so each location maps 1 constant
- since the register bus is 32 bits, 6 writes are needed to load the 192-bit constant

14.5 ALU Constant Store

The following are the details of the ALU constant store:

- the physical memory is 1024 x 128
- a max of 512 constants can be used per context (256 max per vertex or pixel shader)
- each context uses a 128 location remapping table – so each location maps 4 constants
- since the register bus is 32 bits, 16 writes are needed to load the 4 128-bit constants

The memory used for the ALU working remapping tables (WRTs – see section on remapping tables) is 8 128x8 RAMs or one 128x64 RAM.

14.6 Remapping Tables

Remapping tables are used to manage the physical memory available for the constant store, and allow it to be shared by all 8 contexts (states). The Texture and the ALU constant stores both use remapping tables, but the Control Flow Constant Store does not.

The following lists the functionality of the remapping table and constant store loading logic:

- logical addresses (LA) are used by the CP when writing constant data (these addresses are mapped to the SQ register space)
- physical addresses (PA) are managed by physical memory write (PMW) logic, and stored in the remapping table

- remapping tables for each context are read by the constant requestor (the CFSs) to obtain the the PA, which is then used to look up the constant data

See Figure in SQ Arch spec.

14.6.1 Implementation

These are the details of the remapping table and physical memory write control logic (sizes and cycles pertain to the ALU constant store):

- requests to write to the physical constant store memory come from the register bus interface
- the 16 writes required to write the constant data are all to the same logical address
- the LA becomes the remapping table write address
- physical addresses are initially generated by a counter (until they are all used)
- PAs are then taken from a list that are free to be reused
- the PA becomes the remapping table write data
- one remapping table is used for the current context that's being loaded (let's call it the loading remap table, or LRT), and 8 remapping tables are used for the 8 contexts that are in use by the chip (let's call them working remap tables, or WRTs)
- on a draw command (context switch), the LRT is copied to the corresponding WRT
  - at this point a bit for the loading state is set once all the constant writes have completed to the physical memory (PM) – this is for constant synchronization; the IA will not send a vertex control packet to the VSS for this state until the bit is set
  - the bit for the next state is cleared (since it will be the next loading state)
- the constant data being sent by the RBI is locally buffered (4 dword transfers are buffered into one 128-bit chunk) before being written to the PM
- the PA needs to be allocated (from counter or free list) in the time it takes to buffer up the first PM write (so it can be used to address the PM – there is plenty of time to write it into the LRT)
- the constant write data is actually double buffered due to the possibility that the PM write port is unavaliable due to heavy read usage by waterfalling constants (the PM is single ported)
  - again note that the synch bit is not set unitil all the data is written to the PM
- a PA is added to the free list when a logical address is reused between contexts (a new PA is being allocated for this LA, so the existing PA will be free to reuse when the state that is currently using this PA (from its WRT) has left the SQ)
  - e.g. the loading context is 4, and it issues a constant write to LA 23 for which a PA of 45 is allocated. but LA 23 already has a valid PA of 66 from some previous context. the PA of 45 gets written to LA 23, and the PA of 66 is pushed onto the free list – but it cannot be used just yet...
- three pointers are used to manage the free list – a head, a tail, and a free
- the tail ptr points to the end of the list (where PAs are written), and the head ptr points to the front of the list (where PAs are read)
- the free ptr marks the boundary between the PAs that are really free to reuse, and those that are yet to be freed
- a free-up table is used to store the number of PA's that can be reused once this context leaves the SQ; every time a PA is added to the free list by the current context a count is incremented – then on a draw cmd, this count is written to the free-up table indexed by the previous context (which is always the current context minus one). Now when a context leaves the SQ, the corresponding count is read and added to the free pointer (which makes that number of PA's now availabe to use)
  - e.g. head and free are both at 0, and tail is at 2 because the current context, 5, replaced two PAs in the LRT and wrote them to the free list
  - now when state 4 (the previous context, which is the one that loaded the PA's that were replaced (or inherited them from the context before that)) leaves the SQ, the free pointer is incremented to 2
  - now that head != free, PA's can be read from the free list until head == free again, at which point any remaining PA's in the free list are still in use

- when (head == free) and (pa_counter == max), all the PA's are in use; the logic must stall the RBI
- a pair of dirty bits is used for each location of the LRT in order to control the reamapping logic
  - the dirty bits are implemented as 2 128-bit registers
- the first bit is called reset_dirty, and is cleared on reset and set on the first use of a location – this allows the logic to recognize reuse of this LA by different contexts
- the second is called context_dirty, and it is cleared on every context switch and set when a location is used by the current context – this allows the logic to recognize reuse of the same LA within a context and to NOT allocate a new PA for any subsequent uses (constant writes to the same LA by the same context go to the same PA)

15. Timing Diagrams

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
sq.gpr_phase	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	
read cycle	FA	srcA	srcB	srcC	FA	srcA	srcB	srcC	FA	srcA	srcB	srcC	FA	srcA	srcB	srcC	
write cycle	PV	PS	ID	FD	PV	PS	ID	FD	PV	PS	ID	FD	PV	PS	ID	FD	
SQ_SP_gpr_rd_addr			srcA	srcB	srcC						srcA	srcB	srcC				
SQ_SP_instr_start																	
SQ_SP_instr				i0_0	i0_1	i0_2	i0_3					i1_0	i1_1	i1_2	i1_3		
SQ_SP_const				c0_0	c0_1	c0_2	c0_3					c1_0	c1_1	c1_2	c1_3		
SQ_SP_gpr_phase		2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1
SQ_SP_gpr_wr_addr														PV			
sp0.gpr_rd_addr				srcA	srcB	srcC						srcA	srcB	srcC			
sp0.su0.mac0.q_rd_data						srcA	srcB	srcC						srcA	srcB	srcC	
sp0.instr_start																	
					0	1	2	3	4	5	6	7	8	9	10	11	12
mac0.vector alu result												a	b	g	r		
mac0.gpr_phase			2	3	0	1	2	3	0	1	2	3	0	1	2	3	0
GPR write cycle			PV	PS	ID	FD	PV	PS	ID	FD	PV	PS	ID	FD	PV	PS	ID

SQ-SP gpr and instruction interface timing

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
sq.gpr_phase	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2
read cycle	FA	srcA	srcB	srcC	FA	srcA	srcB	srcC	FA	srcA	srcB	srcC	FA	srcA	srcB	srcC	TC
SQ_SP_gpr_rd_addr		FA	srcA	srcB	srcC	FA	srcA	srcB	srcC	FA	srcA	srcB	srcC	FA	srcA	srcB	srcC
sp0.gpr_rd_addr			FA	srcA	srcB	srcC	FA	srcA	srcB	srcC	FA	srcA	srcB	srcC	FA	srcA	srcB
sp0.mac0.q_rd_data					q0				q0				q0				q0
sp0.mac1.q_rd_data						q4				q4				q4			
sp0.mac2.q_rd_data							q8				q8				q8		
sp0.mac3.q_rd_data								q12				q12				q12	
SP_TP_fetch_addr						q0-3	q4-7	q8-11	q12-15	q0-3	q4-7	q8-11	q12-15	q0-3	q4-7	q8-11	q12-15
SQ_TP_instr_rts																	
SQ_TP_instr						i0_0	i0_1	i0_2	i0_3	i1_0	i1_1	i1_2	i1_3				
SQ_TP_const						c0_0	c0_1	c0_2	c0_3	c1_0	c1_1	c1_2	c1_3				
SQ_TP_clause						0				0							
SQ_TP_gpr_wr_addr						56				57							
SQ_TP_end_of_clause																	
SQ_TP_gpr_phase		2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1
TP0_SP0_gpr_wr_addr			56				57										
TP0_SP0_gpr_wr_en			0xf				0xf										
TP0_SP0_gpr_wr_data			q0	q4	q8	q12	q0	q4	q8	q12							
TPC_SQ_clause							0										
TPC_SQ_data_rdy																	
SQ_TP_gpr_phase		0	1	2	3	0	1	2	3	0							
SQ_SP_gpr_phase		0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
SQ_SP_gpr_wr_addr		ID	-	PV	PS	ID	-	PV	PS	ID	-	PV	PS	ID	-	PV	PS

SQ-TP and SP-TP interface timing (texture instruction transfer, data return)