

	ORIGINATE DATE 23 January, 2001	EDIT DATE 4 September, 2015	DOCUMENT-REV. NUM. GEN-CXXXXX-REVA	PAGE 1 of 12
Author: Andrew Gruber, Andi Skende				
Issue To:			Copy No:	
Shader Processor ver 0.1				
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 : HD PC Current Intranet Search Title: Shader Processor				
APPROVALS				
Name/Dept		Signature/Date		
Remarks:				
THIS DOCUMENT CONTAINS [REDACTED] 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 The material in this document constitutes an unpublished work created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this unpublished work.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 [REDACTED] 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

1. STATE 5

1.1 Shader State 5

1.1.1 GPR 5

1.1.2 Constant Registers..... 5

1.2 Texture/Memory State..... 7

1.3 Initial state 7

1.3.1 Vertex Shader 7

1.3.2 Pixel Shader 8

1.3.3 2D Shader 8

1.3.4 RealTime Shader 8

2. PROGRAM FORMAT 8

3. ALU 8

3.1 ALU instruction format..... 8

3.2 ALU Opcodes 9

3.3 Macro opcodes..... 10

4. TEXTURE/MEMORY 10

4.1 Instruction Format 10

4.2 Opcodes..... 11

	ORIGINATE DATE 23 January, 2001	EDIT DATE 4 September, 2015	DOCUMENT-REV. NUM. GEN-CXXXXX-REVA	PAGE 3 of 12
--	------------------------------------	--------------------------------	---------------------------------------	-----------------

Revision Changes:

Rev 0.0 (Andi Skende) Date: May 09, 2001 Initial revision.	Document started
Rev 0.1 (Andi Skende) Date: May 09, 2001 Initial revision.	Updated, added the instruction formant, initial block diagrams and preliminary interface description
Rev 0.2 (Andi Skende) Date: May 10, 2001 Initial revision	A more detailed description of the SP ->RB interface as well as RE/Sequencer ->SP interface.

	ORIGINATE DATE 23 January, 2001	EDIT DATE 4 September, 2015	DOCUMENT-REV. NUM. R400 Shader Processor Model	PAGE 4 of 12
--	------------------------------------	--------------------------------	---	-----------------

Introduction

The shader pipeline processes elements (pixels or vertices) in groups “vectors” of sixteen. R400 operates on tiles of 2x2 pixels or quad of pixels. There will be four sets of four shader pipes. For ease of reference and relative positioning of pixels within the quad that each set of shader blocks operates on, we name this sets as UL (upper left), Upper Right (upper right), LL (lower left) and LR (lower right). Please refer to the R400 Shader Processor Model (architectural specification) for an overall functionality of the shaders from the programmer’s view-point.

1. State

1.1 Shader State

1.1.1 GPR

The general purpose registers are 128 bits wide, composed of four 32 bit values. Depending on the operation these values are interpreted at RGBA, or XYZW, or STQW, or UVQW, or YUVA, or.. to simplify matters the only two aliases used here are XYZW and RGBA.

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 mangment of the shader register file is automatic, and not visible to a program executing on a vector, execept that a program is required to declare the number of GPRs it need 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 simultainiously resident. If there are too few vectors resident, then the latency of memory accesses can no longer be hidded and performance suffers.

There are a total of 128 registers. We do not yet know how many registers per vector is too many, and performance starts suffering.

It is possible for a single program/vector to request all 128 registers. This will make it impossible 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 instructions 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.

An open issue is if the pipeline will need GPR0 to store pixel related information. (coverage mask, position, Z, W). If we chose to do this (to avoid having a separate memory for this data) then GPR0 is unavailble as a general register.

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

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

1.1.2 Constant Registers

There are also (192?) constant registers:

127	95	63	31	0	Const
A/W	B/Z	G/Y	R/X		C0
					C1
					C191

These are ONLY available to vertex and pixel shader program in the primary commands stream. They should not be used for real time stream pixel shaders, or 2D shaders.

	ORIGINATE DATE 23 January, 2001	EDIT DATE 4 September, 2015	DOCUMENT-REV. NUM. R400 Shader Processor Model	PAGE 6 of 12
--	------------------------------------	--------------------------------	---	-----------------

The constant registers are 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.

NEW! Constant registers are also used to hold texture/memory fetch state. **NEW!**

To be able to support multiple textures easily, and to save hardware area the texture registers are stored in constant registers. A pair of constant registers hold 256 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 the 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. The driver will probably decide to allocate a fixed number of the constant registers as texture registers.

The constant registers are backed up by control logic to ensure that a pixel sees the correct state, even when partial state updates are completed. I want to save area in this logic by having the updates occur on a 256 bit granularity. Here is how I expect the driver to work: The driver maintains a copy in cacheable system memory of the constant registers. When the driver needs to upload a change of the constant to the chip (just before drawing) it needs to copy two sequential aligned 128 bit words from the system memory version to the indirect buffer, even if only a 32 bit word within the two constant values has changed. Since the CPU will read in at least the full 256 bits into its cache, the only performance penalty will be the second 128 bit write

1.1.3 Previous Instruction

Within a alu clause the result of the previous operation is explicitly available, without requiring a register read. (in fact due an exposed pipeline delay, the result of the previous operation can not be read from the register file without a one instruction delay slot)

This register is not preserved between the end of one alu clause and the begining of another.

It can be used to avoid using another GPR if the result is not needed.

127	95	63	31	0	
A/W	B/Z	G/Y	R/X		Prev

1.1.4 Texture Temporaries

There are two texture temporary registers:

63	47	31	15	0	
A	B	G	R		Tt0
					Tt1

They are used to implement higher order filters (tri-linear, tri-linear (from a volume texture), Bi-cubic, aniso, arbitrary filters)

Tt0 can be viewed as an accumulation buffer. The result of the bi-linear blend can be written into Tt0, after being summed with the value that is already there.

A trilinear filter can be done with two instructions:

Tt0 = texture(address, and rest of state needed, but with mipcntl set to "lower mip level")
R = Tt0 + texture(address, and rest of state needed, but with mipcntl set to "upper mip level")

Volume textures and mipmapped volume textures are implemented in the same way.

Tt1 is used for implementing filers of arbitrary size. For every four samples in the filter two acceses are made, the first access fetches the filter weights, the second fetches the texture values and uses the contents of Tt1 as the weights instead of a bi-linear filter.

We will have explicit support for bi-cubic filters, and seperable filters to avoid the doubled cycles of the previous method.

	ORIGINATE DATE 23 January, 2001	EDIT DATE 4 September, 2015	DOCUMENT-REV. NUM. GEN-CXXXXX-REVA	PAGE 7 of 12
--	------------------------------------	--------------------------------	---------------------------------------	-----------------

1.2 Texture/Memory State

Texture/memory fetch state is stored in the constant registers; each texture uses two sequential 128 bit registers to hold the 256 bits of state per texture.

The contents of these constant registers were stored in normal registers in previous chips.

A very early version of the data stored in a texture/memory constant register is:

Field	size	Description
Min_MIP_level	4	Clamp mip map level to this level
Max_MIP_level	4	Clamp mip map level to this level
First_MIP_level	4	First mip map level in tree. (do we want/need this?)
Clamp_S	3	Clamp/wrap/
Clamp_T	3	
Clamp_W	3	Clamp control for volume textures
Border_mode	1	
Tx_format	5	
Non_power2	1	0- texture is in range 1 to 0 after clamp mirror, must be multiplied by txwidth/txheight (power2) 1- texture has been multiplied by the texture size in the pixel shader. (need to work out how to deal with clamp modes)
TXWIDTH	4	Texture width (or face0 width)
TXHEIGHT	4	Texture height (or face0 height)
TXDEPTH	4	Texture depth (volume textures)
TXWIDTH_f1	4	
TXHEIGHT_f1	4	
TXWIDTH_f2	4	
TXHEIGHT_f2	4	
TXWIDTH_f3	4	
TXHEIGHT_f3	4	
TXWIDTH_f4	4	
TXHEIGHT_f4	4	
TXWIDTH_f5	4	
TXHEIGHT_f5	4	
Alpha_mask?	1	
Chroma_key?	1	
Tex_coord_type	3	
LOD_BIAS	14	
TX_PITCH	14	Number of bits will decrease, used with non power 2 textures
Offset	32	Texture offset (includes endian and tile control)
Limit	32	Any memory accesses > limit will be killed, and the pixel that made the request will also be killed. If the access was from a vertex shader, then the vertex shader will for the x value of the vertex to be NAN which will kill all triangles that attempt to use the vertex.

1.3 Initial state

1.3.1 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 constant registers by the driver.

	ORIGINATE DATE 23 January, 2001	EDIT DATE 4 September, 2015	DOCUMENT-REV. NUM. R400 Shader Processor Model	PAGE 8 of 12
--	------------------------------------	--------------------------------	---	-----------------

1.3.2 Pixel Shader

The pixel shader has the interpolated values generated from the values exported by the vertex shader. If the vertex shader did expy, and the appropriate control bit in the rasterizer is set, then the register 0 contains the x,y,z,w of the pixel (screen space). If the pixel shader wants a world space x,y,z,w the vertex shader should output that.

1.3.3 2D Shader

to be defined

1.3.4 RealTime Shader

To be defined

2. Program Format

A pixel or vertex shader program consists of 16 clauses, eight texture and eight alu. The instructions in a clause will be executed sequentially,

3. ALU instruction format and other instruction related issues

3.1 ALU instruction format

Field	Size	Description
Vector Opcode	5	Opcode
Scalar/Alpha Opcode	7	Opcode for the Scalar or Alpha channel instruction
Scalar Source Select	1	Selection the input for the scalar operation out of SRC B or SRC C when a scalar instruction is coissued with a vector operation. The vector operation has to be 2 source instructions.
SRC A RGB Select	2	
SRC B RGB Select	2	
SRC C RGB Select	2	
SRC A Alpha Select	2	
SRC B Alpha Select	2	
SRC C Alpha Select	2	
SRC A RGB Reg/Constant Pointer	8	Location of Source A in the register file
SRC B RGB Reg/Constant Pointer	8	Location of Source B in the register file
SRC C RGB Reg/Constant Pointer	8	Location of Source C in the register file
SRC A Alpha Reg/Constant Pointer	8	Location of Source A in the register file
SRC B Alpha Reg/Constant Pointer	8	Location of Source B in the register file
SRC C Alpha Reg/Constant Pointer	8	Location of Source C in the register file
SRC A RGB Arg Mod	2	Argument A modifier
SRC B RGB Arg Mod	2	Argument B modifier
SRC C RGB Arg Mod	2	Argument C modifier
SRC A Alpha Arg Mod	2	Argument A modifier on the alpha channel
SRC B Alpha Arg Mod	2	Argument B modifier on the alpha channel
SRC C Alpha Arg Mod	2	Argument C modifier on the alpha channel
SRC A swizzle	12	3 bits for each component
SRC B swizzle	12	3 bits for each component

		ORIGINATE DATE 23 January, 2001	EDIT DATE 4 September, 2015	DOCUMENT-REV. NUM. GEN-CXXXXX-REVA	PAGE 9 of 12
SRC C swizzle	12	3 bits for each component			
Scalar/Alpha Output Mod	3	Output modifier for the result of the Scalar or Alpha channel operation			
Vector Output Modifier	3	Output modifier on the vector result (RGB) when alpha operation is being coissued or ARGB when scalar operation is being coissued			
Scalar/Alpha Clamp	1				
Vector Clamp	1				
Scalar/Alpha Write Mask	4	Defines which out of 32 bits words (four of them) in the result is written back in the register file			
Vector Write Mask		Defines which out of 32 bits words (four of them) in the result is written back			
Scalar/Alpha result pointer		Specifies the address into the register files for result of scalar/alpha operation			
Vector result pointer					
Scalar Export flag		TBD			
Vector Export flag		TBD			

3.2 ALU Opcodes

The following opcodes are native, core opcodes:

Name	Function	Notes
MACC	$D = A * B + C$	(add is $A*1.0 + C$) (mul is $A*B + 0.0$) (nop/passthrough/move is $A*1.0 + 0.0$)
MSUB	$D = A * B - C$	(sub is $A*1.0 - C$)
MRSB	$D = C - A*B$	
DOT2	$D = (A.x * B.x) + (A.y * B.y)$	
DOT3	$D = (A.x * B.x) + (A.y * B.y) + (A.z * B.z)$	
DOT4	$D = (A.x * B.x) + (A.y * B.y) + (A.z * B.z) + (A.w * B.w)$	
RECP	$D = 1/A.w$	Use broadcast to select something other than w
CMxx	$D = A$ if $C \leq 0.0$, else B	Xx can be: gt,gte,eq (lt,lte,ne can be generated by swapping a and b)
CLMP	$D = A$ if $(B > A > C)$ else B if $(A > B)$ B else C	
ABSV	$D = A$ if $A > 0$ else B	
CEIL	$D = A$; the smallest integer D such that $D > A$	
FLOR	$D = A$ truncated	
RndV	$D = A$ rounded to the nearest integer	
FRAC	$D = A - \text{floor}(A)$	
MINV	$D = \min(A,B)$	
MAXV	$D = \max(A,B)$	
Area	$D = \text{area}(A)$	Possible opcode: Sets D to $A.w, A.w, A.w, A.w$ where each A is from a different pixel in the quad. Can be used to calculate area for LOD calculations, usefull for antialiasing procedural shaders.
Exp	$D = \text{Pow}(a)$	Possible opcode, otherwise texture lookup
RSQR	$D = 1/\text{sqrt}(a)$	Possible opcode, otherwise texture lookup
Log2	$D = \text{Log}(A, \text{base} = 2)$	Possible opcode, otherwise table lookup
Log	$D = \text{Log}(A, \text{base} = B)$	Possible opcode, otherwise table lookup
Pow	$D = A$ to the B 'th power	
SCLP	$D = ABC$	Concatenate clip code test results into a single DWORD
CUBE	$D = A$	Find the largest of x,y,z , place the reciprocal of that value in w , set x and y to the two remaining values, and identify the face as the integer 0 to 5 in Z . Used to setup for a cube map.

					ORIGINATE DATE 23 January, 2001	EDIT DATE 4 September, 2015	DOCUMENT-REV. NUM. R400 Shader Processor Model	PAGE 10 of 12
	The export opcodes must be in the last alu clause:							
EXPP	Export position: position = A, clipcodes = B						Export position and clip codes of vertex, end vertex shader	
EXP	Export vertex component, {D} = A						Export vertex component to vertex cache, 16 possible destinations	
EXPXY	Placeholder, {0} = 1,0,0,0						Placeholder for the rasterizer to put pixel position and Z into pixel shader.	
EXPC	Export color						Export color of pixel, end pixel shader	
EXPZ	Export Z						Export Z of pixel	

There are some extraction and data conversion opcodes to be added.
There will also be a specialized opcode for cube maps.

The export opcodes will be removed in a future version, and replaces with new destinations for general operations.
Exports must still be the last operations executed.

3.3 Macro opcodes

These instructions are NOT implemented in the R400. But their functionality can be implemented with the shown opcodes.

4. Texture/Memory

4.1 Instruction Format

Destination control:

Field	Size	Description
Initialize	1	If set, destination register is set to 1,0,0,0 before any writes are made (w,z,y,x) (a,b,g,r)
Wmask	4	Write mask for result of lookup
Channels	2	0: 1 channel, duplicate across all four channels 1: 2 channels d.x,d.y = a, d.z,d.w = b 2: 2 channels d.x,d.z = a, d.y,d.w = b 3: 4 channels
Data_Format	4	0: unsigned int 1: none- just write to destination register others (Z, apple YUV, mpeg, etc.)
Bias	1?	Do we want a bias other than -128?
Scale	>4	Result value is (range(s)+bias)*scale
Range	1	0 to 1 or 0 to 255/256 (or 65335/65336 etc..)
Texture	7	Which texture we want to fetch from
Texture_t	1	Texture or linear memory array (if linear array, then only offset and limit are noticed, point sampling is forced, format is 32bpp)
Dest	7	Destination address of texture/memory fetch
Src	7	Source register for address
Swizzel	5	Which part of source register contains texture coordinate
MAG_Filter	1	0- Nearest 1- Linear
Min_Filter	1	0- Nearest 1- Linear
Mip_filter	2	0- Disabled 1- Enabled
Volume_filter	1	Filtering between volume texture levels

	ORIGINATE DATE 23 January, 2001	EDIT DATE 4 September, 2015	DOCUMENT-REV. NUM. GEN-CXXXXX-REVA	PAGE 11 of 12
		0- Nearest 1- Linear		
MIP_enable	1	0- no mipmapping 1- mip-mapping		
Filter_mode	2	0- treat input as four 8bit values 1- treat input as two 16bit values 2- treat input as one 32 bit value 3- input is not filtered		
Scale	4	Multiply the x (and y?) components by this value, used for vertex fetching		
Offset	4	Add this integer to the x component, after the scale, used for vertex fetching.		
Combine mode	2	0: R = R 1: R = R + Tt0 2: Use Tt1 values as blend factors 3:		
Output mode	2	0: Do not store result of fetch (NOP) 1: store result in GPR 2: store result in Tt0 3: store result in Tt1		
Mip mode	2	0: normal 2: lower mip filter and bias 3: upper mip filter and bias		
Tex3D mode	2	0: normal (non volume) 2: lower z fetch 3: upper z fetch		
Sample_bias_x	4	2's complement offset for bi-linear sample. 0 will generate a normal fetch, a positive or negative number will fetch the 2x2 samples that distance in 2x2s away. Used to impement bi-cubic and arbitrary sized filters		
Sample_bias_y	4	2's complement offset for bi-linear sample. 0 will generate a normal fetch, a positive or negative number will fetch the 2x2 samples that distance in 2x2s away. Used to impement bi-cubic and arbitrary sized filters		

We can move fields between here and the constant register that hold the rest of the texture fetch state.

4.2 Opcodes

Name	Function	Notes
TF	Texture fetch	
CTF	Cube texture fetch	
AF	Array fetch	Used for vertex array fetches, ignores most of the state in the texture constant registers, allow driver to only store offset+limit values in constant register

4.3 Example

To do a vertex fetch the instruction might be as follows:

Field	Size	Description
Initialize	1	Set for first fetch to each gpr.
Wmask	4	Set to write to correct element of gpr- ie if fetching x, would be set to 1000
Channels	2	0: 1 channel, duplicate across all four channels
Data_Format	4	1: none- just write to destination register or 2? Color, unpack to rgba
Bias	1?	0
Scale	>4	0
Range	1	0 to 255/256
Texture	7	Which texture we want to fetch from
Texture_t	1	linear memory array
Dest	7	Destination address of texture/memory fetch

	ORIGINATE DATE 23 January, 2001	EDIT DATE 4 September, 2015	DOCUMENT-REV. NUM. R400 Shader Processor Model	PAGE 12 of 12
--	------------------------------------	--------------------------------	---	------------------

Src	7	Source register for address
Swizzel	5	Which part of source register contains texture coordinate
MAG_Filter	1	2- Nearest
Min_Filter	1	2- Nearest
Mip_filter	2	2- Disabled
Volume_filter	1	Filtering between volume texture levels 2- Nearest
MIP_enable	1	2- no mipmapping
Filter_mode	2	4- input is not filtered
Scale	4	Multiply the x (and y?) components by this value, used for vertex fetching
Offset	4	Add this integer to the x component, after the scale, used for vertex fetching.

And the constant register pair that texture points to would contain:

Field	size	Description
Min_MIP_level	4	NA
Max_MIP_level	4	NA
First_MIP_level	4	NA
Clamp_S	3	NA
Clamp_T	3	NA
Clamp_w	3	NA
Border_mode	1	NA
Tx_format	5	32bpp
Non_power2	1	2- texture has been multiplied by the texture size in the pixel shader. (need to work out how to deal with clamp modes)
TXWIDTH	4	Texture width (or face0 width)
TXHEIGHT	4	NA
TXDEPTH	4	NA
TXWIDTH_f1	4	NA
TXHEIGHT_f1	4	NA
TXWIDTH_f2	4	NA
TXHEIGHT_f2	4	NA
TXWIDTH_f3	4	NA
TXHEIGHT_f3	4	NA
TXWIDTH_f4	4	NA
TXHEIGHT_f4	4	NA
TXWIDTH_f5	4	NA
TXHEIGHT_f5	4	NA
Alpha_mask?	1	NA
Chroma_key?	1	NA
Tex_coord_type	3	?:1D array
LOD_BIAS	14	0
TX_PITCH	14	Number of bits will decrease, used with non power 2 textures
Offset	32	Texture offset (includes endian and tile control)
Limit	32	Any memory accesses > limit will be killed, and the pixel that made the request will also be killed. If the access was from a vertex shader, then the vertex shader will for the x value of the vertex to be NAN which will kill all triangles that attempt to use the vertex.