	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 201544	GEN-CXXXXX-REVA	1 of 51
Author:	Laurent Lefebvre			
Issue To:		Copy No:		
	R400 S	equencer Spe	cification	
	א איז איז איז איז איז איז איז איז איז אי	ายา เอรี เอม เอะ ห ห เอา เอะ ห เอะ โอะ เอ	מיזעטר איזאי אעטר פטאיאישטר איז	
		SQ		
		20		
		Version 2.07		
		Version 2.07		
Overview: Thi	s is an architectural specific	cation for the R400 Sequen	cer block (SEQ). It provides an ov	erview of the
	uired capabilities and expe cks, and provides internal st		Iso describes the block interfaces,	internal sub-
Δυτομάτις	ALLY UPDATED FIELDS:			
Document L	ocation: C:\pe		locks\sq\R400_Sequencer.doc	
	anet Search Title: R400	0 Sequencer Specification APPROVALS		
	Name/Dept		Signature/Date	
Remarks:				
THIS DO	CUMENT CONTAIN	IS CONFIDENTIAL I	NFORMATION THAT CO	ULD BE
	NTIALLY DETRIME	INTAL TO THE INTE	REST OF ATI TECHNOL	
	INC. THROUGH L	JNAUTHORIZED US	SE OR DISCLOSURE.	
			rial in this document constitutes an provide notice that ATI owns a cop	
unpublished w	ork. The copyright notice	is not an admission that	publication has occurred. This w of this document may be used, re	ork contains
			sion of ATI Technologies Inc."	produced, or
				_
Exhibit 2035.do	cR400_Sequencer.doc 73569 Bytes*** 🤅	ATI Confidential. Referer	nce Copyright Notice on Cover Pa	ge © ***

ATI 2035 LG v. ATI IPR2015-00325

AMD1044_0257711

ATI Ex. 2109 IPR2023-00922 Page 1 of 326

	2	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
	<u>u</u>	24 September, 2001	<u>4 September, 201544</u>		2 of 51
Tab 1. 1.1 1.2	OVER Top Le	evel Block Diagram			9
1.2					
2.	INTER	RPOLATED DATA BU	JS		11
3.					
4. 5.				*****	
5.1	Memo	ry organizations			14
5.2					
5.3					
			-	t	
		,			
5.4 5.5		-			
5.6					
6.					
6.1 6.2		•			
6.3					
6.4 6.5	Data c	lependant predicate i	nstructions		24
6.6 6.7					
6.	7.1 r	Method 1: Debugging	registers		
6.	7.2 r	Method 2: Exporting tl	ne values in the GPR	\$	
7.					
8. 9.					
10.		H ARBITRATION			
11.	ALU A	ARBITRATION	*****		
12.					
13. 14.				IFOS	
15.					
15.1 16.					
				erence Copyright Notice on Cover F	

ATI Ex. 2109 IPR2023-00922 Page 2 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001		GEN-CXXXXX-REVA	3 of 51
17. THE F	• • •	4 September, 201514		

17.1.1				
17.1.2				
17.1.3				
	tration restrictions			32

	•			
19.3.2				

23.2.1	SC_SP#			
23.2.2	SC_SQ			
23.2.3	SQ to SX(SP): Interpo	olator bus		39
23.2.4				
23.2.5	VGT to SQ : Vertex in	terface		39
23.2.6	SQ to SX: Control bus	S		
23.2.7	SX to SQ : Output file	control		
23.2.8	SQ to TP: Control bus	\$		
23.2.9	TP to SQ: Texture sta			43
23.2.10	SQ to SP: Texture sta	all		
23.2.11	SQ to SP: GPR and a	uto counter		44
23.2.13	SP to SQ: Constant a	ddress load/ Predicate	Set/Kill set	
23.2.14	SQ to SPx: constant I	proadcast		
23.2.15	SQ to CP: RBBM bus	i		
23.2.16	CP to SQ: RBBM bus	i		
23.3 Exa	mple of control flow pro	gram execution		
		•	ce Copyright Notice on Cover Pa	

ATI Ex. 2109 IPR2023-00922 Page 3 of 326

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
	24 September, 2001	4 September, 201514		4 of 51
24. OPEN	I ISSUES			

Exhibit 2035.docR400_Sequencer.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257714

ATI Ex. 2109 IPR2023-00922 Page 4 of 326

	ORIGINATI	E DATE	EDIT DATE	DOCUMENT-RE	/. NUM.	PAGE
CAU	24 Septemb	er, 2001	4 September, 201514	GEN-CXXXXX-	REVA	5 of 51
Revision	Changes	S S Soom				
Rev 0.1 (Laur Date: May 7, 2	r ent Lefebvre) 2001		First o	raft.		
Rev 0.2 (Laure Date : July 9, 2 Rev 0.3 (Laure Date : August Rev 0.4 (Laure Date : August Rev 0.5 (Laure Date : Septem Rev 0.6 (Laure Date : Septem Rev 0.7 (Laure Date : Octobel Rev 0.8 (Laure	2001 ent Lefebvre) 6, 2001 ent Lefebvre) 24, 2001 ent Lefebvre) nber 7, 2001 ent Lefebvre) nber 24, 2001 ent Lefebvre) r 5, 2001 ent Lefebvre) ent Lefebvre)		SP. A Revie Augue Addee file ar flow o Addee Chan archit Addee store data o Chan	Changed the interfaces to reflect the changes in the SP. Added some details in the arbitration section. Reviewed the Sequencer spec after the meeting on August 3, 2001. Added the dynamic allocation method for register file and an example (written in part by Vic) of the flow of pixels/vertices in the sequencer. Added timing diagrams (Vic) Changed the spec to reflect the new R400 architecture. Added interfaces. Added constant store management, instruction store management, control flow management and data dependant predication. Changed the control flow method to be more		
Date : Octobe Rev 0.9 (Laure Date : Octobe	ent Lefebvre) r 17, 2001		Incorp flow r the o regist		the 10/18/01 instruction, re mp. Added	moved debug
Rev 1.0 (Laure Date : Octobel Rev 1.1 (Laure Date : Octobel	r 19, 2001 ent Lefebvre) r 26, 2001		Addeo precis Metho	ed interfaces to RB. Adde I SEQ→SP0 interface ion. Changed VGT→SF ds added.	s. Changed 0 interface.	delta Debug
Rev 1.2 (Laure Date : Novemi Rev 1.3 (Laure Date : Novemi	ber 16, 2001 ent Lefebvre)			ces greatly refined. Clear I the different interpolation		C.
Rev 1.4 (Laure Date : Decemi Rev 1.5 (Laure	ent Lefebvre) ber 6, 2001		the Vi mana	I the auto incrementing GT→SQ interface. Added gement. Updated GPRs. ved from the spec all in	content on co	onstant
Date : Decemi	ber 11, 2001		direct const synch	y tied to the SQ. Add int management. ronization fields and expla	ed explanatio Added P. anation.	ons on A→SQ
Rev 1.6 (Laure Date : January	y 7, 2002		detail call ii Addeo updat	I more details on the sta about the parameter ca istruction to a Condition I details on constant ed the diagram.	ches. Chang nnal_call instr managemen	ed the ruction. it and
Rev 1.7 (Laure Date : Februar Rev 1.8 (Laure Date : March 4	ry 4, 2002 ent Lefebvre)		interfa New i	I Real Time parameter ce. Updated the control fl nterfaces to the SX bloc e modifier, removed t stions.	ow section. k. Added the	end of
Date : March 2	18, 2002 irrent Lefebvre) 25, 2002 irrent Lefebvre) 0, 2002 ent Lefebvre)		Reara ensur Updat expor Addeo spec v	ngement of the CF instru- byte alignement. ed the interfaces and a ing rules. I CP state report interfact vith the old control flow so ontrol flow scheme	added a sect e. Last versior	ion on
Exhibit 2035.docR	1400_Sequencer.doc 7	73569 Bytes*** C	ATI Confidential. Refe	ence Copyright Notice	on Cover Pag	e © ***

ATI Ex. 2109 IPR2023-00922 Page 5 of 326

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
	24 September, 2001	4 September, 201514		6 of 51
Rev 2.01 (Lau	rent Lefebvre)	Cha	nged slightly the control flow instru	uctions to
Date : May 2,	2002	allov	v force jumps and calls.	
Rev 2.02 (Lau	rent Lefebvre)	Upd	ated the Opcodes. Added type fie	ld to the
Date : May 13	, 2002	cons	stant/pred interface. Added Last fie	ld to the
		SQ-	→SP instruction load interface.	
Rev 2.03 (Lau	rent Lefebvre)	SP	interface updated to include p	redication
Date : July 15	, 2002	•	nizations. Added the predicate	no stall
			uctions,	
(rent Lefebvre)		umented the new parameter generatio	n scheme
Date :August 2			Y coordinates points and lines STs.	
	rent Lefebvre)		e interface changes and an ar	chitectural
Date : Septem			nge to the auto-counter scheme.	
	rent Lefebvre)		ened the event interface to 5 bits. So	ome other
Date : Octobe	,		typos corrected.	
***************************************	rent Lefebvre)		os, jumps and calls are now using	
Date : Octobe	<u>r 14, 2002</u>		ress which allows to jump and call	
			nd any control flow addresses (<u>does not</u>
		requ	iires to be even anymore).	

Exhibit 2035.doc.R400_Sequences.doc 73569 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257716

ATI Ex. 2109 IPR2023-00922 Page 6 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
	24 September, 2001	4 September, 201514	GEN-CXXXXX-REVA	7 of 51	

1. Overview

The sequencer chooses two ALU threads and a fetch hread to execute, and executes all of the instructions in a block before looking for a new clause of the same type. Two ALU threads are executed interleaved to hide the ALU latency. The arbitrator will give priority to older threads. There are two separate reservation stations, one for pixel vectors and one for vertices vectors. This way a pixel can pass a vertex and a vertex can pass a pixel.

To support the shader pipe the sequencer also contains the shader instruction cache, constant store, control flow constants and texture state. The four shader pipes also execute the same instruction thus there is only one sequencer for the whole chip.

The sequencer first arbitrates between vectors of 64 vertices that arrive directly from primitive assembly and vectors of 16 quads (64 pixels) that are generated in the scan converter.

The vertex or pixel program specifies how many GPRs it needs to execute. The sequencer will not start the next vector until the needed space is available in the GPRs.

Exhibit 2035. doc R400_Sequencer.doc 73569 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257717

ATI Ex. 2109 IPR2023-00922 Page 7 of 326

PROTECTIVE ORDER MATERIAL

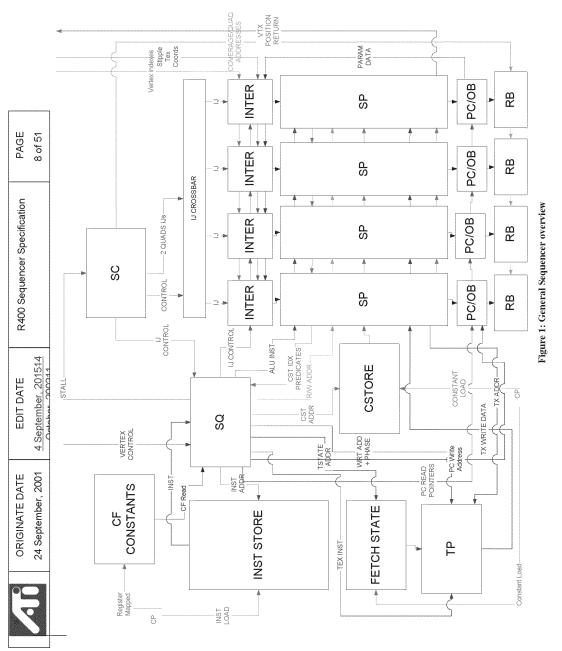


Exhibit 2035 doc/8400_Sequencer.doo 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257718

ATI Ex. 2109 IPR2023-00922 Page 8 of 326

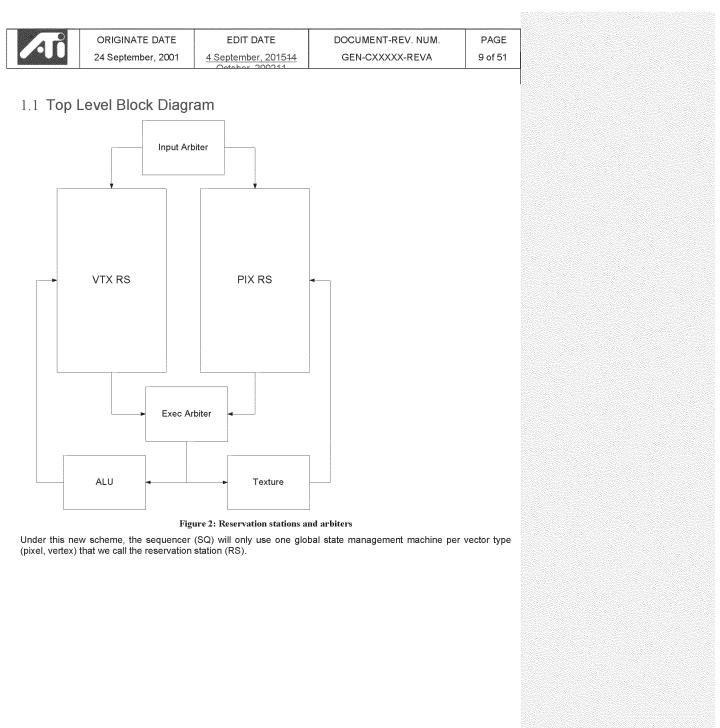
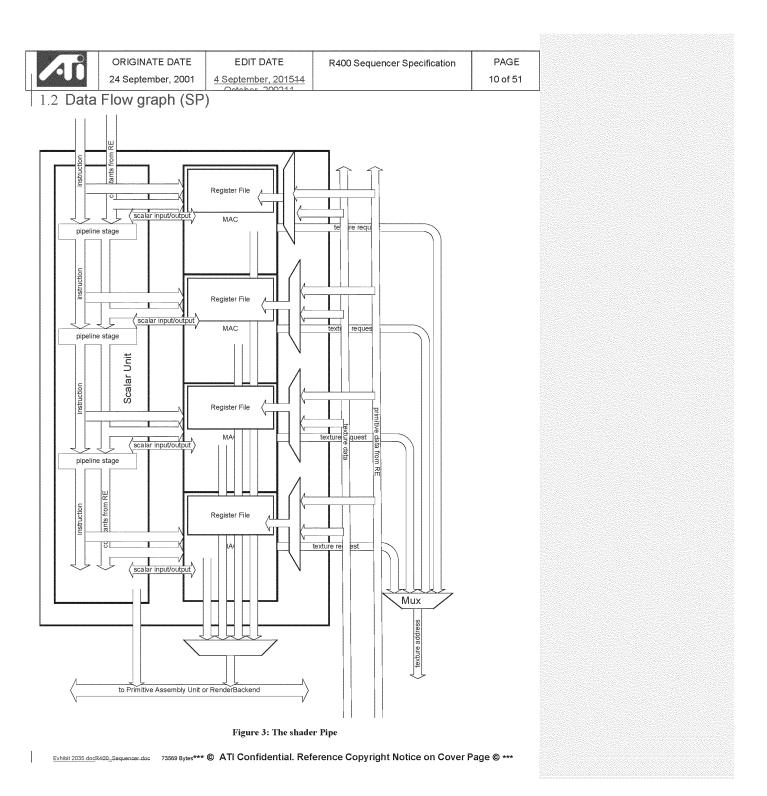


Exhibit 2035.docR400_Sequencer.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257719

ATI Ex. 2109 IPR2023-00922 Page 9 of 326



ATI Ex. 2109 IPR2023-00922 Page 10 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 201544	GEN-CXXXXX-REVA	11 of 51

The gray area represents blocks that are replicated 4 times per shader pipe (16 times on the overall chip).

1.3 Control Graph

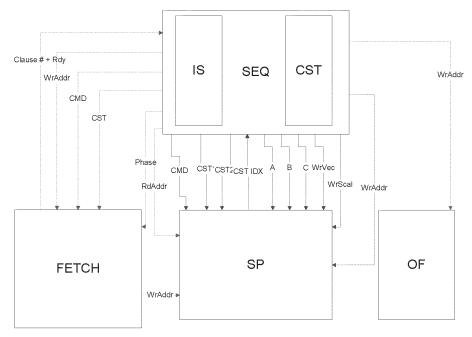


Figure 4: Sequencer Control interfaces

In green is represented the Fetch control interface, in red the ALU control interface, in blue the Interpolated/Vector control interface and in purple is the output file control interface.

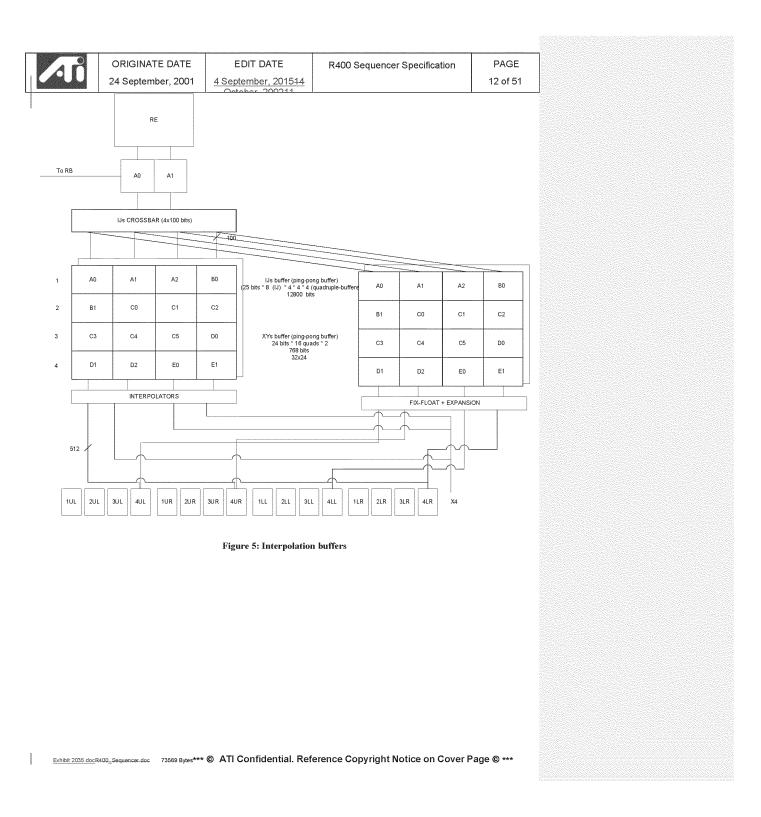
2. Interpolated data bus

The interpolators contain an IJ buffer to pack the information as much as possible before writing it to the register file.

Exhibit 2035. docR400_Sequences.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257721

ATI Ex. 2109 IPR2023-00922 Page 11 of 326



ATI Ex. 2109 IPR2023-00922 Page 12 of 326

PROTECTIVE ORDER MATERIAL

		T23					51 - 48 <	52- 55	56- 59	≥ ° 8	
		T22					V V 32-48- 35 51	အ အွဲ <	V V 40-56- 43 59	V V 44-60- 47 63	X
		T21					> 16- 19	23 ² 0 <	24- 27	31 -28 <	× 15
		T19 T20 T21 T22 T23					> 0-3	> 4-7	, ⇒ ⇔ ,	→ 15 15 <	
		T19							ΕO	Ш	
		T18								8	
щ	51	T17			≳ ଘ	×Π		8	5	ß	
PAGE	13 of 51	T16			В	ш				BO	P2
		T15			ШO	Ш Т	5	D2			۵
NUM.	VA	T14	₹₽	≳8		×۵	ប៊	C4	C5		
DOCUMENT-REV. NUM.	GEN-CXXXX-REVA	T13	δ	D2		ß	B1				
IENT-F	(XXX)	T12	Б	6		ß	AO	A1	A2		
ocuN	GEN-(T10 T11 T12		≿ 2	≿ ນິ				о Ш	Ш	******************
ă		T10	É	5	C5	0	¥			8	
	4	T9		5	C5			8	5	ß	
ΞĽ	2015	Т8	≿ ເິ	≳ 8	≿ ວ	≿ 8				BO	demaana
EDIT DATE	mber,	11	ខ	8	δ	3	5	D2			À
	4 September, 201514	T6	ទ	8	ъ	3	ប៊	5 2	C5		
	~	Т5	×₽			×₿	B1				
ATE	2001	Т4	Bi			BO	AO	A1	A2		
ATE D	mber,	T3	Ð			BO	X 48- 51	XY 52- 55	56 - 56 -	3°X	
RIGINATE DATE	September, 2001	Т2	¥ 8	≿₽	Σ₹		35 23 ₹	အ အွ X	∑ \$ £	XY XY 44- 60- 47 63	\geq
ō	24	11	Ao	A1	A2		, † 19 19	3 \$ X	24-27	3 % X	×
9		D 10	Ao	A1	A2		53 X	× × -4	≿ ∾ ≿	15 ¦2 ≷	
K	3		Ч ^S о	с –	dS ∽	ი წ	с о	с С	р С С	ი ი	

Figure 6: Interpolation timing diagram

Exhibit 2035 doe R400_Sequences 73589 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257723

ATI Ex. 2109 IPR2023-00922 Page 13 of 326

2	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
JU	24 September, 2001	4 September, 201514		14 of 51

Above is an example of a tile the sequencer might receive from the SC. The write side is how the data get stacked into the XY and IJ buffers, the read side is how the data is passed to the GPRs. The IJ information is packed in the IJ buffer 4 quads at a time or two clocks. The sequencer allows at any given time as many as four quads to interpolate a parameter. They all have to come from the same primitive. Then the sequencer controls the write mask to the GPRs to write the valid data in.

3. Instruction Store

There is going to be only one instruction store for the whole chip. It will contain 4096 instructions of 96 bits each.

It is likely to be a 1 port memory; we use 1 clock to load the ALU instruction, 1 clocks to load the Fetch instruction, 1 clock to load 2 control flow instructions and 1 clock to write instructions.

The instruction store is loaded by the CP thru the register mapped registers.

The VS_BASE and PS_BASE context registers are used to specify for each context where its shader is in the instruction memory.

For the Real time commands the story is quite the same but for some small differences. There are no wrap-around points for real time so the driver must be careful not to overwrite regular shader data. The shared code (shared subroutines) uses the same path as real time.

4. Sequencer Instructions

All control flow instructions and move instructions are handled by the sequencer only. The ALUs will perform NOPs during this time (MOV PV,PV, PS,PS) if they have nothing else to do.

5. Constant Stores

5.1 Memory organizations

A likely size for the ALU constant store is 1024x128 bits. The read BW from the ALU constant store is 128 bits/clock and the write bandwidth is 32 bits/clock (directed by the CP bus size not by memory ports).

The maximum logical size of the constant store for a given shader is 256 constants. Or 512 for the pixel/vertex shader pair. The size of the re-mapping table is 128 lines (each line addresses 4 constants). The write granularity is 4 constants or 512 bits. It takes 16 clocks to write the four constants. Real time requires 256 lines in the physical memory (this is physically register mapped).

The texture state is also kept in a similar memory. The size of this memory is 320x96 bits (128 texture states for regular mode, 32 states for RT). The memory thus holds 128 texture states (192 bits per state). The logical size exposes 32 different states total, which are going to be shared between the pixel and the vertex shader. The size of the re-mapping table to for the texture state memory is 32 lines (each line addresses 1 texture state lines in the real memory). The CP write granularity is 1 texture state lines (or 192 bits). The driver sends 512 bits but the CP ignores the top 320 bits. It thus takes 6 clocks to write the texture state. Real time requires 32 lines in the physical memory (this is physically register mapped).

The control flow constant memory doesn't sit behind a renaming table. It is register mapped and thus the driver must reload its content each time there is a change in the control flow constants. Its size is 320*32 because it must hold 8 copies of the 32 dwords of control flow constants and the loop construct constants must be aligned.

The constant re-mapping tables for texture state and ALU constants are logically register mapped for regular mode and physically register mapped for RT operation.

Exhibit 2035.docR400_Sequencer.doc 73669 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257724

ATI Ex. 2109 IPR2023-00922 Page 14 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 201514	GEN-CXXXXX-REVA	15 of 51

5.2 Management of the Control Flow Constants

The control flow constants are register mapped, thus the CP writes to the according register to set the constant, the SQ decodes the address and writes to the block pointed by its current base pointer (CF_WR_BASE). On the read side, one level of indirection is used. A register (SQ_CONTEXT_MISC.CF_RD_BASE) keeps the current base pointer to the control flow block. This register is copied whenever there is a state change. Should the CP write to CF after the state change, the base register is updated with the (current pointer number +1)% number of states. This way, if the CP doesn't write to CF the state is going to use the previous CF constants.

5.3 Management of the re-mapping tables

5.3.1 R400 Constant management

The sequencer is responsible to manage two re-mapping tables (one for the constant store and one for the texture state). On a state change (by the driver), the sequencer will broadside copy the contents of its re-mapping tables to a new one. We have 8 different re-mapping tables we can use concurrently.

The constant memory update will be incremental, the driver only need to update the constants that actually changed between the two state changes.

For this model to work in its simplest form, the requirement is that the physical memory MUST be at least twice as large as the logical address space + the space allocated for Real Time. In our case, since the logical address space is 512 and the reserved RT space can be up to 256 entries, the memory must be of sizes 1280 and above. Similarly the size of the texture store must be of 32*2+32 = 96 entries and above.

5.3.2 Proposal for R400LE constant management

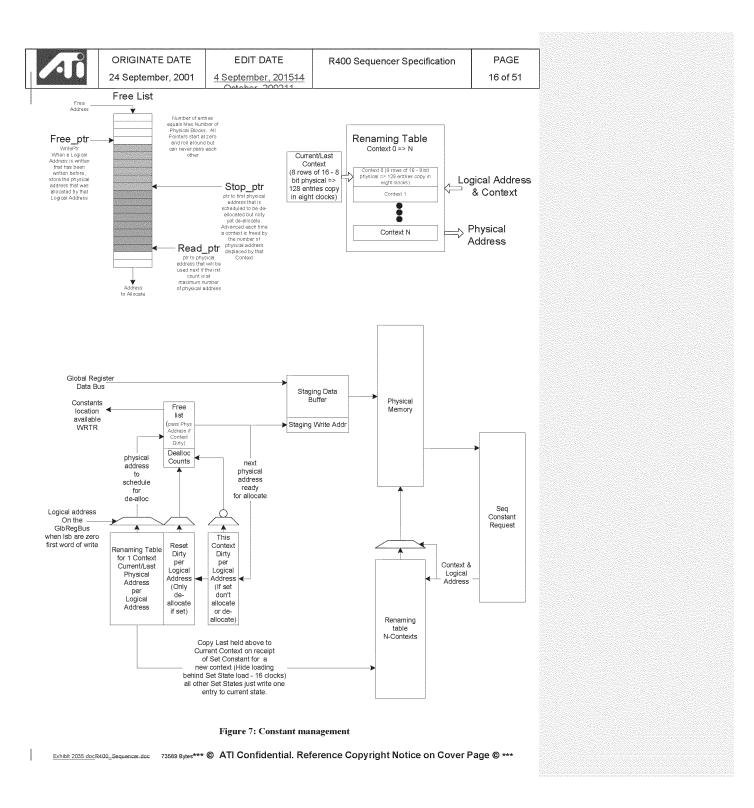
To make this scheme work with only 512+256 = 768 entries, upon reception of a CONTROL packet of state + 1, the sequencer would check for SQ_IDLE and PA_IDLE and if both are idle will erase the content of state to replace it with the new state (this is depicted in <u>Figure 8: De-allocation mechanismFigure 8: De-allocation mechanismFigure 8: De-allocation mechanismFigure 8: De-allocation mechanismFigure 8: De-allocation content of 0 is written to the corresponding de-allocation counter location so that when the SQ is going to report a state change, nothing will be de-allocated upon the first report.</u>

The second path sets all context dirty bits that were used in the current state to 1 (thus allowing the new state to reuse these physical addresses if needed).

Exhibit 2035. docR400_Sequencer.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257725

ATI Ex. 2109 IPR2023-00922 Page 15 of 326



ATI Ex. 2109 IPR2023-00922 Page 16 of 326

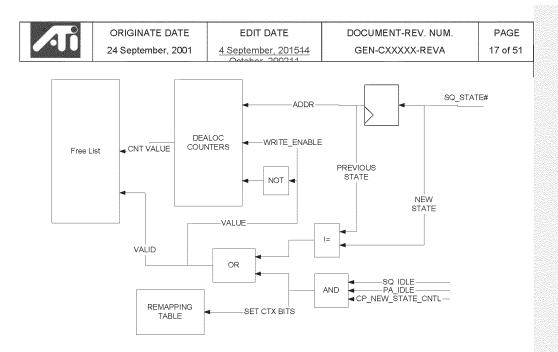


Figure 8: De-allocation mechanism for R400LE

5.3.3 Dirty bits

Two sets of dirty bits will be maintained per logical address. The first one will be set to zero on reset and set when the logical address is addressed. The second one will be set to zero whenever a new context is written and set for each address written while in this context. The reset dirty is not set, then writing to that logical address will not require de-allocation of whatever address stored in the renaming table. If it is set and the context dirty is not set, then the physical address is necessary to store the incoming data. If they are both set, then the data will be written into the physical address held in the renaming for the current logical address. No de-allocation or allocation takes place. This will happen when the driver does a set constant twice to the same logical address between context changes. NOTE: It is important to detect and prevent this, failure to do it will allow multiple writes to allocate all physical memory and thus hang because a context will not fit for rendering to start and thus free up space.

5.3.4 Free List Block

A free list block that would consist of a counter (called the IFC or Initial Free Counter) that would reset to zero and incremented every time a chunk of physical memory is used until they have all been used once. This counter would be checked each time a physical block is needed, and if the original ones have not been used up, us a new one, else check the free list for an available physical block address. The count is the physical address for when getting a chunk from the counter.

Storage of a free list big enough to store all physical block addresses.

Maintain three pointers for the free list that are reset to zero. The first one we will call write_ptr. This pointer will identify the next location to write the physical address of a block to be de-allocated. Note: we can never free more physical memory locations than we have. Once recording address the pointer will be incremented to walk the free list like a ring.

The second pointer will be called stop_ptr. The stop_ptr pointer will be advanced by the number of address chunks de-allocates when a context finishes. The address between the stop_ptr and write_ptr cannot be reused because they are still in use. But as soon as the context using then is dismissed the stop_ptr will be advanced.

The third pointer will be called read_ptr. This pointer will point will point to the next address that can be used for allocation as long as the read_ptr does not equal the stop_ptr and the IFC is at its maximum count.

Exhibit 2035.docR400_Sequencer.doc 73699 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257727

ATI Ex. 2109 IPR2023-00922 Page 17 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 201514		18 of 51

5.3.5 De-allocate Block

This block will maintain a free physical address block count for each context. While in current context, a count shall be maintained specifying how many blocks were written into the free list at the write_ptr pointer. This count will be reset upon reset or when this context is active on the back and different than the previous context. It is actually a count of blocks in the previous context that will no longer be used. This count will be used to advance the write_ptr pointer to make available the set of physical blocks in one clock.

5.3.6 Operation of Incremental model

The basic operation of the model would start with the write_ptr, stop_ptr, read_ptr pointers in the free list set to zero and the free list counter is set to zero. Also all the dirty bits and the previous context will be initialized to zero. When the first set constants happen, the reset dirty bit will not be set, so we will allocate a physical location from the free list counter is not at the max value. The data will be written into physical address zero. Both the additional copy of the renaming table and the context zeros of the big renaming table will be updated for the logical address that was written by set start with physical address of 0. This process will be repeated for any logical address that are not dirty until the context changes. If a logical address is hit that has its dirty bits set while in the same context, both dirty bits would be set, so the new data will be over-written to the last physical address assigned for this logical address. When the first draw command of the context is detected, the previous context stored in the additional renaming table will be copied to the larger renaming table in the current (new) context location. Then the set constant logical address it replaced in the renaming table would be entered at the write_ptr pointer location on the free list and the write_ptr will be incremented. The de-allocation counter for the previous context (eight) will be incremented. This as set states come in for this context one of the following will happen:

- 1.) No dirty bits are set for the logical address being updated. A line will be allocated of the free-list counter or the free list at read_ptr pointer if read_ptr != to stop_ptr .
- 2.) Reset dirty set and Context dirty not set. A new physical address is allocated, the physical address in the renaming table is put on the free list at write_ptr and it is incremented along with the de-allocate counter for the last context.
- 3.) Context dirty is set then the data will be written into the physical address specified by the logical address.

This process will continue as long as set states arrive. This block will provide backpressure to the CP whenever he has not free list entries available (counter at max and stop_ptr == read_ptr). The command stream will keep a count of contexts of constants in use and prevent more than max constants contexts from being sent.

Whenever a draw packet arrives, the content of the re-mapping table is written to the correct re-mapping table for the context number. Also if the next context uses less constants than the current one all exceeding lines are moved to the free list to be de-allocated later. This happens in parallel with the writing of the re-mapping table to the correct memory.

Now preferable when the constant context leaves the last ALU clause it will be sent to this block and compared with the previous context that left. (Init to zero) If they differ than the older context will no longer be referenced and thus can be de-allocated in the physical memory. This is accomplished by adding the number of blocks freed this context to the stop_ptr pointer. This will make all the physical addresses used by this context available to the read_ptr allocate pointer for future allocation.

This device allows representation of multiple contexts of constants data with N copies of the logical address space. It also allows the second context to be represented as the first set plus some new additional data by just storing the delta's. It allows memory to be efficiently used and when the constants updates are small it can store multiple context. However, if the updates are large, less contexts will be stored and potentially performance will be degraded. Although it will still perform as well as a ring could in this case.

5.4 Constant Store Indexing

In order to do constant store indexing, the sequencer must be loaded first with the indexes (that come from the GPRs). There are 144 wires from the exit of the SP to the sequencer (9 bits pointers x 16 vertexes/clock). Since the data must pass thru the Shader pipe for the float to fixed conversion, there is a latency of 4 clocks (1 instruction)

Exhibit 2035.docR400_Sequencer.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257728

ATI Ex. 2109 IPR2023-00922 Page 18 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 201514	GEN-CXXXXX-REVA	19 of 51

between the time the sequencer is loaded and the time one can index into the constant store. The assembly will look like this

MOVA R1.X,R2.X // Loads the sequencer with the content of R2.X, also copies the content of R2.X into R1.X NOP // latency of the float to fixed conversion ADD R3,R4,C0[R2.X]// Uses the state from the sequencer to add R4 to C0[R2.X] into R3

Note that we don't really care about what is in the brackets because we use the state from the MOVA instruction. R2.X is just written again for the sake of simplicity and coherency.

The storage needed in the sequencer in order to support this feature is 2*64*9 bits = 1152 bits.

5.5 Real Time Commands

The real time commands constants are written by the CP using the register mapped registers allocated for RT. It works is the same way than when dealing with regular constant loads BUT in this case the CP is not sending a logical address but rather a physical address and the reads are not passing thru the re-mapping table but are directly read from the memory. The boundary between the two zones is defined by the CONST_EO_RT control register. Similarly, for the fetch state, the boundary between the two zones is defined by the TSTATE_EO_RT control register.

5.6 Constant Waterfalling

In order to have a reasonable performance in the case of constant store indexing using the address register, we are going to have the possibility of using the physical memory port for read only. This way we can read 1 constant per clock and thus have a worst-case waterfall mode of 1 vertex per clock. There is a small synchronization issue related with this as we need for the SQ to make sure that the constants where actually written to memory (not only sent to the sequencer) before it can allow the first vector of pixels or vertices of the state to go thru the ALUs. To do so, the sequencer keeps 8 bits (one per render state) and sets the bits whenever the last render state is written to memory and clears the bit whenever a state is freed.

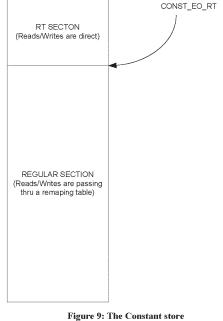


Exhibit 2035.docR400_Sequencer.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044 0257729

ATI Ex. 2109 IPR2023-00922 Page 19 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 201514		20 of 51

6. Looping and Branches

Loops and branches are planned to be supported and will have to be dealt with at the sequencer level. We plan on supporting constant loops and branches using a control program.

6.1 The controlling state.

The R400 controling state consists of:

Boolean[256:0] Loop_count[7:0][31:0] Loop_Start[7:0][31:0] Loop_Step[7:0][31:0]

That is 256 Booleans and 32 loops.

We have a stack of 4 elements for nested calls of subroutines and 4 loop counters to allow for nested loops.

This state is available on a per shader program basis.

6.2 The Control Flow Program

We'd like to be able to code up a program of the form:

1:	Loop	
2:	Exec	TexFetch
3:		TexFetch
4:		ALU
5:		ALU
6:		TexFetch
7:	End Lo	ор
8:	ALU Ex	port

But realize that 3: may be dependent on 2: and 4: is almost certainly dependent on 2: and 3:. Without clausing, these dependencies need to be expressed in the Control Flow instructions. Additionally, without separate 'texture clauses' and 'ALU clauses' we need to know which instructions to dispatch to the Texture Unit and which to the ALU unit. This information will be encapsulated in the flow control instructions.

Each control flow instruction will contain 2 bits of information for each (non-control flow) instruction: a) ALU or Texture

b) Serialize Execution

(b) would force the thread to stop execution at this point (before the instruction is executed) and wait until all textures have been fetched. Given the allocation of reserved bits, this would mean that the count of an 'Exec' instruction would be limited to about 8 (non-control-flow) instructions. If more than this were needed, a second Exec (with the same conditions) would be issued.

Another function that relies upon 'clauses' is allocation and order of execution. We need to assure that pixels and vertices are exported in the correct order (even if not all execution is ordered) and that space in the output buffers are allocated in order. Additionally data can't be exported until space is allocated. A new control flow instruction:

Alloc <buffer select -- position, parameter, pixel or vertex memory. And the size required>.

would be created to mark where such allocation needs to be done. To assure allocation is done in order, the actual allocation for a given thread can not be performed unless the equivalent allocation for all previous threads is already completed. The implementation would also assure that execution of instruction(s) following the serialization due to the Alloc will occur in order -- at least until the next serialization or change from ALU to Texture. In most cases this will allow the exports to occur without any further synchronization. Only 'final' allocations or position allocations are

Exhibit 2035.docR400_Sequencer.doc 73569 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257730

ATI Ex. 2109 IPR2023-00922 Page 20 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	The Association of the Associati
<u></u>	24 September, 2001	4 September, 201514	GEN-CXXXXX-REVA	21 of 51	Contraction of the

guaranteed to be ordered. Because strict ordering is required for pixels, parameters and positions, this implies only a single alloc for these structures. Vertex exports to memory do not require ordering during allocation and so multiple 'allocs' may be done.

6.2.1 Control flow instructions table

Here is the revised control flow instruction set.

Note that whenever a field is marked as RESERVED, it is assumed that all the bits of the field are cleared (0).

NOP							
47 44	43	42 0	20000				
0000	Addressing	RESERVED	10000				

This is a regular NOP.

	Execute										
47 44	43	40 34	33 16	1512	11 0						
0001	Addressing	RESERVED	Instructions type + serialize (9 instructions)	Count	Exec Address						

	Execute_End										
47 44	43	40 34	33 16	1512	11 0						
0010	Addressing	RESERVED	Instructions type + serialize (9 instructions)	Count	Exec Address						
			instructions)								

Execute up to 9 instructions at the specified address in the instruction memory. The Instruction type field tells the sequencer the type of the instruction (LSB) (1 = Texture, 0 = ALU and whether to serialize or not the execution (MSB) (1 = Serialize, 0 = Non-Serialized). If Execute_End this is the last execution block of the shader program.

Conditional_Execute										
4744 43 42 4134 3316 1512 1										
0011	Addressing	Condition	Boolean address	Instructions type + serialize (9 instructions)	Count	Exec Address				
	Conditional_Execute_End									

47 44	43	42	41 34	3316	15 12	11 0
0100	Addressing	Condition	Boolean	Instructions type + serialize (9	Count	Exec Address
			auuress	instructions)		

If the specified Boolean (8 bits can address 256 Booleans) meets the specified condition then execute the specified instructions (up to 9 instructions). If the condition is not met, we go on to the next control flow instruction. If Conditional_Execute_End and the condition is met, this is the last execution block of the shader program.

	Conditional_Execute_Predicates								
47 44	43	42	41 36	35 34	3316	1512	11 0		
0101	Addressing	Condition	RESERVED	Predicate	Instructions	Count	Exec Address		
				vector	type + serialize				
					(9 instructions)				

	Conditional_Execute_Predicates_End									
47 44	43	42	41 36	35 34	3316	1512	11 0			
0110	Addressing	Condition	RESERVED	Predicate	Instructions	Count	Exec Address			
				vector	type + serialize					
					(9 instructions)					

Check the AND/OR of all current predicate bits. If AND/OR matches the condition execute the specified number of instructions. We need to AND/OR this with the kill mask in order not to consider the pixels that aren't valid. If the

Exhibit 2035. doc R400_Sequences.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257731

ATI Ex. 2109 IPR2023-00922 Page 21 of 326

Γ		ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
		24 September, 2001	4 September, 201514		22 of 51
_	condition in n	of mot we go on to the	payt control flow instructio	n If Conditional Execute Predicator	End and the

condition is not met, we go on to the next control flow instruction. If Conditional_Execute_Predicates_End and the condition is met, this is the last execution block of the shader program.

		(Conditional_Exe	cute_Predic	ates_No_Stall		
47 44	43	42	41 36	35 34	3316	1512	11 0
1101	Addressing	Condition	RESERVED	Predicate vector	Instructions type + serialize (9 instructions)	Count	Exec Address
r					- No Ctall Fad		

		Co	nditional_Execut	e_Predicate	s_No_Stall_End		
47 44	43	42	41 36	35 34	3316	1512	11 0
1110	Addressing	Condition	RESERVED	Predicate vector	Instructions type + serialize (9 instructions)	Count	Exec Address

Same as Conditionnal_Execute_Predicates but the SQ is not going to wait for the predicate vector to be updated. You can only set this in the compiler if you know that the predicate set is only a refinement of the current one (like a nested if) because the optimization would still work.

		Loop_Star	t		
47 44	43	42 21	20 16	15123	1 <u>2</u> 4 0
0111	Addressing	RESERVED	loop ID	RESERVED	Jump address

Loop Start. Compares the loop iterator with the end value. If loop condition not met jump to the address. Forward jump only. Also computes the index value. The loop id must match between the start to end, and also indicates which control flow constants should be used with the loop.

			Loop_End				
47 44	43	42 24	23 21	20 16	15 12 13	<u> 11-12</u> 0	
1000	Addressing	RESERVED	Predicate break	loop ID	RESERVED	start address	

Loop end. Increments the counter by one, compares the loop count with the end value. If loop condition met, continue, else, jump BACK to the start of the loop. If predicate break != 0, then compares predicate vector n (specified by predicate break number). If all bits cleared then break the loop.

The way this is described does not prevent nested loops, and the inclusion of the loop id make this easy to do.

			Conditionr	nal_Call		
47 44	43	42	41 34	33 1314	1213	<u>11-12</u> 0
1001	Addressing	Condition	Boolean address	RESERVED	Force Call	Jump address

If the condition is met, jumps to the specified address and pushes the control flow program counter on the stack. If force call is set the condition is ignored and the call is made always.

		Return	
47 44	43	42 0	
1010	Addressing	RESERVED	

Pops the topmost address from the stack and jumps to that address. If nothing is on the stack, the program will just continue to the next instruction.

				Conditionr	nal_Jump			
47 44	43	42	41 34	33	32 13 <u>14</u>	<u> 1213</u>	<u> 11-12 0</u>	
1011	Addressing	Condition	Boolean	FW only	RESERVED	Force Jump	Jump address	
			address					- 8

If force jump is set the condition is ignored and the jump is made always. If FW only is set then only forward jumps are allowed.

Exhibit 2035.docR400_Sequencer.doc 73569 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257732

ATI Ex. 2109 IPR2023-00922 Page 22 of 326

	AÎ		INATE DATE otember, 2001	4 Septemt	DATE		T-REV. NUM. XXX-REVA	PAGE 23 of 51
					Allocate			
4	17 44	43	424	41	4	03	20	
	1100	Debug	Buffer S	elect	RES	SERVED	Size	

Buffer Select takes a value of the following:

01 – position export (ordered export)

10 - parameter cache or pixel export (ordered export)

11 - pass thru (out of order exports).

Size field is only used to reserve space in the export buffer for pass thru exports. Valid values are 1 (1 line) thru 9 (9 lines). It should be determined by the compiler/assembler by taking max index used +1.

If debug is set this is a debug alloc (ignore if debug DB_ON register is set to off).

6.3 Implementation

The envisioned implementation has a buffer that maintains the state of each thread. A thread lives in a given location in the buffer during its entire life, but the buffer has FIFO qualities in that threads leave in the order that they enter. Actually two buffers are maintained -- one for Vertices and one for Pixels. The intended implementation would allow for:

16 entries for vertices 48 entries for pixels.

From each buffer, arbitration logic attempts to select 1 thread for the texture unit and 1 (interleaved) thread for the ALU unit. Once a thread is selected it is read out of the buffer, marked as invalid, and submitted to appropriate execution unit. It is returned to the buffer (at the same place) with its status updated once all possible sequential instructions have been executed. A switch from ALU to TEX or visa-versa or a Serialize_Execution modifier forces the thread to be returned to the buffer.

Each entry in the buffer will be stored across two physical pieces of memory - most bits will be stored in a 1 read port device. Only bits needed for thread arbitration will be stored in a highly multi-ported structure. The bits kept in the 1 read port device will be termed 'state'. The bits kept in the multi-read ported device will be termed 'status'.

'State Bits' needed include:

- 1. Control Flow Instruction Pointer (13 bits),
- 2. Execution Count Marker 4 bits),
- 3. Loop Iterators (4x9 bits),
- 4. Loop Counters (4x9 bits),
- 4.5. Call return pointers (4x12 4x13 bits),
- 5.6. Predicate Bits (64 bits),
- 6.7. Export ID (1 bit),
- 7.8. Parameter Cache base Ptr (7 bits),
- 8.9. GPR Base Ptr (8 bits),
- 9.10. Context Ptr (3 bits).
- 10.11. LOD corrections (6x16 bits)
- 11.12. Valid bits (64 bits)
- 42.13. RT (1 bit) Signifies that this thread is a Real Time thread. This bit must be sent to the Constant store state machine when reading it.

Absent from this list are 'Index' pointers. These are costly enough that I'm presuming that they are instead stored in the GPRs. The first seven fields above (Control Flow Ptr, Execution Count, Loop Counts, call return ptrs, Predicate bits, PC base ptr and export ID) are updated every time the thread is returned to the buffer based on how much progress has been mode on thread execution. GPR Base Ptr, Context Ptr and LOD corrections are unchanged throughout execution of the thread.

Exhibit 2035.docR400_Sequences.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

Formatted: Bullets and Numbering

AMD1044_0257733

ATI Ex. 2109 IPR2023-00922 Page 23 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 201514		24 of 51

'Status Bits' needed include:

- Valid Thread
- Texture/ALU engine needed
- Texture Reads are outstanding
- Waiting on Texture Read to Complete
- Allocation Wait (2 bits)
- 00 No allocation needed
- 01 Position export allocation needed (ordered export)
- 10 Parameter or pixel export needed (ordered export)
- 11 pass thru (out of order export)
- Allocation Size (4 bits)
- Position Allocated Mem/Color Allocated
- First thread of a new context
- Event thread (NULL thread that needs to trickle down the pipe)
- Last (1 bit)
- Pulse SX (1 bit)

All of the above fields from all of the entries go into the arbitration circuitry. The arbitration circuitry will select a winner for both the Texture Engine and for the ALU engine. There are actually two sets of arbitration -- one for pixels and one for vertices. A final selection is then done between the two. But the rest of this implementation summary only considers the 'first' level selection which is similar for both pixels and vertices.

Texture arbitration requires no allocation or ordering so it is purely based on selecting the 'oldest' thread that requires the Texture Engine.

ALU arbitration is a little more complicated. First, only threads where either of Texture_Reads_outstanding or Waiting_on_Texture_Read_to_Complete are '0' are considered. Then if Allocation_Wait is active, these threads are further filtered based on whether space is available. If the allocation is position allocation, then the thread is only considered if all 'older' threads have already done their position allocation (position allocated bits set). If the allocation is parameter or pixel allocation, then the thread is only considered if it is the oldest thread. Also a thread is not considered if it is a parameter or pixel or position allocation, has its First_thread_of_a_new_context bit set and would cause ALU interleaving with another thread performing the same parameter or pixel or position allocation. Finally the 'oldest' of the threads that pass through the above filters is selected. If the thread needed to allocate, then at this time the allocation is done, based on Allocation_Size. If a thread has its "last" bit set, then it is also removed from the buffer, never to return.

If I now redefine 'clauses' to mean 'how many times the thread is removed from the thread buffer for the purpose of exection by either the ALU or Texture engine', then the minimum number of clauses needed is 2 -- one to perform the allocation for exports (execution automatically halts after an 'Alloc' instruction) (but doesn't performs the actual allocation) and one for the actual ALU/export instructions. As the 'Alloc' instruction could be part of a texture clause (presumably the final instruction in such a clause), a thread could still execute in this minimal number of 2 clauses, even if it involved texture fetching.

The Texture_Reads_Outstanding bit must be updated by the sequencer, based on keeping track of how many Texture Clauses have been executed by a given thread that have not yet had there data returned. Any number above 0 results in this bit being set. We could consider forcing synchronization such that two texture clauses for a given thread may not be outstanding at any time (that would be my preference for simplicity reasons and because it would require only very little change in the texture pipe interface). This would allow the sequencer to set the bit on execution of the texture clause, and allow the texture unit to return a pointer to the thread buffer on completion that clears the bit.

6.4 Data dependant predicate instructions

Data dependant conditionals will be supported in the R400. The only way we plan to support those is by supporting three vector/scalar predicate operations of the form:

Exhibit 2035.doc.F400_Sequencer.doc 73569 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044 0257734

ATI Ex. 2109 IPR2023-00922 Page 24 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 201514	GEN-CXXXXX-REVA	25 of 51

PRED_SETE_# - similar to SETE except that the result is 'exported' to the sequencer. PRED_SETNE_# - similar to SETNE except that the result is 'exported' to the sequencer. PRED_SETGT_# - similar to SETGT except that the result is 'exported' to the sequencer PRED_SETGTE_# - similar to SETGTE except that the result is 'exported' to the sequencer.

For the scalar operations only we will also support the two following instructions: PRED_SETE0_# – SETE0 PRED_SETE1_# – SETE1

The export is a single bit - 1 or 0 that is sent using the same data path as the MOVA instruction. The sequencer will maintain 4 sets of 64 bit predicate vectors (in fact 8 sets because we interleave two programs but only 4 will be exposed) and use it to control the write masking. This predicate is not maintained across clause boundaries. The # sign is used to specify which predicate set you want to use 0 thru 3.

Then we have two conditional execute bits. The first bit is a conditional execute "on" bit and the second bit tells us if we execute on 1 or 0. For example, the instruction:

P0_ADD_# R0,R1,R2

Is only going to write the result of the ADD into those GPRs whose predicate bit is 0. Alternatively, P1_ADD_# would only write the results to the GPRs whose predicate bit is set. The use of the P0 or P1 without precharging the sequencer with a PRED instruction is undefined.

{Issue: do we have to have a NOP between PRED and the first instruction that uses a predicate?}

6.5 HW Detection of PV,PS

Because of the control program, the compiler cannot detect statically dependant instructions. In the case of nonmasked writes and subsequent reads the sequencer will insert uses of PV,PS as needed. This will be done by comparing the read address and the write address of consecutive instructions. For masked writes, the sequencer will insert NOPs wherever there is a dependant read/write.

The sequencer will also have to insert NOPs between PRED_SET and MOVA instructions and their uses.

6.6 Register file indexing

Because we can have loops in fetch clause, we need to be able to index into the register file in order to retrieve the data created in a fetch clause loop and use it into an ALU clause. The instruction will include the base address for register indexing and the instruction will contain these controls:

Bit7	Bit 6	
0	0	'absolute register'
0	1	'relative register'
1	0	'previous vector'
1	1	'previous scalar'

In the case of an absolute register we just take the address as is. In the case of a relative register read we take the base address and we add to it the loop_index and this becomes our new address that we give to the shader pipe.

The sequencer is going to keep a loop index computed as such:

Index = Loop_iterator*Loop_step + Loop_start.

We loop until loop_iterator = loop_count. Loop_step is a signed value [-128...127]. The computed index value is a 10 bit counter that is also signed. Its real range is [-256,256]. The tenth bit is only there so that we can provide an out of range value to the "indexing logic" so that it knows when the provided index is out of range and thus can make the necessary arrangements.

Exhibit 2035.docR400_Sequencer.doc 7369 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257735

ATI Ex. 2109 IPR2023-00922 Page 25 of 326

				r				
	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE				
	24 September, 2001	4 September, 201514		26 of 51				
6.7 De	bugging the Shade	ers						
In order to	In order to be able to debug the pixel/vertex shaders efficiently, we provide 2 methods.							
671 A	lethod 1: Debugging	rogistore						
	ins are to expose 2 debugging	-	ietore:					
1. address	register where the first error of the number of errors		191619.					
	ncer will detect the following g	groups of errors:						
	ndexing overflow							
- register ir	dexing overflow							
Compiler r - jump err	ecognizable errors: ors							
rel - call stac	ative jump address > size of t k	he control flow program						
ca	l with stack full urn with stack empty							
	or will always cause the progr ther clauses to be executed.	ram to preak. In this case,	a break means that a clause will halt	execution, but				
	e other errors, program can c OB_BREAK register is set.	continue to run, potentially	to worst-case limits. The program wi	ll only break if				
the value v			an overflow error, the hardware is spe e error tokens, by reserving and initia					
{ISSUE : Ir	terrupt to the driver or not?}							
6.7.2 N	lethod 2: Exporting th	ne values in the GF	PRs					
			It register and an address register for	this mode.				
Under the	normal mode execution follow	vs the normal course.						
to the SX b		vill been turned off (chang	exporting n debug vectors and that all ed into NOPs) by the sequencer (eve					
7. Pixe	I Kill Mask							
	64 bits is kept by the seque nd allow the shader pipe to ki		vertices. Its purpose is to optimize the g instructions:	e texture fetch				
	SK_SETE							
	\SK_SETNE \SK_SETGT							
MA	SK_SETGTE							
8. <u>Mul</u> t	ipass vertex shade	<u>ers (HOS)</u>						
Multipass \	rertex shaders are able to exp	port from the 6 last clauses	s but to memory ONLY.					
Exhibit 2035	docR400_Sequencer.doc 73569 Bytes***	© ATI Confidential. Ref	erence Copyright Notice on Cover F	age © ***				
				2				

ATI Ex. 2109 IPR2023-00922 Page 26 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 201544	GEN-CXXXXX-REVA	27 of 51

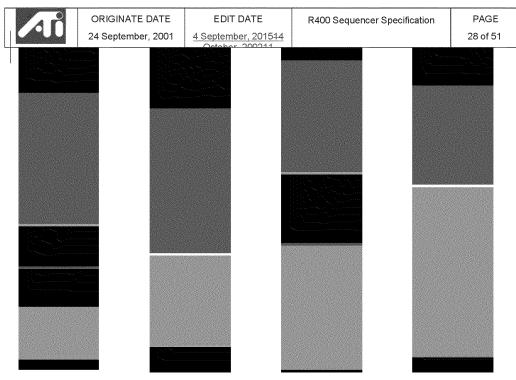
9. Register file allocation

The register file allocation for vertices and pixels can either be static or dynamic. In both cases, the register file in managed using two round robins (one for pixels and one for vertices). In the dynamic case the boundary between pixels and vertices is allowed to move, in the static case it is fixed to 128-VERTEX_REG_SIZE for vertices and PIXEL_REG_SIZE for pixels.

Exhibit 2035.docR400_Sequences.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257737

ATI Ex. 2109 IPR2023-00922 Page 27 of 326



Above is an example of how the algorithm works. Vertices come in from top to bottom; pixels come in from bottom to top. Vertices are in orange and pixels in green. The blue line is the tail of the vertices and the green line is the tail of the pixels. Thus anything between the two lines is shared. When pixels meets vertices the line turns white and the boundary is static until both vertices and pixels share the same "unallocated bubble". Then the boundary is allowed to move again. The numbering of the GPRs starts from the bottom of the picture at index 0 and goes up to the top at index 127.

10. Fetch Arbitration

The fetch arbitration logic chooses one of the n potentially pending fetch clauses to be executed. The choice is made by looking at the Vs and Ps reservation stations and picking the first one ready to execute. Once chosen, the clause state machine will send one 2x2 fetch per clock (or 4 fetches in one clock every 4 clocks) until all the fetch instructions of the clause are sent. This means that there cannot be any dependencies between two fetches of the same clause.

The arbitrator will not wait for the fetches to return prior to selecting another clause for execution. The fetch pipe will be able to handle up to X(?) in flight fetches and thus there can be a fair number of active clauses waiting for their fetch return data.

11. ALU Arbitration

ALU arbitration proceeds in almost the same way than fetch arbitration. The ALU arbitration logic chooses one of the n potentially pending ALU clauses to be executed. The choice is made by looking at the Vs and Ps reservation stations and picking the first one ready to execute. There are two ALU arbitrers, one for the even clocks and one for the odd clocks. For example, here is the sequencing of two interleaved ALU clauses (E and O stands for Even and Odd sets of 4 clocks):

Einst0 Oinst0 Einst1 Oinst1 Einst2 Oinst2 Einst0 Oinst3 Einst1 Oinst4 Einst2 Oinst0...

Exhibit 2035.docF4400_Sequencer.doc 73669 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257738

ATI Ex. 2109 IPR2023-00922 Page 28 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 201514	GEN-CXXXXX-REVA	29 of 51

Proceeding this way hides the latency of 8 clocks of the ALUs. Also note that the interleaving also occurs across clause boundaries.

12. Handling Stalls

When the output file is full, the sequencer prevents the ALU arbitration logic from selecting the last clause (this way nothing can exit the shader pipe until there is place in the output file. If the packet is a vertex packet and the position buffer is full (POS_FULL) then the sequencer also prevents a thread from entering an exporting clause. The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU arbitrer will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

13. Content of the reservation station FIFOs

The reservation FIFOs contain the state of the vector of pixels and vertices. We have two sets of those: one for pixels, and one for vertices. They contain 3 bits of Render State 7 bits for the base address of the GPRs, some bits for LOD correction and coverage mask information in order to fetch fetch for only valid pixels, the quad address.

14. The Output File

The output file is where pixels are put before they go to the RBs. The write BW to this store is 256 bits/clock. Just before this output file are staging registers with write BW 512 bits/clock and read BW 256 bits/clock. The staging registers are 4x128 (and there are 16 of those on the whole chip).

15. IJ Format

The IJ information sent by the PA is of this format on a per quad basis:

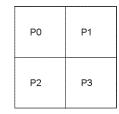
We have a vector of IJ's (one IJ per pixel at the centroid of the fragment or at the center of the pixel depending on the mode bit). All pixel's parameters are always interpolated at full 20x24 mantissa precision.

$$P0 = A + I(0) * (B - A) + J(0) * (C - A)$$

$$P1 = A + I(1) * (B - A) + J(1) * (C - A)$$

$$P2 = A + I(2) * (B - A) + J(2) * (C - A)$$

$$P3 = A + I(3) * (B - A) + J(3) * (C - A)$$



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

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

Total number of bits : 20*8 + 4*8 + 4*2 = 200.

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 +/- 1024.

15.1 Interpolation of constant attributes

Because of the floating point imprecision, we need to take special provisions if all the interpolated terms are the same or if two of the terms are the same.

Exhibit 2035.docR400_Sequencer.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257739

ATI Ex. 2109 IPR2023-00922 Page 29 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 201514		30 of 51

16. Staging Registers

In order for the reuse of the vertices to be 14, the sequencer will have to re-order the data sent IN ORDER by the VGT for it to be aligned with the parameter cache memory arrangement. Given the following group of vertices sent by the VGT:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 || 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 || 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 || 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

The sequencer will re-arrange them in this fashion:

0 1 2 3 16 17 18 19 32 33 34 35 48 49 50 51 || 4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 || 8 9 10 11 24 25 26 27 40 41 42 43 56 57 58 59 || 12 13 14 15 28 29 30 31 44 45 46 47 60 61 62 63

The || markers show the SP divisions. In the event a shader pipe is broken, the SQ is responsible to insert padding to account for the missing pipe. For example, if SP1 is broken, vertices 4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 will not be sent by the VGT to the SQ AND the SQ is responsible to "jump" over these vertices in order for no valid vertices to be sent to an invalid SP.

The most straightforward, *non-compressed* interface method would be to convert, in the VGT, the data to 32-bit floating point prior to transmission to the VSISRs. In this scenario, the data would be transmitted to (and stored in) the VSISRs in full 32-bit floating point. This method requires three 24-bit fixed-to-float converters in the VGT. Unfortunately, it also requires and additional 3,072 bits of storage across the VSISRs. This interface is illustrated in Figure 11Figure-11Figure-11. The area of the fixed-to-float converters and the VSISRs for this method is roughly estimated as 0.759sqmm using the R300 process. The gate count estimate is shown in Figure 10Figure-10.

Basis for 8-deep Latch Memory (fror	n R300)			
8x24-bit	11631 μ^2		$60.57813\mu^2\text{per}$ b	
Area of 96x8-deep Latch Memory	46524	μ^2		
Area of 24-bit Fix-to-float Converter	4712	μ^2 per conve	erter	
Method 1	Block	Quantity	Area	
	F2F	3	14136	
	8x96 Latch	16_	744384	
			758520 µ ²	

Figure 10: Area Estimate for VGT to Shader Interface

Exhibit 2035.docR400_Sequencer.doc 73569 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257740

ATI Ex. 2109 IPR2023-00922 Page 30 of 326

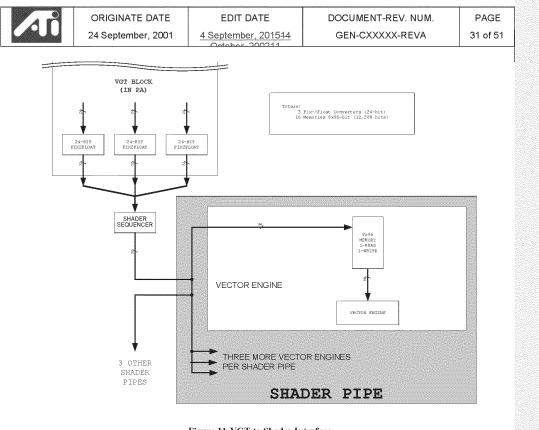


Figure 11:VGT to Shader Interface

17. The parameter cache

The parameter cache is where the vertex shaders export their data. It consists of 16 128x128 memories (1R/1W). The reuse engine will make it so that all vertexes of a given primitive will hit different memories. The allocation method for these memories is a simple round robin. The parameter cache pointers are mapped in the following way: 4MSBs are the memory number and the 7 LSBs are the address within this memory.

MEMORY NUMBER 4 bits	ADDRESS 7 bits

The PA generates the parameter cache addresses as the positions come from the SQ. All it needs to do is keep a Current_Location pointer (7 bits only) and as the positions comes increment the memory number. When the memory number field wraps around, the PA increments the Current_Location by VS_EXPORT_COUNT (a snooped register from the SQ). As an example, say the memories are all empty to begin with and the vertex shader is exporting 8 parameters per vertex (VS_EXPORT_COUNT = 8). The first position received is going to have the PC address 00000000000 the second one 00010000000, third one 0010000000 and so on up to 11110000000. Then the next position received (the 17th) is going to have the address 0000001000, the 18th 00010001000, the 19th 0010001000 and so on. The Current_location is NEVER reset BUT on chip resets. The only thing to be careful about is that if the SX doesn't send you a full group of positions (<64) then you need to fill the address space so that the next group starts correctly aligned (for example if you receive only 33 positions then you need to add 2*VS_EXPORT_COUNT to Current_Location and reset the memory count to 0 before the next vector begins).

Exhibit 2035.docR400_Sequences.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257741

ATI Ex. 2109 IPR2023-00922 Page 31 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 201514		32 of 51

17.1 Export restrictions

17.1.1 Pixel exports:

Pixels can export 1,2,3 or 4 color buffers to the SX(+z). The exports will be done in order. The PRED_OPTIMIZE function has to be turned of if the exports are done using interleaved predicated instructions. The exports will always be ordered to the SX.

17.1.2 Vertex exports:

Position or parameter caches can be exported in any order in the shader program. It is always better to export position as soon as possible. Position has to be exported in a single export block (no texture instructions can be placed between the exports). Parameter cache exports can be done in any order with texture instructions interleaved. The PRED_OPTIMIZE function has to be turned of if the exports are done using interleaved predicated instructions to the Parameter cache (see Arbitration restrictions for details). The exports will always be allocated in order to the SX.

17.1.3 Pass thru exports:

Pass thru exports have to be done in groups of the form:

Alloc 4 (8 or 12) Execute ALU(ADDR) ALU(DATA) ALU(DATA) ALU(DATA)...

They cannot have texture instructions interleaved in the export block. These exports are not guaranteed to be ordered.

Also, when doing a pass thru export, Position MUST be exported AFTER all pass thru exports. This position export is used to synchronize the chip when doing a transition from pass thru shader to regular shader and vice versa.

17.2 Arbitration restrictions

Here are the Sequencer arbitration restrictions:

- 1) Cannot execute a serialized thread if the corresponding texture pending bit is set
- 2) Cannot allocate position if any older thread has not allocated position
- 3) If last thread is marked as not valid AND marked as last and we are about to execute the second to oldest thread also marked last then:
 - a. Both threads must be from the same context (cannot allow a first thread)
 - b. Must turn off the predicate optimization for the second thread
- 4) Cannot execute a texture clause if texture reads are pending
- 5) Cannot execute last if texture pending (even if not serial)

18. Export Types

The export type (or the location where the data should be put) is specified using the destination address field in the ALU instruction. Here is a list of all possible export modes:

18.1 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
- 38:47 Empty
- 48:52 5 debug export (interpret as normal memory export)
- 60 export addressing mode
- 61 Empty
- 62 position

Exhibit 2035.docR400_Sequencer.doc 73669 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257742

ATI Ex. 2109 IPR2023-00922 Page 32 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
	24 September, 2001	4 September, 201514	GEN-CXXXXX-REVA	33 of 51	
63 - sprite size export that goes with position export					

sign means VtxKill.)

(X= point size, Y= edge flag is bit 0, Z= VtxKill is bitwise OR of bits 30:0. Any bit other than

18.2 Pixel Shading

2

- Color for buffer 0 (primary) 0
- Color for buffer 1
- Color for buffer 2 - Color for buffer 3
- 3 4:15 - Empty
- Buffer 0 Color/Fog (primary) 16
- 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.
- 38:47 Empty
- 48:52 - 5 debug exports (interpret as normal memory export)
- 60 - export addressing mode
- 61 - Z for primary buffer (Z exported to 'alpha' component)
- 62:63 Empty

19. Special Interpolation modes

19.1 Real time commands

We are unable to use the parameter memory since there is no way for a command stream to write into it. Instead we need to add three 16x128 memories (one for each of three vertices x 16 interpolants). These will be mapped onto the register bus and written by type 0 packets, and output to the the parameter busses (the sequencer and/or PA need to be able to address the reatime parameter memory as well as the regular parameter store. For higher performance we should be able able to view them as two banks of 16 and do double buffering allowing one to be loaded, while the other is rasterized with. Most overlay shaders will need 2 or 4 scalar coordinates, one option might be to restrict the memory to 16x64 or 32x64 allowing only two interpolated scalars per cycle, the only problem I see with this is, if we view support for 16 vector-4 interpolants important (true only if we map Microsoft's high priority stream to the realtime stream), then the PA/sequencer need to support a realtime-specific mode where we need to address 32 vectors of parameters instead of 16. This mode is triggered by the primitive type: REAL TIME. The actual memories are in the in the SX blocks. The parameter data memories are hooked on the RBBM bus and are loaded by the CP using register mapped memory.

19.2 Sprites/ XY screen coordinates/ FB information

XY screen coordinates may be needed in the shader program. This functionality is controlled by the param_gen_l0 register (in SQ) in conjunction with the SND_XY register (in SC) and the param_gen_pos. Also it is possible to send the faceness information (for OGL front/back special operations) to the shader using the same control register. Here is a list of all the modes and how they interact together:

The Data is going to be written in the register specified by the param_gen_pos register.

Param_Gen_I0 disable, snd_xy disable = No modification Param_Gen_I0 disable, snd_xy enable = No modification Param_Gen_I0 enable, snd_xy disable = Sign(faceness)garbage,(Sign Point)garbage,Sign(Line)s, t Param_Gen_I0 enable, snd_xy enable = Sign(faceness)screenX,(Sign Point)screenY,Sign(Line)s, t

In other words,

The generated vector is (X in RED, Y in GREEN, S in BLUE and T in ALPHA): X.Y.S.T

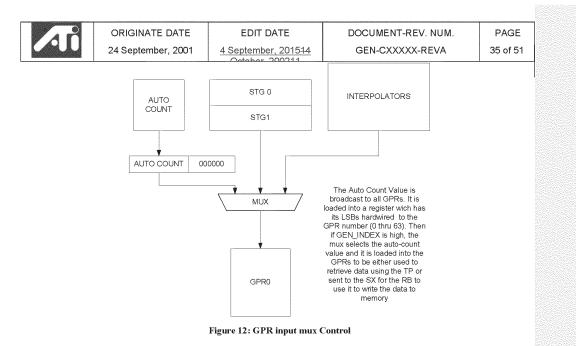
Exhibit 2035. doc R400_Sequences.doc 73669 Bytes*** [©] ATI Confidential. Reference Copyright Notice on Cover Page [©] ***

AMD1044 0257743

ATI Ex. 2109 IPR2023-00922 Page 33 of 326

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE]
	24 September, 2001	4 September, 201514		34 of 51	
 (as thei SignX = SignY = SignS = SignT = If !Point I would stipple if(Y<0) } else if if (Y<0) } else if if (Y<0) } else if } else if } else if if (Y<0) <	values are always support is sign bits will now be use = BackFacing = Point Primitive = Line Primitive = currently unused as a f t & !Line, then it is a Poly assume that one impler and AA for the driver wo { R = 0.0 (Point) (S < 0) { R = 1.0 (Line) R = 2.0 (Poly) generated cou e are dealing with multipount to write the 1 st pass ways generated in the as (pixel or vertex). This is p two counters, one for GPRs the counter is incl orresponding counter is ecific values making the rtices and the 4 LSBs (nters add to be positive and any sed for flags). alag. Alag.	generic texture lookup (using 3D map generic texture lookup (using 3D map n use the count to retrieve the data or o the shader in a slightly different way the GEN_INDEX_PIX/VTX register. T res. Every time a full vector of vertice (ST_PIX_COUNT or RST_VTX_COUI by one count broadcast to the GPRs nents in the vector. Since the count mud to the corresponding shader unit th this case the phase must be appende	BS function BS function (BS) for poly (BS) f	
Co	unter (19 bits)	Phase (2 bits)	Hardwired (4 bits)		
2) Maintai	n a 21 bits counter that e	counts sub-vectors of 16. I	n this case only the counter is sent to	the Sps:	
Counte	r (21 bits)	Hard	vired (4 bits)		
1031 1/0	tex shaders				
In the case of	vertex shaders, if GEN_	INDEX_VTX is set, the da GPRs in all multipass ver	ata will be put into the x field of the th rtex shader modes).	hird register (it	
19.3.2 Pix	el shaders				
In the case of register.	pixel shaders, if GEN_II	NDEX_PIX is set, the data	a will be put in the x field of the para	m_gen_pos+1	
Exhibit 2035 docR44	20_Sequencer.doc 73569 Bytes***	© ATI Confidential. Ref	erence Copyright Notice on Cover F	⊃age © ***	

ATI Ex. 2109 IPR2023-00922 Page 34 of 326



20. State management

Every clock, the sequencer will report to the CP the oldest states still in the pipe. These are the states of the programs as they enter the last ALU clause.

20.1 Parameter cache synchronization

In order for the sequencer not to begin a group of pixels before the associated group of vertices has finished, the sequencer will keep a 6 bit count per state (for a total of 8 counters). These counters are initialized to 0 and every time a vertex shader exports its data TO THE PARAMETER CACHE, the corresponding pointer is incremented. When the SC sends a new vector of pixels with the SC_SQ_new_vector bit asserted, the sequencer will first check if the count is greater than 0 before accepting the transmission (it will in fact accept the transmission but then lower its ready to receive). Then the sequencer waits for the count to go to one and decrements it. The sequencer can then issue the group of pixels to the interpolators. Every time the state changes, the new state counter is initialized to 0.

21. XY Address imports

The SC will be able to send the XY addresses to the GPRs. It does so by interleaving the writes of the IJs (to the IJ buffer) with XY writes (to the XY buffer). Then when writing the data to the GPRs, the sequencer is going to interpolate the IJ data or pass the XY data thru a Fix→float converter and expander and write the converted values to the GPRs. The Xys are currently SCREEN SPACE COORDINATES. The values in the XY buffers will wrap. See section 19.2 for details on how to control the interpolation in this mode.

21.1 Vertex indexes imports

In order to import vertex indexes, we have 16 8x96 staging registers. These are loaded one line at a time by the VGT block (96 bits). They are loaded in floating point format and can be transferred in 4 or 8 clocks to the GPRs.

22. Registers

Please see the auto-generated web pages for register definitions.

Exhibit 2035.docR400_Sequencer.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257745

ATI Ex. 2109 IPR2023-00922 Page 35 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 201514		36 of 51

23. Interfaces

23.1 External Interfaces

Whenever an x is used, it means that the bus is broadcast to all units of the same name. For example, if a bus is named SQ \rightarrow SPx it means that SQ is going to broadcast the same information to all SP instances.

23.2 SC to SP Interfaces

23.2.1 SC_SP#

There is one of these interfaces at front of each of the SP (buffer to stage pixel interpolators). This interface transmits the I,J data for pixel interpolation. For the entire system, two quads per clock are transferred to the 4 SPs, so each of these 4 interfaces transmits one half of a quad per clock. The interface below describes a half of a quad worth of data.

The actual data which is transferred per quad is

Ref Pix I => S4.20 Floating Point I value *4

Ref Pix J => S4.20 Floating Point J value *4

This equates to a total of 200 bits which transferred over 2 clocks and therefor needs an interface 100 bits wide

Additionally, X,Y data (12-bit unsigned fixed) is conditionally sent across this data bus over the same wires in an additional clock. The X,Y data is sent on the lower 24 bits of the data bus with faceness in the msb. Transfers across these interfaces are synchronized with the SC_SQ IJ Control Bus transfers.

The data transfer across each of these busses is controlled by a IJ_BUF_INUSE_COUNT in the SC. Each time the SC has sent a pixel vector's worth of data to the SPs, he will increment the IJ_BUF_INUSE_COUNT count. Prior to sending the next pixel vectors data, he will check to make sure the count is less than MAX_BUFER_MINUS_2, if not the SC will stall until the SQ returns a pipelined pulse to decrement the count when he has scheduled a buffer free. Note: We could/may optimize for the case of only sending only IJ to use all the buffers to pre-load more. Currently it is planned for the SP to hold 2 double buffers of I,J data and two buffers of X,Y data, so if either X,Y or Centers and Centroids are on, then the SC can send two Buffers.

In at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will increment buffer write address pointers correctly all the time. (We may revisit this for both the SX,SP,SQ and add a EndOfVector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of packet and multiple new vector signals should cause a partial vector and that this would not really be significant performance hit.)

Bits	Description
100	IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit)
	Type 0 or 1, First clock I, second clk J
	Field ULC URC LLC LRC
	Bits [63:39] [38:26] [25:13] [12:0]
	Format SE4M20 SE4M20 SE4M20 SE4M20
	Type 2
	Field Face X Y
	Bits [24] [23:12] [11:0]
	Format Bit Unsigned Unsigned
	Format Dit Onsigned Onsigned
1	Valid
1	This bit will be set on the last transfer of data per quad.
2	0 -> Indicates centroids
	1 -> Indicates centers
	2 -> Indicates X.Y Data and faceness on data bus
	The SC shall look at state data to determine how many types to send for the
	100 1 1

Exhibit 2035.docR400_Sequencer.doc 73569 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257746

ATI Ex. 2109 IPR2023-00922 Page 36 of 326

A î	ORIGINATE DATE 24 September, 2001	EDIT DATE <u>4 September, 201514</u> Optober, 200211	DOCUMENT-REV. NUM. GEN-CXXXXX-REVA	PAGE 37 of 51
	i	nterpolation process.		

The # is included for clarity in the spec and will be replaced with a prefix of u#_ in the verilog module statement for the SC and the SP block will have neither because the instantiation will insert the prefix.

23.2.2 SC_SQ

This is the control information sent to the sequencer in order to synchronize and control the interpolation and/or loading data into the GPRs needed to execute a shader program on the sent pixels. This data will be sent over two clocks per transfer with 1 to 16 transfers. Therefore the bus (approx 108 bits) could be folded in half to approx 54 bits.

Name	Bits	Description
Name SC_SQ_data	Bits 46	Description Control Data sent to the SQ 1 clk transfers Event - valid data consist of event_id and state_id. Instruct SQ to post an event vector to send state id and event_id through request fifo and onto the reservation stations making sure state id and/or event_id gets back to the CP. Events only follow end of packets so no pixel vectors will be in progress. Empty Quad Mask – Transfer Control data consisting of pc_dealloc or new_vector. Receipt of this is to transfer pc_dealloc or new_vector without any valid quad data. New vector will always be posted to request fifo and pc_dealloc will be attached to any pixel vector outstanding or posted in request fifo if no valid quad outstanding. 2 clk transfers Quad Data Valid – Sending quad data with or without new_vector or pc_dealloc. New vector will be posted to request fifo with or without a pixel vector and
		New vector will be posted to request
SC_SQ_valid	1	The Quad mask set but the pixel corresponding pixel mask set to zero. SC sending valid data, 2 nd clk could be all zeroes

SC_SQ_data – first clock and second clock transfers are shown in the table below.

Name	BitField	Bits	Description
1 st Clock Transfer			
SC_SQ_event	0	1	This transfer is a 1 clock event vector Force quad_mask =
			new_vector=pc_dealloc=0
SC_SQ_event_id	[5:1]	4	This field identifies the event 0 => denotes an End Of State Event 1

Exhibit 2035.docR400_Sequences.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257747

ATI Ex. 2109 IPR2023-00922 Page 37 of 326

Г				E	DIT DATE	D 400 0	PAGE
		ORIGINATEL		E	DITUATE	R400 Sequencer Specification	PAGE
		24 September	, 2001		ember, 201514		38 of 51
Ч			L		=> TBD		1
	SC SQ state	id	[8:6]	3	State/constant pointer (6*3+3)		
	SC SQ pc de		[11:9]	3		ken for the Parameter Cache	
	SC SQ new		12	1		ait for Vertex shader done count > 0 a	nd after
	00_00					Pixel Vector the SQ will decrement th	
	SC SQ quad	mask	[16:13]	4	<u></u>	sk left to right SP0 => SP3	
	SC_SQ_end_	of_prim	17	1	End Of the prim	itive	
	SC_SQ_pix_n		[33:18]	16	Valid bits for all	pixels SP0=>SP3 (UL,UR,LL,LR)	
	SC_SQ_provo	ok_vtx	[35:34]	2	Provoking verte	x for flat shading	
	SC_SQ_lod_c	orrect_0	[44:36]	9	LOD correction for quad 0 (SP0) (9 bits per quad)		
	SC_SQ_lod_c	orrect_1	[53:45]	9	LOD correction for quad 1 (SP1) (9 bits per quad)		
	2nd Clock Tra	nsfer					
	SC_SQ_lod_c	orrect_2	[8:0]	9	LOD correction	for quad 2 (SP2) (9 bits per quad)	
	SC_SQ_lod_c	1000	[17:9]	9	LOD correction for quad 3 (SP3) (9 bits per quad)		
	SC_SQ_pc_pt		[28:18]	11	Parameter Cache pointer for vertex 0		
	SC_SQ_pc_pt		[39:29]	11	Parameter Cac	he pointer for vertex 1	
	SC_SQ_pc_pt		[50:40]	11		he pointer for vertex 2	
	SC_SQ_prim_	type	[53:51]	3		d Real time command need to load te	x cords from
					alternate buffer		
					000: Sprite (poi	nt)	
					001: Line		
					010: Tri_rect	Navita (naint)	
					100: Realtime S 101: Realtime L		
					101: Realtime I 110: Realtime T		
			I		TTO. Realtime I		

Name	Bits	Description
SQ_SC_free_buff	1	Pipelined bit that instructs SC to decrement count of buffers in use.
SQ_SC_dec_cntr_cnt	1	Pipelined bit that instructs SC to decrement count of new vector and/or event
		sent to prevent SC from overflowing SQ interpolator/Reservation request fifo.

The scan converter will submit a partial vector whenever:

1.) He gets a primitive marked with an end of packet signal.

2.) A current pixel vector is being assembled with at least one or more valid quads and the vector has been marked for deallocate when a primitive marked new_vector arrives. The Scan Converter will submit a partial vector (up to 16quads with zero pixel mask to fill out the vector) prior to submitting the new_vector marker\primitive.

(This will prevent a hang which can be demonstrated when all primitives in a packet three vectors are culled except for a one quad primitive that gets marked pc_dealloc (vertices maximum size). In this case two new_vectors are submitted and processed, but then one valid quad with the pc_dealloc creates a vector and then the new would wait for another vertex vector to be processed, but the one being waited for could never export until the pc_dealloc signal made it through and thus the hang.)

Exhibit 2035. doc R400_Sequencer.doc 73569 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257748

ATI Ex. 2109 IPR2023-00922 Page 38 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 201544	GEN-CXXXXX-REVA	39 of 51
00.00.00				

23.2.3 SQ to SX(SP): Interpolator bus

Name	Direction	Bits	Description
SQ SPx interp flat vtx	SQ→SPx	2	Provoking vertex for flat shading
SQ SPx interp flat gouraud	SQ→SPx	1	Flat or gouraud shading
SQ SPx interp cyl wrap	SQ→SPx	4	Wich channel needs to be cylindrical wrapped
SQ_SPx_interp_param_gen	SQ→SPx	1	Generate Parameter
SQ SPx interp prim type	SQ→SPx	2	Bits [1:0] of primitive type sent by SC
SQ SPx interp buff swap	SQ→SPx	1	Swapp IJ buffers
SQ SPx interp IJ line	SQ→SPx	2	IJ line number
SQ SPx interp mode	SQ→SPx	1	Center/Centroid sampling
SQ SXx pc ptr0	SQ→SXx	11	Parameter Cache Pointer
SQ_SXx_pc_ptr1	SQ→SXx	11	Parameter Cache Pointer
SQ_SXx_pc_ptr2	SQ→SXx	11	Parameter Cache Pointer
SQ_SXx_rt_sel	SQ→SXx	1	Selects between RT and Normal data (Bit 2 of prim type)
SQ_SX0_pc_wr_en	SQ→SX0	8	Write enable for the PC memories
SQ_SX1_pc_wr_en	SQ→SX1	8	Write enable for the PC memories
SQ_SXx_pc_wr_addr	SQ→SXx	7	Write address for the PCs
SQ_SXx_pc_channel_mask	SQ→SXx	4	Channel mask
SQ_SXx_pc_ptr_valid	SQ→SXx	1	Read pointers are valid.
SQ_SPx_interp_valid	SQ→SPx	1	Interpolation control valid

23.2.4 SQ to SP: Staging Register Data

This is a broadcast bus that sends the VSISR information to the staging registers of the shader pipes.

Name	Direction	Bits	Description
SQ_SPx_vsr_data	SQ→SPx	96	Pointers of indexes or HOS surface information
SQ_SPx_vsr_double	SQ→SPx	1	0: Normal 96 bits per vert 1: double 192 bits per vert
SQ_SP0_vsr_valid	SQ→SP0	1	Data is valid
SQ_SP1_vsr_valid	SQ→SP1	1	Data is valid
SQ_SP2_vsr_valid	SQ→SP2	1	Data is valid
SQ_SP3_vsr_valid	SQ→SP3	1	Data is valid
SQ_SPx_vsr_read	SQ→SPx	1	Increment the read pointers

23.2.5 VGT to SQ : Vertex interface

23.2.5.1 Interface Signal Table

The area difference between the two methods is not sufficient to warrant complicating the interface or the state requirements of the VSISRs. <u>Therefore, the POR for this interface is that the VGT will transmit the data to the VSISRs (via the Shader Sequencer) in full, 32-bit floating-point format.</u> The VGT can transmit up to six 32-bit floating-point values to each VSISR where four or more values require two transmission clocks. The data bus is 96 bits wide. In the case where an event is sent the 5 LSBs of VGT_SQ_vsisr_data contain the eventID.

Name	Bits	Description				
VGT_SQ_vsisr_data	96	Pointers of indexes or HOS surface information				
VGT_SQ_event	1	VGT is sending an event				
VGT_SQ_vsisr_continued	1	0: Normal 96 bits per vert 1: double 192 bits per vert				
VGT_SQ_end_of_vtx_vect	1	Indicates the last VSISR data set for the current process vector (for double vector data, "end of vector" is set on the first vector)				
VGT_SQ_indx_valid	1	Vsisr data is valid				
VGT_SQ_state	3	Render State (6*3+3 for constants). This signal is guaranteed to be correct when "VGT_SQ_vgt_end_of_vector" is high.				
VGT_SQ_send	1	Data on the VGT_SQ is valid receive (see write-up for standard R400 SEND/RTR interface handshaking)				
SQ_VGT_rtr	1	Ready to receive (see write-up for standard R400 SEND/RTR interface handshaking)				

23.2.5.2 Interface Diagrams

Exhibit 2035.docR400_Sequences.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257749

ATI Ex. 2109 IPR2023-00922 Page 39 of 326

PROTECTIVE ORDER MATERIAL

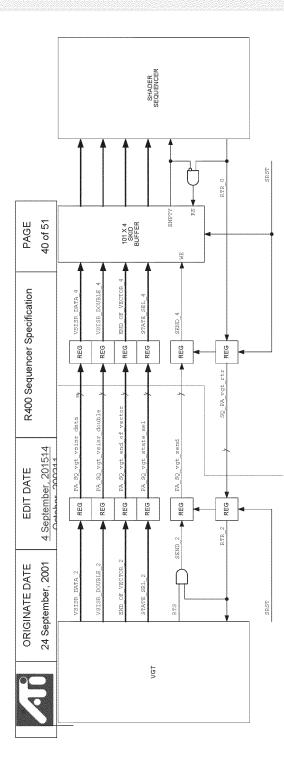
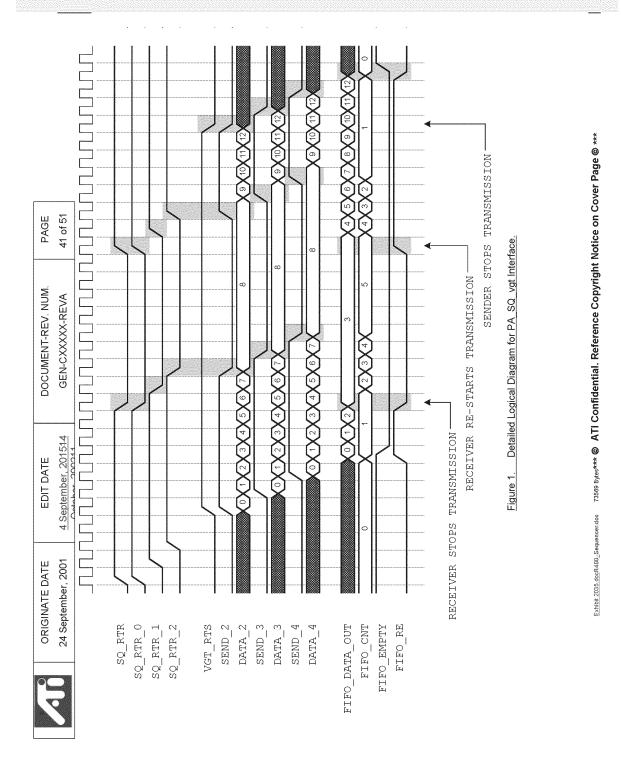


Exhibit 2035.docR400_Sequencer.doo 73589 Bytes*** @ ATI Confidential. Reference Copyright Notice on Cover Page @ ***

AMD1044_0257750

ATI Ex. 2109 IPR2023-00922 Page 40 of 326

PROTECTIVE ORDER MATERIAL



AMD1044_0257751

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
	24 September, 2001	4 September, 201514		42 of 51

23.2.6 SQ to SX: Control bus

Name	Direction	Bits	Description
SQ_SXx_exp_type	SQ→SXx	2	00: Pixel without z (1 to 4 buffers) 01: Pixel with z (1 to 4 buffers) 10: Position (1 or 2 results) 11: Pass thru (4,8 or 12 results aligned)
SQ_SXx_exp_number	SQ→SXx	2	Number of locations needed in the export buffer (encoding depends on the type see bellow).
SQ_SXx_exp_alu_id	SQ→SXx	1	ALU ID
SQ_SXx_exp_valid	SQ→SXx	1	Valid bit
SQ_SXx_exp_state	SQ→SXx	3	State Context
SQ_SXx_free_done	SQ→SXx	1	Pulse that indicates that the previous export is finished from the point of view of the SP. This does not necessarily mean that the data has been transferred to RB or PA, or that the space in export buffer for that particular vector thread has been freed up.
SQ_SXx_free_alu_id	SQ→SXx	1	ALU ID

Depending on the type the number of export location changes:

- Type 00 : Pixels without Z
 - 00 = 1 buffer
 - 01 = 2 buffers
 - 10 = 3 buffers
 - 11 = 4 buffer

.

.

- Type 01: Pixels with Z
 - 00 = 2 Buffers (color + Z)
 - 01 = 3 buffers (2 color + Z)
 - \circ 10 = 4 buffers (3 color + Z)
 - 11 = 5 buffers (4 color + Z)
- Type 10 : Position export
 - 00 = 1 position
 - 01 = 2 positions
 - 1X = Undefined
 - Type 11: Pass Thru
 - 00 = 4 buffers
 - 01 = 8 buffers
 - 10 = 12 buffers
 - 11 = Undefined

Below the thick black line is the end of transfer packet that tells the SX that a given export is finished. The report packet will always arrive either before or at the same time than the next export to the same ALU id.

23.2.7 SX to SQ : Output file control

Name	Direction	Bits	Description
SXx_SQ_exp_count_rdy	SXx→SQ	1	Raised by SX0 to indicate that the following two fields reflect the result of the most recent export
SXx_SQ_exp_pos_avail	SXx→SQ	2	Specifies whether there is room for another position. 00 : 0 buffers ready 01 : 1 buffer ready 10 : 2 or more buffers ready
SXx_SQ_exp_buf_avail	SXx→SQ	7	Specifies the space available in the output buffers. 0: buffers are full 1: 2K-bits available (32-bits for each of the 64

Exhibit 2035.doc.F4400_Sequencer.doc 73569 Bytes***
CATI Confidential. Reference Copyright Notice on Cover Page

AMD1044_0257752

ATI Ex. 2109 IPR2023-00922 Page 42 of 326

ORIGINATE DATE 24 September, 2001	 DATE er, 201514	DOCUMENT-REV. NUM. GEN-CXXXXX-REVA	PAGE 43 of 51
	 64: 128 64 pixe	n a clause) 3K-bits available (16 128-bit entries fo els) ∵ RESERVED	r each of

23.2.8 SQ to TP: Control bus

Once every clock, the fetch unit sends to the sequencer on which RS line it is now working and if the data in the GPRs is ready or not. This way the sequencer can update the fetch valid bits flags for the reservation station. The sequencer also provides the instruction and constants for the fetch to execute and the address in the register file where to write the fetch return data.

Name	Direction	Bits	Description
TPx_SQ_data_rdy	$TPx \rightarrow SQ$	1	Data ready
TPx_SQ_rs_line_num	$TPx \rightarrow SQ$	6	Line number in the Reservation station
TPx_SQ_type	TPx→ SQ	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_TPx_send	SQ→TPx	1	Sending valid data
SQ_TPx_const	SQ→TPx	48	Fetch state sent over 4 clocks (192 bits total)
SQ_TPx_instr	SQ→TPx	24	Fetch instruction sent over 4 clocks
SQ_TPx_end_of_group	SQ→TPx	1	Last instruction of the group
SQ_TPx_Type	SQ→TPx	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_TPx_gpr_phase	SQ→TPx	2	Write phase signal
SQ_TP0_lod_correct	SQ→TP0	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP0_pix_mask	SQ→TP0	4	Pixel mask 1 bit per pixel
SQ_TP1_lod_correct	SQ→TP1	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP1_pix_mask	SQ→TP1	4	Pixel mask 1 bit per pixel
SQ_TP2_lod_correct	SQ→TP2	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP2_pix_mask	SQ→TP2	4	Pixel mask 1 bit per pixel
SQ_TP3_lod_correct	SQ→TP3	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP3_pix_mask	SQ→TP3	4	Pixel mask 1 bit per pixel
SQ_TPx_rs_line_num	SQ→TPx	6	Line number in the Reservation station
SQ_TPx_write_gpr_index	SQ->TPx	7	Index into Register file for write of returned Fetch Data
SQ_TPx_ctx_id	SQ→TPx	3	The state context ID (needed for multisample resolves)

23.2.9 TP to SQ: Texture stall

The TP sends this signal to the SQ and the SPs when its input buffer is full.

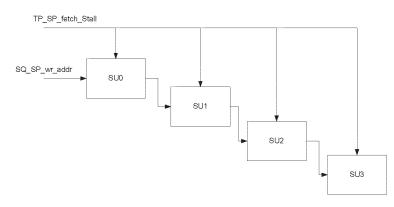


Exhibit 2035.docR400_Sequences.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257753

ATI Ex. 2109 IPR2023-00922 Page 43 of 326

AI	ORIGINATE DATE 24 September, 2001			R400 Sequencer Specification	PAGE 44 of 51
NameDirectionTP_SQ_fetch_stallTP \rightarrow S			Bits 1	 cription not send more texture request if asserted	d

23.2.10 SQ to SP: Texture stall

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

23.2.11 SQ to SP: GPR and auto counter

Name	Direction	Bits	Description
SQ_SPx_gpr_wr_addr	SQ→SPx	7	Write address
SQ_SPx_gpr_rd_addr	SQ→SPx	7	Read address
SQ_SPx_gpr_rd_en	SQ→SPx	1	Read Enable
SQ_SP0_gpr_wr_en	SQ→SPx	4	Write Enable for the GPRs of SP0
SQ_SP1_gpr_wr_en	SQ→SPx	4	Write Enable for the GPRs of SP1
SQ_SP2_gpr_wr_en	SQ→SPx	4	Write Enable for the GPRs of SP2
SQ_SP3_gpr_wr_en	SQ→SPx	4	Write Enable for the GPRs of SP3
SQ_SPx_gpr_phase	SQ→SPx	2	The phase mux (arbitrates between inputs, ALU SRC
			reads and writes)
SQ_SPx_channel_mask	SQ→SPx	4	The channel mask
SQ_SPx_gpr_input_sel	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_auto_count	SQ→SPx	21	Auto count generated by the SQ, common for all shader pipes

Exhibit 2035.docR400_Sequencer.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257754

ATI Ex. 2109 IPR2023-00922 Page 44 of 326

22	ORIGINATE	DATE	EDIT	DATE	DOCUMENT-REV. NUM.	PAGE
40	24 September		4 Septemb	er, 201544	GEN-CXXXXX-REVA	45 of 51
.2.12 \$	SQ to SPx: In	structio		200211	1	1
Name		Direct		Bits Des	cription	
SQ_SPx_i		SQ→S			uction start	
SQ_SP_in	Istr	SQ→S	5PX	0: s s v l	isferred over 4 cycles RC A Negate Argument Modifier 0:0 RC A Abs Argument Modifier 1:1 RC A Swizzle 9:2 ector Dst 15:10 ler channel Select 23:16 00: GPR 01: PV 10: PS 11: Constant (if 11 has to nnels)	be 11 for all
				9 9 9 1	RC B Negate Argument Modifier 0:0 RC B Abs Argument Modifier 1:1 RC B Swizzle 9:2 calar Dst 15:10 ter channel Select 23:16 00: GPR 01: PV 10: PS 11: Constant (if 11 has to nnels)	be 11 for all
				S S L I	RC C Negate Argument Modifier 0:0 RC C Abs Argument Modifier 1:1 RC C Swizzle 9:2 nused 15:10 ler channel Select 23:16 00: GPR 01: PV 10: PS 11: Constant (if 11 has to nnels)	be 11 for all
					ector Opcode 4:0 calar Opcode 10:5 ector Clamp 11:11 calar Clamp 12:12 ector Write Mask 16:13 calar Write Mask 20:17 nused 23:21	
SQ_SP0_	pred_override	SQ→S	3P0	logi	se per channel RGBA field (enables the ;, if not set only pay attention to the 11 s se GPR	
	pred_override	SQ→S		logi 1: U	se per channel RGBA field (enables the e, if not set only pay attention to the 11 s se GPR	eting).
	pred_override	SQ→S		logi 1: L	se per channel RGBA field (enables the , if not set only pay attention to the 11 s se GPR	eting).
	pred_override	SQ→S		logi 1: U	se per channel RGBA field (enables the c, if not set only pay attention to the 11 s se GPR	
SQ_SPx_	exp id	SQ→S	SPx	1 GP		

ATI Ex. 2109 IPR2023-00922 Page 45 of 326

	Aî	ORIGINATE DAT 24 September, 20	-	EDIT DA	201514	R400 Sequencer Specification	PAGE 46 of 51
	SQ_SPx_exporting SQ→SF		→SPx	1	0: Not Exporting		
1						1: Exporting	
	SQ_SPx_stall SQ→SPx		→SPx	1	Stall signal		

23.2.13 SQ to SX: write mask interface (must be aligned with the SP data)

Name	Direction	Bits	Description
SQ_SX0_write_mask	SQ→SP0	8	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. This is for the data coming of SP0 and SP2.
SQ_SX1_ write_mask	SQ→SP1	8	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. This is for the data coming of SP1 and SP3.

23.2.14 SP to SQ: Constant address load/ Predicate Set/Kill set

Name	Direction	Bits	Description
SP0_SQ_const_addr	SP0→SQ	36	Constant address load / predicate vector load (4 bits only)/
			Kill vector load (4 bits only) to the sequencer
SP0_SQ_valid	SP0→SQ	1	Data valid
SP1_SQ_const_addr	SP1→SQ	36	Constant address load / predicate vector load (4 bits only)/
			Kill vector load (4 bits only) to the sequencer
SP1_SQ_valid	SP1→SQ	1	Data valid
SP2_SQ_const_addr	SP2→SQ	36	Constant address load / predicate vector load (4 bits only)/
			Kill vector load (4 bits only) to the sequencer
SP2_SQ_valid	SP2→SQ	1	Data valid
SP3_SQ_const_addr	SP3→SQ	36	Constant address load / predicate vector load (4 bits only)/
			Kill vector load (4 bits only) to the sequencer
SP3_SQ_valid	SP3→SQ	1	Data valid
SP0_SQ_data_type	SP→SQ	2	Data Type
			0: Constant Load
			1: Predicate Set
			2: Kill vector load

Because of the sharing of the bus none of the MOVA, PREDSET or KILL instructions may be coissued.

23.2.15 SQ to SPx: constant broadcast

Name	Direction	Bits	Description	
SQ_SPx_const	SQ→SPx	128	Constant broadcast	

23.2.16 SQ to CP: RBBM bus

Name	Direction	Bits	Description
SQ_RBB_rs	SQ→CP	1	Read Strobe
SQ_RBB_rd	SQ→CP	32	Read Data
SQ_RBBM_nrtrtr	SQ→CP	1	Optional
SQ_RBBM_rtr	SQ→CP	1	Real-Time (Optional)

23.2.17 CP to SQ: RBBM bus

Name	Direction	Bits	Description	
rbbm_we	CP→SQ	1	Write Enable	
rbbm_a	CP→SQ	15	Address Upper Extent is TBD (16:2)	
rbbm_wd	CP→SQ	32	Data	
rbbm_be	CP→SQ	4	Byte Enables	
rbbm_re	CP→SQ	1	Read Enable	
rbb_rs0	CP→SQ	1	Read Return Strobe 0	

Exhibit 2035.doc.R400_Sequencer.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257756

	ORIGINATE DATE 24 September, 2001		EDIT DATE			DOCUMENT-REV. NUM. GEN-CXXXXX-REVA	PAGE 47 of 51
rbb_rs1		CP→SQ		1	Read I	Return Strobe 1	
rbb_rd0		CP→SQ		32	Read	Data 0	
rbb_rd1		CP→SQ		32	Read	Data 0	
RBBM_S	Q_soft_reset	CP→SQ		1	Soft R	eset	

23.2.18 SQ to CP: State report

Name	Direction	Bits	Description
SQ_CP_vs_event	SQ→CP	1	Vertex Shader Event
SQ_CP_vs_eventid	SQ→CP	5	Vertex Shader Event ID
SQ_CP_ps_event	SQ→CP	1	Pixel Shader Event
SQ_CP_ps_eventid	SQ→CP	5	Pixel Shader Event ID

23.3 Example of control flow program execution

We now provide some examples of execution to better illustrate the new design.

Given the program:

Alu 0 Alu 1 Tex 0 Tex 1 Alu 3 Serial Alu 4 Tex 2 Alu 5 Alu 6 Serial Tex 3 Alu 7 Alloc Position 1 buffer Alu 8 Export Tex 4 Alloc Parameter 3 buffers Alu 9 Export 0 Tex 5 Alu 10 Serial Export 2 Alu 11 Export 1 End

Would be converted into the following CF instructions:

Execute 0 Alu 0 Alu 0 Tex 0 Tex 1 Alu 0 Alu 0 Tex 0 Alu 1 Alu 0 Tex Execute 0 Alu Alloc Position 1 Execute 0 Alu 0 Tex Alloc Param 3 Execute end 0 Alu 0 Tex 1 Alu 0 Alu

And the execution of this program would look like this:

Put thread in Vertex RS:

Control Flow Instruction Pointer (12 bits), (CFP) Execution Count Marker (3 or 4 bits), (ECM) Loop Iterators (4x9 bits), (LI) Call return pointers (4x12 bits), (CRP) Predicate Bits(4x64 bits), (PB) Export ID (1 bit), (EXID) GPR Base Ptr (8 bits), (GPR)

Exhibit 2035. docR400_Sequences.doc 73569 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257757

ATI Ex. 2109 IPR2023-00922 Page 47 of 326

	ORIGIN	ATE DATE		EDIT DA	те	R40	0 Sequenc	er Speci	fication	PAGE		
	24 Septe	mber, 2001		otember,						48 of 51		
Export Base Ptr (7 bits), (EB) Context Ptr (3 bits).(CPTR) LOD correction bits (16x6 bits) (LOD)												
State Bits												
CFP ECM LI CRP PB EXID GPR EB CPTR LOD												
0 0	0 0 0 0 0 0 0 0 0 0 0											
Texture// Texture I Waiting 0 Allocation 00 – 01 – 10 – Allocation Position First thre	Valid Thread (VALID) Texture/ALU engine needed (TYPE) Texture Reads are outstanding (PENDING) Waiting on Texture Read to Complete (SERIAL) Allocation Wait (2 bits) (ALLOC) 00 – No allocation needed 01 – Position export allocation needed (ordered export) 10 – Parameter or pixel export needed (ordered export) 11 – pass thru (out of order export) Allocation Size (4 bits) (SIZE) Position Allocated (POS_ALLOC) First thread of a new context (FIRST) Last (1 bit), (LAST) Status Bits											
	TYPE	PENDING	SEF		ALLOC	SIZE	POS_AL	LOC	FIRST	LAST		
1	ALU	0	0	1	0	0	0		1	0		
Execute 0 It execut	Then the thread is picked up for the execution of the first control flow instruction: Execute 0 Alu 0 Alu 0 Tex 0 Tex 1 Alu 0 Alu 0 Tex 0 Alu 1 Alu 0 Tex It executes the first two ALU instructions and goes back to the RS for a resource request change. Here is the state returned to the RS:											

State Bits											
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD		
0	2	0	0	0	0	0	0	0	0		

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	0	0	0	0	0	1	0

Then when the texture pipe frees up, the arbiter picks up the thread to issue the texture reads. The thread comes back in this state:

State Bits	State Bits											
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD			
0	4	0	0	0	0	0	0	0	0			

Status Bits					201010-0012-002-002-002-002-002-002-002-			
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	1	0	0	0	1	0

Because of the serial bit the arbiter must wait for the texture to return and clear the PENDING bit before it can pick the thread up. Lets say that the texture reads are complete, then the arbiter picks up the thread and returns it in this state:

Exhibit 2035.docR400_Sequences.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257758

ATI Ex. 2109 IPR2023-00922 Page 48 of 326

ORIGINATE DATE EDIT DATE DOCUMENT-REV. NUM. 24 September, 2001 4 September, 201544 GEN-CXXXXX-REVA State Bits												
State Bits CFP ECM LI CRP PB EXID GPR EB CPTR LOD												
0	6	0 0		0	0	0	0	0	0			

Status Bit	s							
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	0	0	0	0	0	1	0

Again the TP frees up, the arbiter picks up the thread and executes. It returns in this state:

State Bits											
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD		
0	7	0	0	0	0	0	0	0	0		

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	0	0	0	1	0

Now, even if the texture has not returned we can still pick up the thread for ALU execution because the serial bit is not set. The thread will however come back to the RS for the second ALU instruction because it has the serial bit set.

State Bits										
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD	
0	8	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	1	0	0	0	1	0

As soon as the TP clears the pending bit the thread is picked up and returns:

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
0	9	0	0	0	0	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	0	0	0	0	0	1	0

Picked up by the TP and returns:

Exe	cute 0 Al	u							
State B									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
1	0	0	0	0	0	0	0	0	0

Exhibit 2035.doc.B400_Sequences.doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257759

ATI Ex. 2109 IPR2023-00922 Page 49 of 326

		TE DATE nber, 2001	EDIT E	r, 201544	R400) Sequencer Spec	PAGE 50 of 51	
Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	0	0	0	1	0

Picked up by the ALU and returns (lets say the TP has not returned yet): Alloc Position 1 $\,$

State Bits													
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD				
2	0	0	0	0	0	0	0	0	0				

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	01	1	0	1	0

If the SX has the place for the export, the SQ is going to allocate and pick up the thread for execution. It returns to the RS in this state:

Execute 0 Alu 0 Tex

State Bits	****								
	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
3	1	0	0	0	0	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	1	0	0	0	1	1	0

Now, since the TP has not returned yet, we must wait for it to return because we cannot issue multiple texture requests. The TP returns, clears the PENDING bit and we proceed:

Alloc Param 3

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
4	0	0	0	0	1	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	10	3	1	1	0

Once again the SQ makes sure the SX has enough room in the Parameter cache before it can pick up this thread.

Execute_end 0 Alu 0 Tex 1 Alu 0 Alu

State Bit										
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD	
5	1	0	0	0	1	0	100	0	0	

Exhibit 2035. doc R400_Sequencer.doc 73569 Bytes***
C ATI Confidential. Reference Copyright Notice on Cover Page
C ***

AMD1044_0257760

ATI Ex. 2109 IPR2023-00922 Page 50 of 326

A n		IATE DATE ember, 2001	EDIT DATE <u>4 September, 201514</u>			DOCUMENT-REV. NUM. GEN-CXXXXX-REVA			PAGE 51 of 51
Status Bits			October 200211						
VALID	TYPE PENDING		SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LA	ST
1	TEX	1	0	0	0	1	1	0	

This executes on the TP and then returns:

CFP ECM LI CRP PB EXID GPR EB CPTR		
CPP ECMI LI CRP PD EXID GPR ED CPIR	LOD	
5 2 0 0 0 1 0 100 0	0	

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	1	0	0	1	1	1

Waits for the TP to return because of the textures reads are pending (and SERIAL in this case). Then executes and does not return to the RS because the LAST bit is set. This is the end of this thread and before dropping it on the floor, the SQ notifies the SX of export completion.

24. Open issues

Need to do some testing on the size of the register file as well as on the register file allocation method (dynamic VS static).

Saving power?

Exhibit 2035. docR400_Sequences..doc 73569 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257761

ATI Ex. 2109 IPR2023-00922 Page 51 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 201546	GEN-CXXXXX-REVA	1 of 51
Author:	Laurent Lefebvre	October 200214		1
Issue To:		Copy No:		
		I		
	R400 S	equencer Spe	cification	
	14000	equencer ope		
		0.0		
		SQ		
		Version 2.0 <u>8</u> 7		
Overview: Th	is is an architectural specifi	cation for the R400 Sequen	cer block (SEQ). It provides an ov	verview of the
re		cted uses of the block. It a	lso describes the block interfaces,	
50	ono, una provideo internal o	ate diagramo.		
	CALLY UPDATED FIELDS:			
Document I	_ocation: C:\p		locks\sq\R400_Sequencer.doc	
	anet Search Title: R40	0 Sequencer Specification APPROVALS		
	Name/Dept		Signature/Date	
Remarks:				
THIS DC	CUMENT CONTAIN	IS CONFIDENTIAL I	NFORMATION THAT CC	ULD BE
	ANTIALLY DETRIME	INTAL TO THE INTE	REST OF ATI TECHNOL	
	INC. THROUGH U	JNAUTHORIZED US	SE OR DISCLOSURE.	
			rial in this document constitutes ar provide notice that ATI owns a co	
			publication has occurred. This v of this document may be used, re	
			sion of ATI Technologies Inc."	
	ncR400_Sequencer.doc 73119 Bytes*** (nce Copyright Notice on Cover Pa	

ATI 2036 LG v. ATI IPR2015-00325

AMD1044_0257762

ATI Ex. 2109 IPR2023-00922 Page 52 of 326

	100			Ι	
	ň	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
	<u>M</u>	24 September, 2001	4 September, 201546		2 of 51
Table	Of (<u>Contents</u>			
1. C	VEF		****		7
1.1 T	op L	evel Block Diagram			9
		,			
5.3.	1	R400 Constant manag	gement		
5.3.				t	
5.3.					
5.3.					
5.3.					
5.3.					
		0			
6.2.		0			
6.5 H	łW D	etection of PV,PS			
6.6 F	Regis	ter file indexing			
6.7 E)ebuį	gging the Shaders			
6.7.	1	Method 1: Debugging	registers		
6.7.	2	Method 2: Exporting tl	he values in the GPRs	3	
7. F	IXEL	KILL MASK			<u>27</u> 26
1			()		Contraction of the Contraction o
1				****	CONCORANT
				IFOS	
				IFUS	
15.1					
		1			
Exhibit	2036.doc	R400_Sequencer.doc 73119 Bytes***	ATI Confidential. Ref	erence Copyright Notice on Cover P	age © ***

ATI Ex. 2109 IPR2023-00922 Page 53 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE				
	24 September, 2001	4 September, 201546	GEN-CXXXXX-REVA	3 of 51				
,								
17.1.1								
17.1.2								
17.1.3								
18. EXPORT TYPES 32 18.1 Vertex Shading 32								
18.2 Pixel Shading								
19. SPECIAL INTERPOLATION MODES								
19.3.1								
19.3.2								
	,							
23.2.1								
23.2.2								
23.2.3								
23.2.4								
23.2.5								
23.2.6								
23.2.7								
23.2.7	,							
23.2.8								
23.2.9								
23.2.11								
23.2.12								
23.2.13			Set/Kill set					
23.2.14								
23.2.15								
23.3 Exa	mple of control flow pro	gram execution						

ATI Ex. 2109 IPR2023-00922 Page 54 of 326

ſ	Añ	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
		24 September, 2001	4 September, 201516		4 of 51
	24. OPEN	ISSUES	*****		

Exhibit 2036.doc R400_Sequencer.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257765

ATI Ex. 2109 IPR2023-00922 Page 55 of 326

				Ē			
	ORIGINAT	E DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE		
Can	24 Septemb	per, 2001	4 September, 201546	GEN-CXXXXX-REVA	5 of 51		
Revision	Changes	x x					
				~			
Rev 0.1 (Laur Date: May 7, 2	rent Lefebvre) 2001		First di	aft.			
Rev 0.2 (Laure			Changed the interfaces to reflect the changes in the				
Date : July 9, 2 Rev 0.3 (Laure			SP. Added some details in the arbitration section. Reviewed the Sequencer spec after the meeting on				
Date : August				t 3, 2001.	any on		
Rev 0.4 (Laure			Added	the dynamic allocation method for			
Date : August	24, 2001			d an example (written in part by Vic) of the		
Rev 0.5 (Laure	ent Lefebvre)			pixels/vertices in the sequencer. timing diagrams (Vic)			
Date : Septem							
Rev 0.6 (Laure				ed the spec to reflect the new	R400		
Date : Septem Rev 0.7 (Laure	,			cture. Added interfaces. constant store management, ins	truction		
Date : October				management, control flow management			
D 0.0 //				ependant predication.			
Rev 0.8 (Laure Date : October			-	ed the control flow method to be e. Also updated the external interfaces.			
Rev 0.9 (Laure				prated changes made in the 10/18/01			
Date : October	r 17, 2001			neeting. Added a NOP instruction, re			
			the c registe	onditional_execute_or_jump. Added	depug		
Rev 1.0 (Laure	ent Lefebvre)		•	d interfaces to RB. Added state registe	ers.		
Date : October							
Rev 1.1 (Laure Date : October				SEQ→SP0 interfaces. Changed on. Changed VGT→SP0 interface.			
Dute : Obtober	1 20, 2001		Methods added.				
Rev 1.2 (Laure			Interfaces greatly refined. Cleaned up the spec.				
Date : Novemb Rev 1.3 (Laure			Added the different interpolation modes.				
Date : Novemb			Added the different interpolation modes.				
Rev 1.4 (Laure			Added the auto incrementing counters. Changed				
Date : Decemt	ber 6, 2001		the VGT→SQ interface. Added content on constant management. Updated GPRs.				
Rev 1.5 (Laure	ent Lefebvre)		Removed from the spec all interfaces that weren't				
Date : Decemb	ber 11, 2001		directly tied to the SQ. Added explanations on				
				constant management. Added PA→SQ synchronization fields and explanation.			
Rev 1.6 (Laure	ent Lefebvre)			more details on the staging register.	Added		
Date : January			detail	about the parameter caches. Chang	ged the		
				struction to a Conditionnal_call inst details on constant management			
				d the diagram.			
Rev 1.7 (Laure				Real Time parameter control in	the SX		
Date : Februar Rev 1.8 (Laure				ce. Updated the control flow section. hterfaces to the SX block. Added the	and of		
Date : March 4				modifier, removed the end of			
			instruc	tions.			
Rev 1.9 (Laure Date : March 1				ngement of the CF instruction bits in byte alignement.	order to		
Rev 1.10 (Lau				ed the interfaces and added a sec	tion on		
Date : March 2				ng rules.			
Rev 1.11 (Lau Date : April 19				CP state report interface. Last versio vith the old control flow scheme	n of the		
Rev 2.0 (Laure	ent Lefebvre)			ontrol flow scheme			
Date : April 19	, 2002						
			ATI Confidential Defer	ance Convright Notice on Cover De	10 @ +++		
Exhibit 2036.docR	1400_Sequencer.doc	73119 Bytes*** C	A li Conndential. Kefer	ence Copyright Notice on Cover Pag	Je © ***		

ATI Ex. 2109 IPR2023-00922 Page 56 of 326

	F	e		r			
	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE			
	24 September, 2001	4 September, 201546		6 of 51			
Rev 2.01 (Lau	rent Lefebvre)	Cha	nged slightly the control flow instru	uctions to			
Date : May 2,	2002	allov	v force jumps and calls.				
	rent Lefebvre)		ated the Opcodes. Added type fie				
Date : May 13	, 2002		constant/pred interface. Added Last field to the				
			→SP instruction load interface.				
	rent Lefebvre)		SP interface updated to include predication				
Date : July 15	, 2002		mizations. Added the predicate	no stall			
		instructions,					
	rent Lefebvre)	Documented the new parameter generation scheme					
Date :August 2		for XY coordinates points and lines STs.					
	rent Lefebvre)		Some interface changes and an architectural				
Date : Septem			nge to the auto-counter scheme.	.,			
	rent Lefebvre)		Widened the event interface to 5 bits. Some other				
Date : Octobe	/		little typos corrected.				
	rent Lefebvre)		Loops, jumps and calls are now using a 13 bit				
Date : Octobe	r 14, 2002		ress which allows to jump and call				
			and any control flow addresses (does not			
Day 2.00 (Lau	rent Lefebvre)		ires to be even anymore). ification updates after discussion with t	Clay			
Date : Octobe			incation updates alter discussion with	Ulay.			
Date . Octobe	110, 2002						

Exhibit 2036.docR400_Sequencer.doc 73119 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257767

ATI Ex. 2109 IPR2023-00922 Page 57 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 201546	GEN-CXXXXX-REVA	7 of 51

1. Overview

The sequencer chooses two ALU threads and a fetch hread to execute, and executes all of the instructions in a block before looking for a new clause of the same type. Two ALU threads are executed interleaved to hide the ALU latency. The arbitrator will give priority to older threads. There are two separate reservation stations, one for pixel vectors and one for vertices vectors. This way a pixel can pass a vertex and a vertex can pass a pixel.

To support the shader pipe the sequencer also contains the shader instruction cache, constant store, control flow constants and texture state. The four shader pipes also execute the same instruction thus there is only one sequencer for the whole chip.

The sequencer first arbitrates between vectors of 64 vertices that arrive directly from primitive assembly and vectors of 16 quads (64 pixels) that are generated in the scan converter.

The vertex or pixel program specifies how many GPRs it needs to execute. The sequencer will not start the next vector until the needed space is available in the GPRs.

Exhibit 2036.docR400_Sequencer.doc 73119 Bytes**** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257768

ATI Ex. 2109 IPR2023-00922 Page 58 of 326

PROTECTIVE ORDER MATERIAL

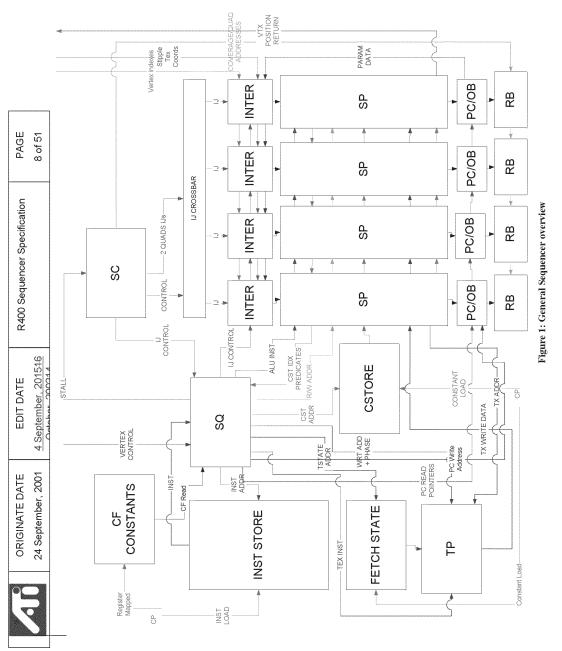


Exhibit 2036 docR400_Sequencerdoo 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257769

ATI Ex. 2109 IPR2023-00922 Page 59 of 326

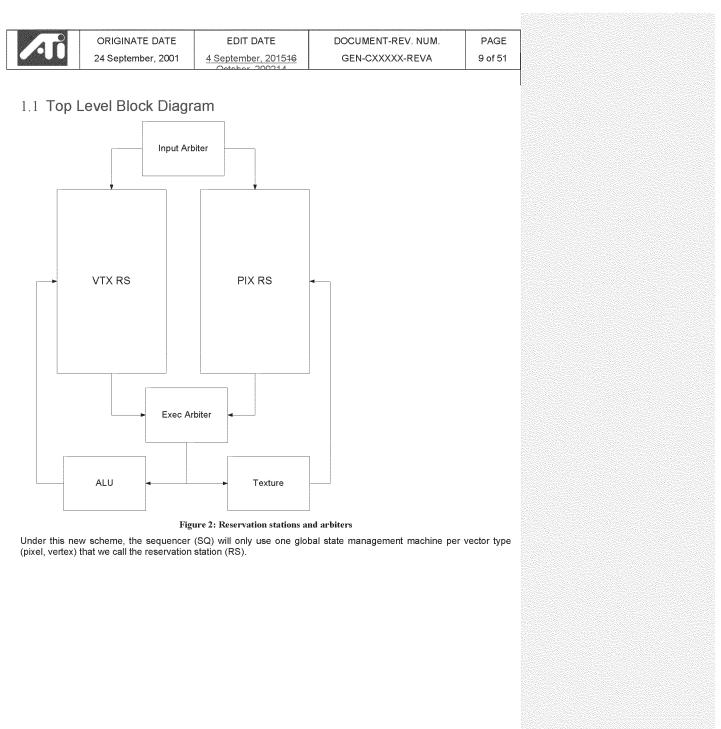
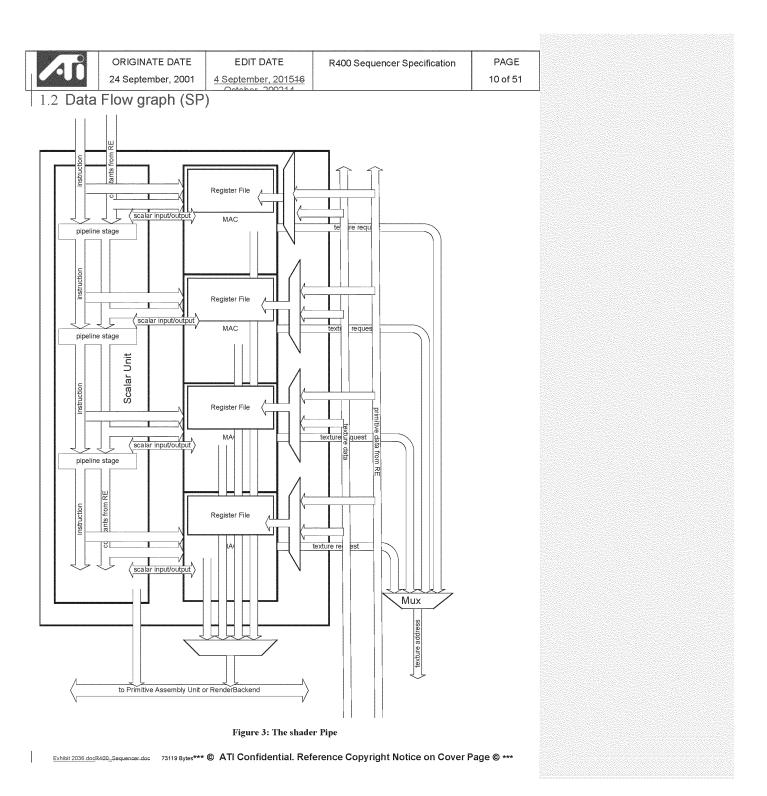


Exhibit 2036.docR400_Sequences.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257770

ATI Ex. 2109 IPR2023-00922 Page 60 of 326



ATI Ex. 2109 IPR2023-00922 Page 61 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 201546	GEN-CXXXXX-REVA	11 of 51

The gray area represents blocks that are replicated 4 times per shader pipe (16 times on the overall chip).

1.3 Control Graph

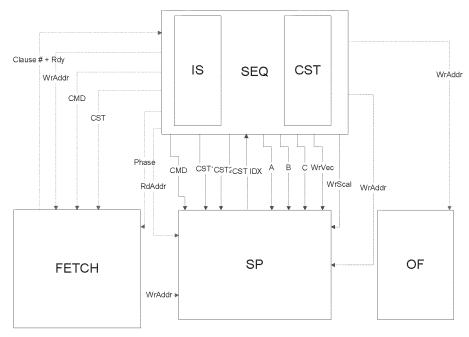


Figure 4: Sequencer Control interfaces

In green is represented the Fetch control interface, in red the ALU control interface, in blue the Interpolated/Vector control interface and in purple is the output file control interface.

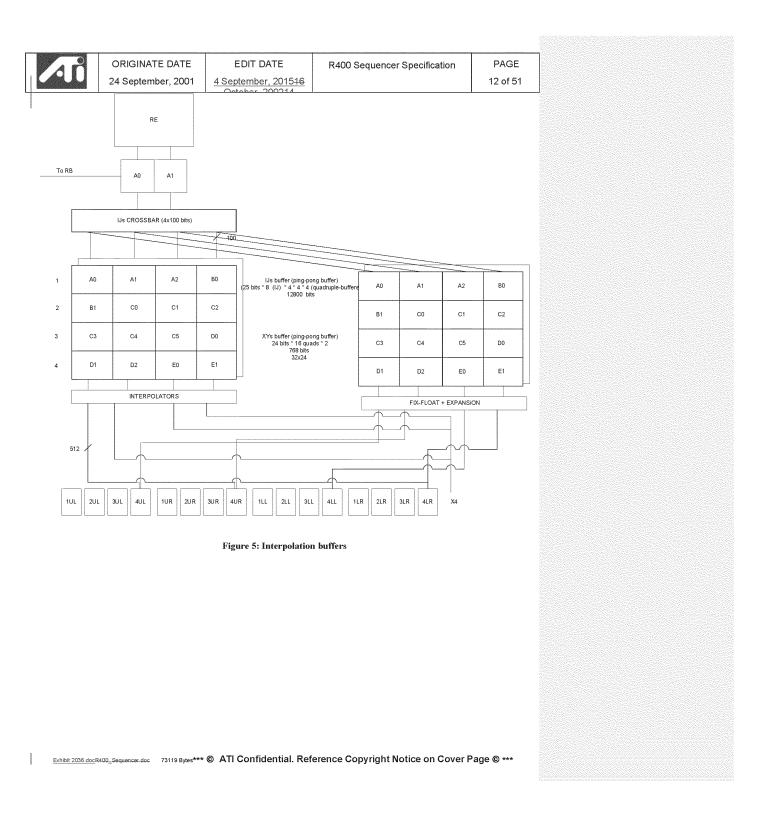
2. Interpolated data bus

The interpolators contain an IJ buffer to pack the information as much as possible before writing it to the register file.

Exhibit 2036 doc R400_Sequences.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257772

ATI Ex. 2109 IPR2023-00922 Page 62 of 326



ATI Ex. 2109 IPR2023-00922 Page 63 of 326

PROTECTIVE ORDER MATERIAL

		T23					5 48 <	52- 55	59 - 56 -<	2 0 8	
		T22					V V 32- 48- 35 51	V V 36-52- 39 55	V V 40-56- 43 59	V V 44-60- 47 63	×
		T18 T19 T20 T21 T22 T23					- 16 − 19 16 −	23 ² 0 <	24- 27	31 <mark>2</mark> 8 <	× L>
		T20					> %	> 7-4	, ⇔ =	>4²	
		Т19							EO	Ш	
										8	
щ	51	T17			≿ ଘ	×Ξ		8	5	ß	
PAGE	13 of 51	T16			ы	ш				BO	Р2
		T15			В	Ш.	Б	D2			Δ
NUM.	VA	T14	۶۶	≿8		≿8	ប	C4	C5		
SEV. P	X-RE	T13	δ	8		ß	B1				
DOCUMENT-REV. NUM.	GEN-CXXXX-REVA	T12	б	6		8	AO	A1	A2		
DCUN	GEN-(T11		≿ 2	≿ ვ				ОШ	μ	
Ď	_	T10 T11 T12	i.	2	C2	0	¥			8	
		T9		5	C5			8	5	ß	
Щ	20154	Т8	≿ຬ	≿ ວ	<u>ک</u> 2	∑ S				BO	
EDIT DATE	mber,	1	ខ	8	δ	S	Б	D 2			à
Ш	4 September, 201516	Т6	ខ	8	δ	3	ប៊	5	C5		
		Т5	×₽			∑ 8	B1				
ATE	2001	T4	8			BO	AO	A1	A2		
ATED	mber,	T3	Б			BO	51 48 ×	55 55	56 ⁻ X	S°X	
RIGINATE DATE	September, 2001	T2	ΧQ	۶¥	∑		35 ³ 2 ×	လ္လ ဗို ဗိ	∑ 4 £	XY XY 44- 60- 47 63	\geq
ō	24	È	Ao	A1	A2		, 16 19 19	33 ² 0 ×	24- 27		\times
90		D 10	Ao	A1	A2		53 X	× × -4	≿ ∾ ≿	15 ¦2 ≷	
	Z		g o	- S	⊳ S	ი ი	q° о	с С	⊳ SP	ыSр	

Figure 6: Interpolation timing diagram

Exhibit 2036 doc/R400_Sequences 73119 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257774

ATI Ex. 2109 IPR2023-00922 Page 64 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 201546		14 of 51

Above is an example of a tile the sequencer might receive from the SC. The write side is how the data get stacked into the XY and IJ buffers, the read side is how the data is passed to the GPRs. The IJ information is packed in the IJ buffer 4 quads at a time or two clocks. The sequencer allows at any given time as many as four quads to interpolate a parameter. They all have to come from the same primitive. Then the sequencer controls the write mask to the GPRs to write the valid data in.

3. Instruction Store

There is going to be only one instruction store for the whole chip. It will contain 4096 instructions of 96 bits each.

It is likely to be a 1 port memory; we use 1 clock to load the ALU instruction, 1 clocks to load the Fetch instruction, 1 clock to load 2 control flow instructions and 1 clock to write instructions.

The instruction store is loaded by the CP thru the register mapped registers.

The VS_BASE and PS_BASE context registers are used to specify for each context where its shader is in the instruction memory.

For the Real time commands the story is quite the same but for some small differences. There are no wrap-around points for real time so the driver must be careful not to overwrite regular shader data. The shared code (shared subroutines) uses the same path as real time.

4. Sequencer Instructions

All control flow instructions and move instructions are handled by the sequencer only. The ALUs will perform NOPs during this time (MOV PV,PV, PS,PS) if they have nothing else to do.

5. Constant Stores

5.1 Memory organizations

A likely size for the ALU constant store is 1024x128 bits. The read BW from the ALU constant store is 128 bits/clock and the write bandwidth is 32 bits/clock (directed by the CP bus size not by memory ports).

The maximum logical size of the constant store for a given shader is 256 constants. Or 512 for the pixel/vertex shader pair. The size of the re-mapping table is 128 lines (each line addresses 4 constants). The write granularity is 4 constants or 512 bits. It takes 16 clocks to write the four constants. Real time requires 256 lines in the physical memory (this is physically register mapped).

The texture state is also kept in a similar memory. The size of this memory is 320x96 bits (128 texture states for regular mode, 32 states for RT). The memory thus holds 128 texture states (192 bits per state). The logical size exposes 32 different states total, which are going to be shared between the pixel and the vertex shader. The size of the re-mapping table to for the texture state memory is 32 lines (each line addresses 1 texture state lines in the real memory). The CP write granularity is 1 texture state lines (or 192 bits). The driver sends 512 bits but the CP ignores the top 320 bits. It thus takes 6 clocks to write the texture state. Real time requires 32 lines in the physical memory (this is physically register mapped).

The control flow constant memory doesn't sit behind a renaming table. It is register mapped and thus the driver must reload its content each time there is a change in the control flow constants. Its size is 320*32 because it must hold 8 copies of the 32 dwords of control flow constants and the loop construct constants must be aligned.

The constant re-mapping tables for texture state and ALU constants are logically register mapped for regular mode and physically register mapped for RT operation.

Exhibit 2036 docR400_Sequencer.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257775

ATI Ex. 2109 IPR2023-00922 Page 65 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 201546	GEN-CXXXXX-REVA	15 of 51

5.2 Management of the Control Flow Constants

The control flow constants are register mapped, thus the CP writes to the according register to set the constant, the SQ decodes the address and writes to the block pointed by its current base pointer (CF_WR_BASE). On the read side, one level of indirection is used. A register (SQ_CONTEXT_MISC.CF_RD_BASE) keeps the current base pointer to the control flow block. This register is copied whenever there is a state change. Should the CP write to CF after the state change, the base register is updated with the (current pointer number +1)% number of states. This way, if the CP doesn't write to CF the state is going to use the previous CF constants.

5.3 Management of the re-mapping tables

5.3.1 R400 Constant management

The sequencer is responsible to manage two re-mapping tables (one for the constant store and one for the texture state). On a state change (by the driver), the sequencer will broadside copy the contents of its re-mapping tables to a new one. We have 8 different re-mapping tables we can use concurrently.

The constant memory update will be incremental, the driver only need to update the constants that actually changed between the two state changes.

For this model to work in its simplest form, the requirement is that the physical memory MUST be at least twice as large as the logical address space + the space allocated for Real Time. In our case, since the logical address space is 512 and the reserved RT space can be up to 256 entries, the memory must be of sizes 1280 and above. Similarly the size of the texture store must be of 32*2+32 = 96 entries and above.

5.3.2 Proposal for R400LE constant management

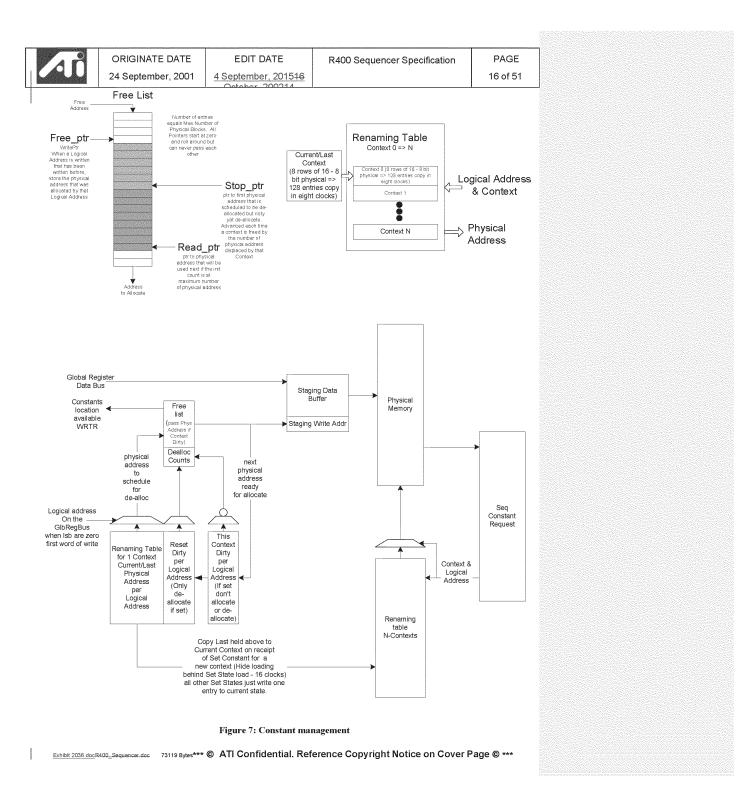
To make this scheme work with only 512+256 = 768 entries, upon reception of a CONTROL packet of state + 1, the sequencer would check for SQ_IDLE and PA_IDLE and if both are idle will erase the content of state to replace it with the new state (this is depicted in <u>Figure 8: De-allocation mechanismFigure 8: De-allocation mechanismFigure 8: De-allocation mechanismFigure 8: De-allocation mechanismFigure 8: De-allocation content of 0 is written to the corresponding de-allocation counter location so that when the SQ is going to report a state change, nothing will be de-allocated upon the first report.</u>

The second path sets all context dirty bits that were used in the current state to 1 (thus allowing the new state to reuse these physical addresses if needed).

Exhibit 2036.docR400_Sequencer.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257776

ATI Ex. 2109 IPR2023-00922 Page 66 of 326



ATI Ex. 2109 IPR2023-00922 Page 67 of 326

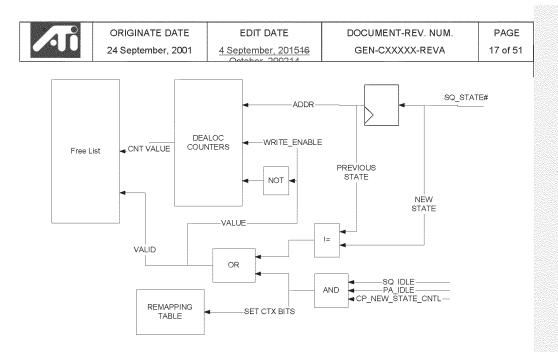


Figure 8: De-allocation mechanism for R400LE

5.3.3 Dirty bits

Two sets of dirty bits will be maintained per logical address. The first one will be set to zero on reset and set when the logical address is addressed. The second one will be set to zero whenever a new context is written and set for each address written while in this context. The reset dirty is not set, then writing to that logical address will not require de-allocation of whatever address stored in the renaming table. If it is set and the context dirty is not set, then the physical address is necessary to store the incoming data. If they are both set, then the data will be written into the physical address held in the renaming for the current logical address. No de-allocation or allocation takes place. This will happen when the driver does a set constant twice to the same logical address between context changes. NOTE: It is important to detect and prevent this, failure to do it will allow multiple writes to allocate all physical memory and thus hang because a context will not fit for rendering to start and thus free up space.

5.3.4 Free List Block

A free list block that would consist of a counter (called the IFC or Initial Free Counter) that would reset to zero and incremented every time a chunk of physical memory is used until they have all been used once. This counter would be checked each time a physical block is needed, and if the original ones have not been used up, us a new one, else check the free list for an available physical block address. The count is the physical address for when getting a chunk from the counter.

Storage of a free list big enough to store all physical block addresses.

Maintain three pointers for the free list that are reset to zero. The first one we will call write_ptr. This pointer will identify the next location to write the physical address of a block to be de-allocated. Note: we can never free more physical memory locations than we have. Once recording address the pointer will be incremented to walk the free list like a ring.

The second pointer will be called stop_ptr. The stop_ptr pointer will be advanced by the number of address chunks de-allocates when a context finishes. The address between the stop_ptr and write_ptr cannot be reused because they are still in use. But as soon as the context using then is dismissed the stop_ptr will be advanced.

The third pointer will be called read_ptr. This pointer will point will point to the next address that can be used for allocation as long as the read_ptr does not equal the stop_ptr and the IFC is at its maximum count.

Exhibit 2036.docR400_Sequencer.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257778

ATI Ex. 2109 IPR2023-00922 Page 68 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 201516		18 of 51

5.3.5 De-allocate Block

This block will maintain a free physical address block count for each context. While in current context, a count shall be maintained specifying how many blocks were written into the free list at the write_ptr pointer. This count will be reset upon reset or when this context is active on the back and different than the previous context. It is actually a count of blocks in the previous context that will no longer be used. This count will be used to advance the write_ptr pointer to make available the set of physical blocks in one clock.

5.3.6 Operation of Incremental model

The basic operation of the model would start with the write_ptr, stop_ptr, read_ptr pointers in the free list set to zero and the free list counter is set to zero. Also all the dirty bits and the previous context will be initialized to zero. When the first set constants happen, the reset dirty bit will not be set, so we will allocate a physical location from the free list counter is not at the max value. The data will be written into physical address zero. Both the additional copy of the renaming table and the context zeros of the big renaming table will be updated for the logical address that was written by set start with physical address of 0. This process will be repeated for any logical address that are not dirty until the context changes. If a logical address is hit that has its dirty bits set while in the same context, both dirty bits would be set, so the new data will be over-written to the last physical address assigned for this logical address. When the first draw command of the context is detected, the previous context location. Then the set constant logical address with be loaded with a new physical address during the copy and if the reset dirty was set, the physical address it replaced in the renaming table would be entered at the write_ptr pointer location on the free list and the write_ptr will be incremented. The de-allocation counter for the previous context (eight) will be incremented. This as set states come in for this context one of the following will happen:

- 1.) No dirty bits are set for the logical address being updated. A line will be allocated of the free-list counter or the free list at read_ptr pointer if read_ptr != to stop_ptr .
- 2.) Reset dirty set and Context dirty not set. A new physical address is allocated, the physical address in the renaming table is put on the free list at write_ptr and it is incremented along with the de-allocate counter for the last context.
- 3.) Context dirty is set then the data will be written into the physical address specified by the logical address.

This process will continue as long as set states arrive. This block will provide backpressure to the CP whenever he has not free list entries available (counter at max and stop_ptr == read_ptr). The command stream will keep a count of contexts of constants in use and prevent more than max constants contexts from being sent.

Whenever a draw packet arrives, the content of the re-mapping table is written to the correct re-mapping table for the context number. Also if the next context uses less constants than the current one all exceeding lines are moved to the free list to be de-allocated later. This happens in parallel with the writing of the re-mapping table to the correct memory.

Now preferable when the constant context leaves the last ALU clause it will be sent to this block and compared with the previous context that left. (Init to zero) If they differ than the older context will no longer be referenced and thus can be de-allocated in the physical memory. This is accomplished by adding the number of blocks freed this context to the stop_ptr pointer. This will make all the physical addresses used by this context available to the read_ptr allocate pointer for future allocation.

This device allows representation of multiple contexts of constants data with N copies of the logical address space. It also allows the second context to be represented as the first set plus some new additional data by just storing the delta's. It allows memory to be efficiently used and when the constants updates are small it can store multiple context. However, if the updates are large, less contexts will be stored and potentially performance will be degraded. Although it will still perform as well as a ring could in this case.

5.4 Constant Store Indexing

In order to do constant store indexing, the sequencer must be loaded first with the indexes (that come from the GPRs). There are 144 wires from the exit of the SP to the sequencer (9 bits pointers x 16 vertexes/clock).

Exhibit 2036.doc/R400_Sequencer.doc 73119 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257779

ATI Ex. 2109 IPR2023-00922 Page 69 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 201546	GEN-CXXXXX-REVA	19 of 51

Since the data must pass thru the Shader pipe for the float to fixed conversion, there is a latency of 4 clocks (1 instruction) between the time the sequencer is loaded and the time one can index into the constant store. The assembly will look like this

MOVA R1.X,R2.X // Loads the sequencer with the content of R2.X, also copies the content of R2.X into R1.X NOP // latency of the float to fixed conversion

ADD R3,R4,C0[R2.X]// Uses the state from the sequencer to add R4 to C0[R2.X] into R3

Note that we don't really care about what is in the brackets because we use the state from the MOVA instruction. R2.X is just written again for the sake of simplicity and coherency.

The storage needed in the sequencer in order to support this feature is 2*64*9 bits = 1152 bits.

The address register is a signed integer, which ranges from -256 to 255.

5.5 Real Time Commands

The real time commands constants are written by the CP using the register mapped registers allocated for RT. It works is the same way than when dealing with regular constant loads BUT in this case the CP is not sending a logical address but rather a physical address and the reads are not passing thru the re-mapping table but are directly read from the memory. The boundary between the two zones is defined by the CONST_EO_RT control register. Similarly, for the fetch state, the boundary between the two zones is defined by the TSTATE_EO_RT control register.

5.6 Constant Waterfalling

In order to have a reasonable performance in the case of constant store indexing using the address register, we are going to have the possibility of using the physical memory port for read only. This way we can read 1 constant per clock and thus have a worst-case waterfall mode of 1 vertex per clock. There is a small synchronization issue related with this as we need for the SQ to make sure that the constants where actually written to memory (not only sent to the sequencer) before it can allow the first vector of pixels or vertices of the state to go thru the ALUs. To do so, the sequencer keeps 8 bits (one per render state) and sets the bits whenever the last render state is written to memory and clears the bit whenever a state is freed.

Exhibit 2036.docR400_Sequences.doc 73119 Bytes***
C ATI Confidential. Reference Copyright Notice on Cover Page

AMD1044_0257780

ATI Ex. 2109 IPR2023-00922 Page 70 of 326

	ORIGINATE D	ATE	EDIT DATE	R400 Sequencer Specification	PAGE
	24 September,	2001	4 September, 201546		20 of 51
			SECTON /rites are direct)	CONST_EO_RT	
		(Reads/W	AR SECTION rites are passing maping table)		
			Figure 9: The Cons	itant store	
6. Loop	ing and Bran	nches	ь Э		
			supported and will hav using a control program	e to be dealt with at the sequencer leven.	el. We plan on
6.1 The	controlling s	state.			
	ontroling state consis				
Boolean[255 Loop_count] Loop_Start[7 Loop_Step[7	7:0][31:0] 7:0][31:0]				
That is 256 I	Booleans and 32 loo	ops.			
We have a s	stack of 4 elements f	for neste	d calls of subroutines a	nd 4 loop counters to allow for nested lo	oops.
This state is	available on a per s	shader p	rogram basis.		
6.2 The	Control Flov	w Pro	aram		
	be able to code up a		-		
1: Loo 2: Exe					
Exhibit 2036.de	bcR400_Sequencer.doc 7311	19 Bytes***	© ATI Confidential. R	eference Copyright Notice on Cover I	⊃age © ***

ATI Ex. 2109 IPR2023-00922 Page 71 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
	24 September, 2001	4 September, 201546	GEN-CXXXXX-REVA	21 of 51	
3:	TexFetch			•	
4:	ALU				
5:	ALU				
6:	TexFetch				
7. End	Loop				

8: ALU Export

But realize that 3: may be dependent on 2: and 4: is almost certainly dependent on 2: and 3:. Without clausing, these dependencies need to be expressed in the Control Flow instructions. Additionally, without separate 'texture clauses' and 'ALU clauses' we need to know which instructions to dispatch to the Texture Unit and which to the ALU unit. This information will be encapsulated in the flow control instructions.

Each control flow instruction will contain 2 bits of information for each (non-control flow) instruction: a) ALU or Texture

b) Serialize Execution

(b) would force the thread to stop execution at this point (before the instruction is executed) and wait until all textures have been fetched. Given the allocation of reserved bits, this would mean that the count of an 'Exec' instruction would be limited to about 8 (non-control-flow) instructions. If more than this were needed, a second Exec (with the same conditions) would be issued.

Another function that relies upon 'clauses' is allocation and order of execution. We need to assure that pixels and vertices are exported in the correct order (even if not all execution is ordered) and that space in the output buffers are allocated in order. Additionally data can't be exported until space is allocated. A new control flow instruction:

Alloc <buffer select -- position, parameter, pixel or vertex memory. And the size required>.

would be created to mark where such allocation needs to be done. To assure allocation is done in order, the actual allocation for a given thread can not be performed unless the equivalent allocation for all previous threads is already completed. The implementation would also assure that execution of instruction(s) following the serialization due to the Alloc will occur in order -- at least until the next serialization or change from ALU to Texture. In most cases this will allow the exports to occur without any further synchronization. Only 'final' allocations or position allocations are guaranteed to be ordered. Because strict ordering is required for pixels, parameters and positions, this implies only a single alloc for these structures. Vertex exports to memory do not require ordering during allocation and so multiple 'allocs' may be done.

6.2.1 Control flow instructions table

Here is the revised control flow instruction set.

Note that whenever a field is marked as RESERVED, it is assumed that all the bits of the field are cleared (0).

NOP				
4744 43 420				
0000	Addressing	RESERVED		

This is a regular NOP.

	Execute							
47 44	43	40 34	33 16	1512	11 0			
0001	Addressing	RESERVED	Instructions type + serialize (9 instructions)	Count	Exec Address			
	Execute_End							
47 44	43	40 34	33 16	1512	11 0			
0010	Addressing	RESERVED	Instructions type + serialize (9	Count	Exec Address			

instructions)

Exhibit 2036 docR400_Sequences.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257782

ATI Ex. 2109 IPR2023-00922 Page 72 of 326

	AR	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
	<u>AU</u>	24 September, 2001	4 September, 201546		22 of 51
4			L Chickor USCI14		

Execute up to 9 instructions at the specified address in the instruction memory. The Instruction type field tells the sequencer the type of the instruction (LSB) (1 = Texture, 0 = ALU and whether to serialize or not the execution (MSB) (1 = Serialize, 0 = Non-Serialized). If Execute_End this is the last execution block of the shader program.

	Conditional_Execute									
47 44	43	42	41 34	3316	15 12	11 0				
0011	Addressing	Condition	Boolean	Instructions type + serialize (9	Count	Exec Address				
			address	instructions)						

	Conditional_Execute_End										
47 44	47 44 43 42		41 34	3316	15 12	11 0					
0100	Addressing	ssing Condition Boolean		Instructions type + serialize (9	Count	Exec Address					
			address	instructions)							

If the specified Boolean (8 bits can address 256 Booleans) meets the specified condition then execute the specified instructions (up to 9 instructions). If the condition is not met, we go on to the next control flow instruction. If Conditional_Execute_End and the condition is met, this is the last execution block of the shader program.

	Conditional_Execute_Predicates										
47 44	43	42	41 36	35 34	3316	1512	11 0	1000			
0101	Addressing	Condition	RESERVED	Predicate	Instructions	Count	Exec Address	200000			
				vector	type + serialize (9 instructions)			1000			

	Conditional_Execute_Predicates_End										
47 44	43	42	41 36	35 34	3316	1512	11 0				
0110	Addressing	Condition	RESERVED	Predicate	Instructions	Count	Exec Address				
				vector	type + serialize						
					(9 instructions)						

Check the AND/OR of all current predicate bits. If AND/OR matches the condition execute the specified number of instructions. We need to AND/OR this with the kill mask in order not to consider the pixels that aren't valid. If the condition is not met, we go on to the next control flow instruction. If Conditional_Execute_Predicates_End and the condition is met, this is the last execution block of the shader program.

	Conditional_Execute_Predicates_No_Stall										
47 44	43	42	41 36	35 34	3316	1512	11 0				
1101	Addressing	Condition	RESERVED	Predicate vector	Instructions type + serialize (9 instructions)	Count	Exec Address				

	Conditional_Execute_Predicates_No_Stall_End										
47 44	43	42	41 36	35 34	3316	1512	11 0				
1110	Addressing	Condition	RESERVED	Predicate vector	Instructions type + serialize (9 instructions)	Count	Exec Address				

Same as Conditionnal_Execute_Predicates but the SQ is not going to wait for the predicate vector to be updated. You can only set this in the compiler if you know that the predicate set is only a refinement of the current one (like a nested if) because the optimization would still work.

Loop_Start										
47 44	43	42 21	20 16	1513	12 0	18				
0111	Addressing	RESERVED	loop ID	RESERVED	Jump address	18				

Exhibit 2036.docR400_Sequences.doc 73119 Bytes**** 🕲 ATI Confidential. Reference Copyright Notice on Cover Page 🕲 ***

AMD1044_0257783

ATI Ex. 2109 IPR2023-00922 Page 73 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 201546	GEN-CXXXXX-REVA	23 of 51

Loop Start. Compares the loop iterator with the end value. If loop condition not met jump to the address. Forward jump only. Also computes the index value. The loop id must match between the start to end, and also indicates which control flow constants should be used with the loop.

	Loop_End										
47	43	42	<u>41 36</u>	3534	<u>33 22</u> 23	21	20 16	1513	12 0		
44		24			21						
1000	Addressing	RES	RESERVED	Predicate	RESERVED	Pred	loop ID	RESERVED	start		
	-	ERV		Vector	Predicate	break			address		
		EDC			break						
		ond									

Loop end. Increments the counter by one, compares the loop count with the end value. If loop condition met, continue, else, jump BACK to the start of the loop. If predicate break != 0, then compares predicate vector n (specified by predicate break number<u>Vector</u>) to condition. If all bits cleared then break the loop<u>meet condition then</u> break the loop.

The way this is described does not prevent nested loops, and the inclusion of the loop id make this easy to do.

Conditionnal_Call									
47 44	43	42	41 34	33 14	13	12 0			
1001	Addressing	Condition	Boolean address	RESERVED	Force Call	Jump address			

If the condition is met, jumps to the specified address and pushes the control flow program counter on the stack. If force call is set the condition is ignored and the call is made always.

	Return								
47		43	42 0	10.00					
10	10	Addressing	RESERVED	1000					

Pops the topmost address from the stack and jumps to that address. If nothing is on the stack, the program will just continue to the next instruction.

	Conditionnal_Jump						
47 44	43	42	41 34	33	32 14	13	12 0
1011	Addressing	Condition	Boolean	FW only	RESERVED	Force Jump	Jump address
			address	_			

If force jump is set the condition is ignored and the jump is made always. If FW only is set then only forward jumps are allowed.

	Allocate					
	47 44	43	4241	40 3	20	
Γ	1100	Debug	Buffer Select	RESERVED	Size	

Buffer Select takes a value of the following:

01 - position export (ordered export)

10 - parameter cache or pixel export (ordered export)

11 - pass thru (out of order exports).

Size field is only used to reserve space in the export buffer for pass thru exports. Valid values are 1 (1 line) thru 9 (9 lines). It should be determined by the compiler/assembler by taking max index used +1.

If debug is set this is a debug alloc (ignore if debug DB_ON register is set to off).

6.3 Implementation

Exhibit 2036 docR400_Sequencer.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257784

ATI Ex. 2109 IPR2023-00922 Page 74 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 201516		24 of 51

The envisioned implementation has a buffer that maintains the state of each thread. A thread lives in a given location in the buffer during its entire life, but the buffer has FIFO qualities in that threads leave in the order that they enter. Actually two buffers are maintained – one for Vertices and one for Pixels. The intended implementation would allow for:

16 entries for vertices 48 entries for pixels.

From each buffer, arbitration logic attempts to select 1 thread for the texture unit and 4-2 (interleaved) thread for the ALU unit. Once a thread is selected it is read out of the buffer, marked as invalid, and submitted to appropriate execution unit. It is returned to the buffer (at the same place) with its status updated once all possible sequential instructions have been executed. A switch from ALU to TEX or visa-versa or a Serialize_Execution modifier forces the thread to be returned to the buffer.

Each entry in the buffer will be stored across two physical pieces of memory - most bits will be stored in a 1 read port device. Only bits needed for thread arbitration will be stored in a highly multi-ported structure. The bits kept in the 1 read port device will be termed 'state'. The bits kept in the multi-read ported device will be termed 'state'.

'State Bits' needed include:

- 1. Control Flow Instruction Pointer (13 bits),
- 2. Execution Count Marker 4 bits),
- 3. Loop Iterators (4x9 bits),
- 4. Loop Counters (4x9 bits),
- 5. Call return pointers (4x13 bits),
- 6. Predicate Bits (64 bits),
- 7. Export ID (1 bit),
- 8. Parameter Cache base Ptr (7 bits),
- 9. GPR Base Ptr (8 bits),
- 10. Context Ptr (3 bits).
- 11. LOD corrections (6x16 bits)
- 12. Valid bits (64 bits)
- 13. RT (1 bit) Signifies that this thread is a Real Time thread. This bit must be sent to the Constant store state machine when reading it.

Absent from this list are 'Index' pointers. These are costly enough that I'm presuming that they are instead stored in the GPRs. The first seven fields above (Control Flow Ptr, Execution Count, Loop Counts, call return ptrs, Predicate bits, PC base ptr and export ID) are updated every time the thread is returned to the buffer based on how much progress has been mode on thread execution. GPR Base Ptr, Context Ptr and LOD corrections are unchanged throughout execution of the thread.

'Status Bits' needed include:

- Valid Thread
- Texture/ALU engine needed
- Texture Reads are outstanding
- Waiting on Texture Read to Complete
- Allocation Wait (2 bits)
- 00 No allocation needed
- 01 Position export allocation needed (ordered export)
- 10 Parameter or pixel export needed (ordered export)
- 11 pass thru (out of order export)
- Allocation Size (4 bits)
- Position Allocated
- Mem/Color Allocated
- First thread of a new context
- Event thread (NULL thread that needs to trickle down the pipe)

Exhibit 2036. doc R400_Sequences.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044 0257785

ATI Ex. 2109 IPR2023-00922 Page 75 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 201546	GEN-CXXXXX-REVA	25 of 51
I ast (1 bit)				

Pulse SX (1 bit)

All of the above fields from all of the entries go into the arbitration circuitry. The arbitration circuitry will select a winner for both the Texture Engine and for the ALU engine. There are actually two sets of arbitration -- one for pixels and one for vertices. A final selection is then done between the two. But the rest of this implementation summary only considers the 'first' level selection which is similar for both pixels and vertices.

Texture arbitration requires no allocation or ordering so it is purely based on selecting the 'oldest' thread that requires the Texture Engine.

ALU arbitration is a little more complicated. First, only threads where either of Texture_Reads_outstanding or Waiting_on_Texture_Read_to_Complete are '0' are considered. Then if Allocation_Wait is active, these threads are further filtered based on whether space is available. If the allocation is position allocation, then the thread is only considered if all 'older' threads have already done their position allocation (position allocated bits set). If the allocation is parameter or pixel allocation, then the thread is only considered if it is the oldest thread. Also a thread is not considered if it is a parameter or pixel or position allocation, has its First_thread_of_a_new_context bit set and would cause ALU interleaving with another thread performing the same parameter or pixel or position allocation. Finally the 'oldest' of the threads that pass through the above filters is selected. If the thread needed to allocate, then at this time the allocation is done, based on Allocation_Size. If a thread has its "last" bit set, then it is also removed from the buffer, never to return.

If I now redefine 'clauses' to mean 'how many times the thread is removed from the thread buffer for the purpose of exection by either the ALU or Texture engine', then the minimum number of clauses needed is 2 -- one to perform the allocation for exports (execution automatically halts after an 'Alloc' instruction) (but doesn't performs the actual allocation) and one for the actual ALU/export instructions. As the 'Alloc' instruction could be part of a texture clause (presumably the final instruction in such a clause), a thread could still execute in this minimal number of 2 clauses, even if it involved texture fetching.

The Texture_Reads_Outstanding bit must be updated by the sequencer, based on keeping track of how many Texture Clauses have been executed by a given thread that have not yet had there data returned. Any number above 0 results in this bit being set. We could consider forcing synchronization such that two texture clauses for a given thread may not be outstanding at any time (that would be my preference for simplicity reasons and because it would require only very little change in the texture pipe interface). This would allow the sequencer to set the bit on execution of the texture clause, and allow the texture unit to return a pointer to the thread buffer on completion that clears the bit.

6.4 Data dependant predicate instructions

Data dependant conditionals will be supported in the R400. The only way we plan to support those is by supporting three vector/scalar predicate operations of the form:

PRED_SETE_#__<u>PUSH</u> - similar to SETE except that the result is 'exported' to the sequencer. PRED_SETNE_<u>PUSH</u># - similar to SETNE except that the result is 'exported' to the sequencer. PRED_SETGT_<u>PUSH</u>#-- similar to SETGT except that the result is 'exported' to the sequencer PRED_SETGTE_<u>PUSH</u># - similar to SETGTE except that the result is 'exported' to the sequencer

For the scalar operations only we will also support the two following instructions: PRED_SETE0_#-SETE0

PR	ED_S	ETE1_	# <u>NE</u> – SETE
PR	ED S	ETGT	1
PR	ED S	ET IN	/
PR	ED S	ET PC	P
PR	ED_S	ET_CL	R
PR	ED S	ET RE	STORE

Details about actual implementation of these opcodes are in the shader pipe architectural spec.

Exhibit 2036. doc R400_Sequencer.do 73119 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257786

ATI Ex. 2109 IPR2023-00922 Page 76 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 201546		26 of 51

The export is a single bit - 1 or 0 that is sent using the same data path as the MOVA instruction. The sequencer will maintain 4.1 sets of 64 bits predicate vectors (in fact 28 sets because we interleave two programs but only 4.1 will be exposed) and use it to control the write masking. This predicate is not-maintained across clause boundaries. The # sign is used to specify which predicate set you want to use 0 thru 3.

Then we have two conditional execute bits. The first bit is a conditional execute "on" bit and the second bit tells us if we execute on 1 or 0. For example, the instruction:

P0_P0_ADD_# R0,R1,R2

Is only going to write the result of the ADD into those GPRs whose predicate bit is 0. Alternatively, P1_ADD_# would only write the results to the GPRs whose predicate bit is set. The use of the P0 or P1 without precharging the sequencer with a PRED instruction is undefined.

{Issue: do we have to have a NOP between PRED and the first instruction that uses a predicate?}

6.5 HW Detection of PV,PS

Because of the control program, the compiler cannot detect statically dependant instructions. In the case of nonmasked writes and subsequent reads the sequencer will insert uses of PV,PS as needed. This will be done by comparing the read address and the write address of consecutive instructions. For masked writes, the sequencer will insert NOPs wherever there is a dependant read/write.<u>detect wich channels to read from the GPRs and which ones</u> to read from the PV/PS.

The sequencer will also have to insert NOPs between PRED_SET and MOVA instructions and their uses.

6.6 Register file indexing

Because we can have loops in fetch clause, we need to be able to index into the register file in order to retrieve the data created in a fetch clause loop and use it into an ALU clause. The instruction will include the base address for register indexing and the instruction will contain these controls:

Bit7	Bit 6	
0	0	'absolute register'
0	1	'relative register'
1	0	'previous vector'
1	1	previous scalar

In the case of an absolute register we just take the address as is. In the case of a relative register read we take the base address and we add to it the loop_index and this becomes our new address that we give to the shader pipe.

The sequencer is going to keep a loop index computed as such:

Index = Loop_iterator*Loop_step + Loop_start.

We loop until loop_iterator = loop_count. Loop_step is a signed value [-128...127]. The computed index value is a 10 bit counter that is also signed. Its real range is [-256,256]. The tenth bit is only there so that we can provide an out of range value to the "indexing logic" so that it knows when the provided index is out of range and thus can make the necessary arrangements.

6.7 Debugging the Shaders

In order to be able to debug the pixel/vertex shaders efficiently, we provide 2 methods.

6.7.1 *Method 1: Debugging registers*

Current plans are to expose 2 debugging, or error notification, registers: 1. address register where the first error occurred 2. count of the number of errors

Exhibit 2036 docR400_Sequencer.doc 73119 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257787

ATI Ex. 2109 IPR2023-00922 Page 77 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 201516	GEN-CXXXXX-REVA	27 of 51

The sequencer will detect the following groups of errors:

count overflow

- constant indexing overflow

- register indexing overflow

Compiler recognizable errors:

- jump errors

relative jump address > size of the control flow program

- call stack call with stack full
 - return with stack empty

A jump error will always cause the program to break. In this case, a break means that a clause will halt execution, but allowing further clauses to be executed.

With all the other errors, program can continue to run, potentially to worst-case limits. The program will only break if the DB_PROB_BREAK register is set.

If indexing outside of the constant or the register range, causing an overflow error, the hardware is specified to return the value with an index of 0. This could be exploited to generate error tokens, by reserving and initializing the 0th register (or constant) for errors.

{ISSUE : Interrupt to the driver or not?}

6.7.2 Method 2: Exporting the values in the GPRs

1) The sequencer will have a debug active, count register and an address register for this mode.

Under the normal mode execution follows the normal course.

Under the debug mode it is assumed that the program is always exporting n debug vectors and that all other exports to the SX block (<u>but for</u> position, color, z, cet) will been turned off (changed into NOPs) by the sequencer (even if they occur before the address stated by the ADDR debug register).

7. Pixel Kill Mask

A vector of 64 bits is kept by the sequencer per group of pixels/vertices. Its purpose is to optimize the texture fetch requests and allow the shader pipe to kill pixels using the following instructions:

MASK_SETE MASK_SETNE MASK_SETGT MASK_SETGTE

8. Multipass vertex shaders (HOS)

Multipass vertex shaders are able to export from the 6 last clauses but to memory ONLY.

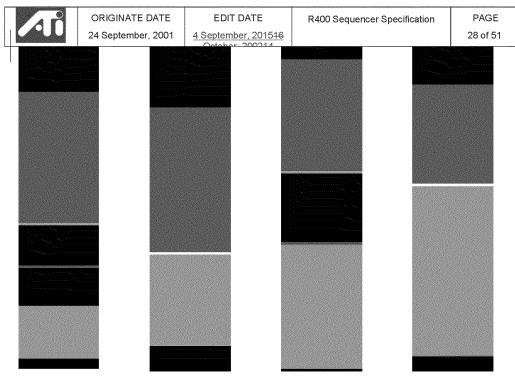
9. Register file allocation

The register file allocation for vertices and pixels can either be static or dynamic. In both cases, the register file in managed using two round robins (one for pixels and one for vertices). In the dynamic case the boundary between pixels and vertices is allowed to move, in the static case it is fixed to 128-VERTEX_REG_SIZE for vertices and PIXEL_REG_SIZE for pixels.

Exhibit 2036.docR400_Sequencer.doc 73119 Bytes***
ATI Confidential. Reference Copyright Notice on Cover Page

AMD1044_0257788

ATI Ex. 2109 IPR2023-00922 Page 78 of 326



Above is an example of how the algorithm works. Vertices come in from top to bottom; pixels come in from bottom to top. Vertices are in orange and pixels in green. The blue line is the tail of the vertices and the green line is the tail of the pixels. Thus anything between the two lines is shared. When pixels meets vertices the line turns white and the boundary is static until both vertices and pixels share the same "unallocated bubble". Then the boundary is allowed to move again. The numbering of the GPRs starts from the bottom of the picture at index 0 and goes up to the top at index 127.

10. Fetch Arbitration

The fetch arbitration logic chooses one of the n potentially pending fetch clauses to be executed. The choice is made by looking at the Vs and Ps reservation stations and picking the first one ready to execute. Once chosen, the clause state machine will send one 2x2 fetch per clock (or 4 fetches in one clock every 4 clocks) until all the fetch instructions of the clause are sent. This means that there cannot be any dependencies between two fetches of the same clause.

The arbitrator will not wait for the fetches to return prior to selecting another clause for execution. The fetch pipe will be able to handle up to X(?) in flight fetches and thus there can be a fair number of active clauses waiting for their fetch return data.

11. ALU Arbitration

ALU arbitration proceeds in almost the same way than fetch arbitration. The ALU arbitration logic chooses one of the n potentially pending ALU clauses to be executed. The choice is made by looking at the Vs and Ps reservation stations and picking the first one ready to execute. There are two ALU arbitrers, one for the even clocks and one for the odd clocks. For example, here is the sequencing of two interleaved ALU clauses (E and O stands for Even and Odd sets of 4 clocks):

Einst0 Oinst0 Einst1 Oinst1 Einst2 Oinst2 Einst0 Oinst3 Einst1 Oinst4 Einst2 Oinst0...

Exhibit 2036 docR400_Sequencer.doc 73119 Bytes***
C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257789

ATI Ex. 2109 IPR2023-00922 Page 79 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 201546	GEN-CXXXXX-REVA	29 of 51

Proceeding this way hides the latency of 8 clocks of the ALUs. Also note that the interleaving also occurs across clause boundaries.

12. Handling Stalls

When the output file is full, the sequencer prevents the ALU arbitration logic from selecting the last clause (this way nothing can exit the shader pipe until there is place in the output file. If the packet is a vertex packet and the position buffer is full (POS_FULL) then the sequencer also prevents a thread from entering an exporting clause. The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU arbitrer will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

13. Content of the reservation station FIFOs

The reservation FIFOs contain the state of the vector of pixels and vertices. We have two sets of those: one for pixels, and one for vertices. They contain 3 bits of Render State 7 bits for the base address of the GPRs, some bits for LOD correction and coverage mask information in order to fetch fetch for only valid pixels, the quad address.

14. The Output File

The output file is where pixels are put before they go to the RBs. The write BW to this store is 256 bits/clock. Just before this output file are staging registers with write BW 512 bits/clock and read BW 256 bits/clock. The staging registers are 4x128 (and there are 16 of those on the whole chip).

15. IJ Format

The IJ information sent by the PA is of this format on a per quad basis:

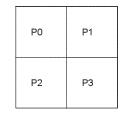
We have a vector of IJ's (one IJ per pixel at the centroid of the fragment or at the center of the pixel depending on the mode bit). All pixel's parameters are always interpolated at full 20x24 mantissa precision.

$$P0 = A + I(0) * (B - A) + J(0) * (C - A)$$

$$P1 = A + I(1) * (B - A) + J(1) * (C - A)$$

$$P2 = A + I(2) * (B - A) + J(2) * (C - A)$$

$$P3 = A + I(3) * (B - A) + J(3) * (C - A)$$



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

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

Total number of bits : 20*8 + 4*8 + 4*2 = 200.

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 +/- 1024.

15.1 Interpolation of constant attributes

Because of the floating point imprecision, we need to take special provisions if all the interpolated terms are the same or if two of the terms are the same.

Exhibit 2036 docR400_Sequencer.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257790

ATI Ex. 2109 IPR2023-00922 Page 80 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 201516		30 of 51

16. Staging Registers

In order for the reuse of the vertices to be 14, the sequencer will have to re-order the data sent IN ORDER by the VGT for it to be aligned with the parameter cache memory arrangement. Given the following group of vertices sent by the VGT:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 || 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 || 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 || 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

The sequencer will re-arrange them in this fashion:

0 1 2 3 16 17 18 19 32 33 34 35 48 49 50 51 || 4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 || 8 9 10 11 24 25 26 27 40 41 42 43 56 57 58 59 || 12 13 14 15 28 29 30 31 44 45 46 47 60 61 62 63

The || markers show the SP divisions. In the event a shader pipe is broken, the SQ is responsible to insert padding to account for the missing pipe. For example, if SP1 is broken, vertices 4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 will not be sent by the VGT to the SQ **AND** the SQ is responsible to "jump" over these vertices in order for no valid vertices to be sent to an invalid SP.

The most straightforward, *non-compressed* interface method would be to convert, in the VGT, the data to 32-bit floating point prior to transmission to the VSISRs. In this scenario, the data would be transmitted to (and stored in) the VSISRs in full 32-bit floating point. This method requires three 24-bit fixed-to-float converters in the VGT. Unfortunately, it also requires and additional 3,072 bits of storage across the VSISRs. This interface is illustrated in Figure 11Figure-11Figure-11. The area of the fixed-to-float converters and the VSISRs for this method is roughly estimated as 0.759sqmm using the R300 process. The gate count estimate is shown in Figure 10Figure-10.

Basis for 8-deep Latch Memory (fror	,			
8x24-bit	11631	μ^2	60.57813 µ² per l	
Area of 96x8-deep Latch Memory	46524	μ^2		
Area of 24-bit Fix-to-float Converter	$4712\mu^2$ per conv		erter	
Method 1	Block	Quantity	Area	
	F2F	3	14136	
	8x96 Latch	16_	744384	
			758520 µ ²	

Figure 10: Area Estimate for VGT to Shader Interface

Exhibit 2036 docR400_Sequencer.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257791

ATI Ex. 2109 IPR2023-00922 Page 81 of 326

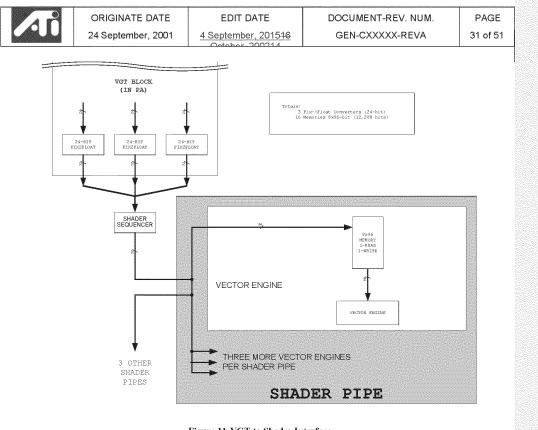


Figure 11:VGT to Shader Interface

17. The parameter cache

The parameter cache is where the vertex shaders export their data. It consists of 16 128x128 memories (1R/1W). The reuse engine will make it so that all vertexes of a given primitive will hit different memories. The allocation method for these memories is a simple round robin. The parameter cache pointers are mapped in the following way: 4MSBs are the memory number and the 7 LSBs are the address within this memory.

MEMORY NUMBER 4 bits	ADDRESS 7 bits

The PA generates the parameter cache addresses as the positions come from the SQ. All it needs to do is keep a Current_Location pointer (7 bits only) and as the positions comes increment the memory number. When the memory number field wraps around, the PA increments the Current_Location by VS_EXPORT_COUNT (a snooped register from the SQ). As an example, say the memories are all empty to begin with and the vertex shader is exporting 8 parameters per vertex (VS_EXPORT_COUNT = 8). The first position received is going to have the PC address 00000000000 the second one 00010000000, third one 0010000000 and so on up to 11110000000. Then the next position received (the 17th) is going to have the address 0000001000, the 18th 00010001000, the 19th 0010001000 and so on. The Current_location is NEVER reset BUT on chip resets. The only thing to be careful about is that if the SX doesn't send you a full group of positions (<64) then you need to fill the address space so that the next group starts correctly aligned (for example if you receive only 33 positions then you need to add 2*VS_EXPORT_COUNT to Current_Location and reset the memory count to 0 before the next vector begins).

Exhibit 2036. doc R400_Sequencer.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257792

ATI Ex. 2109 IPR2023-00922 Page 82 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 201516		32 of 51

17.1 Export restrictions

17.1.1 Pixel exports:

Pixels can export 1,2,3 or 4 color buffers to the SX(+z). The exports will be done in order. The PRED_OPTIMIZE function has to be turned of if the exports are done using interleaved predicated instructions. The exports will always be ordered to the SX.

17.1.2 Vertex exports:

Position or parameter caches can be exported in any order in the shader program. It is always better to export position as soon as possible. Position has to be exported in a single export block (no texture instructions can be placed between the exports). Parameter cache exports can be done in any order with texture instructions interleaved. The PRED_OPTIMIZE function has to be turned of if the exports are done using interleaved predicated instructions to the Parameter cache (see Arbitration restrictions for details). The exports will always be allocated in order to the SX.

17.1.3 Pass thru exports:

Pass thru exports have to be done in groups of the form:

Alloc 4 (<u>or 88 or 12</u>) Execute ALU(ADDR) ALU(DATA) ALU(DATA) ALU(DATA)...

They cannot have texture instructions interleaved in the export block. These exports are not guaranteed to be ordered.

Also, when doing a pass thru export, Position MUST be exported AFTER all pass thru exports. This position export is used to synchronize the chip when doing a transition from pass thru shader to regular shader and vice versa.

17.2 Arbitration restrictions

Here are the Sequencer arbitration restrictions:

- 1) Cannot execute a serialized thread if the corresponding texture pending bit is set
- 2) Cannot allocate position if any older thread has not allocated position
- 3) Cannot have more than 2 opened allocs of type : Memory, position and Color.
- 3)<u>4)</u>If last thread is marked as not valid AND marked as last and we are about to execute the second to oldest thread also marked last then:
 - a. Both threads must be from the same context (cannot allow a first thread)
 - b. Must turn off the predicate optimization for the second thread
- 4)5)Cannot execute a texture clause if texture reads are pending
- 5)6)Cannot execute last if texture pending (even if not serial)
- 7) Cannot allocate if not last or second to last for color exports.

18. Export Types

The export type (or the location where the data should be put) is specified using the destination address field in the ALU instruction. Here is a list of all possible export modes:

18.1 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
- 38:47 Empty48:52 5 debug export (interpret as normal memory export)

Exhibit 2036.doc.R400_Sequencer.doc 73119 Bytes**** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

- Formatted

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

AMD1044_0257793

ATI Ex. 2109 IPR2023-00922 Page 83 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE		
	24 September, 2001	4 September, 201546	GEN-CXXXXX-REVA	33 of 51		
60 - export addressing mode						

61 - Empty

62 - position

- 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.)

18.2 Pixel Shading

63

- 0 Color for buffer 0 (primary)
- 1 Color for buffer 1 2 - Color for buffer 2
 - Color for buffer 2
- 3 Color fe 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 Address33:37 5 exports for multipass pixel shaders.
- 38:47 Empty
- 48:52 5 debug exports (interpret as normal memory export)
- 60 export addressing mode
- Z for primary buffer (Z exported to 'alpha' component)
- 62:63 Empty

19. Special Interpolation modes

19.1 Real time commands

We are unable to use the parameter memory since there is no way for a command stream to write into it. Instead we need to add three 16x128 memories (one for each of three vertices x 16 interpolants). These will be mapped onto the register bus and written by type 0 packets, and output to the the parameter busses (the sequencer and/or PA need to be able to address the reatime parameter memory as well as the regular parameter store. For higher performance we should be able to view them as two banks of 16 and do double buffering allowing one to be loaded, while the other is rasterized with. Most overlay shaders will need 2 or 4 scalar coordinates, one option might be to restrict the memory to 16x64 or 32x64 allowing only two interpolated scalars per cycle, the only problem I see with this is, if we view support for 16 vector-4 interpolants important (true only if we map Microsoft's high priority stream to the realtime stream), then the PA/sequencer need to support a realtime-specific mode where we need to address 32 vectors of parameters instead of 16. This mode is triggered by the primitive type: REAL TIME. The actual memories are in the in the SX blocks. The parameter data memories are hooked on the RBBM bus and are loaded by the CP using register mapped memory.

19.2 Sprites/ XY screen coordinates/ FB information

XY screen coordinates may be needed in the shader program. This functionality is controlled by the param_gen_I0 register (in SQ) in conjunction with the SND_XY register (in SC) and the param_gen_pos. Also it is possible to send the faceness information (for OGL front/back special operations) to the shader using the same control register. Here is a list of all the modes and how they interact together:

The Data is going to be written in the register specified by the param_gen_pos register.

Param_Gen_I0 disable, snd_xy disable = No modification Param_Gen_I0 disable, snd_xy enable = No modification Param_Gen_I0 enable, snd_xy disable = Sign(faceness)garbage,(Sign Point)garbage,Sign(Line)s, t Param_Gen_I0 enable, snd_xy enable = Sign(faceness)screenX,(Sign Point)screenY,Sign(Line)s, t

Exhibit 2036 doc R400_Sequences.doc 73119 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257794

ATI Ex. 2109 IPR2023-00922 Page 84 of 326

A î	ORIGINATE DATE 24 September, 2001	EDIT DATE	R400 Sequencer Specification	PAGE 34 of 51				
	In other words,							

The generated vector is (X in RED, Y in GREEN, S in BLUE and T in ALPHA): X.Y.S.T

These values are always supposed to be positive and any shader use of them should use the ABS function (as their sign bits will now be used for flags).

SignX = BackFacing

SignY = Point Primitive SignS = Line Primitive

SignT = currently unused as a flag.

If !Point & !Line, then it is a Poly.

I would assume that one implementation which allows for generic texture lookup (using 3D maps) for poly stipple and AA for the driver would be

if(Y<0) { . R = 0.0 (Point) else if (S < 0)R = 1.0 (Line) } else { R = 2.0 (Poly)

}

19.3 Auto generated counters

In the cases we are dealing with multipass shaders, the sequencer is going to generate a vector count to be able to both use this count to write the 1^{st} pass data to memory and then use the count to retrieve the data on the 2^{nd} pass. The count is always generated in the same way but it is passed to the shader in a slightly different way depending on the shader type (pixel or vertex). This is toggled on and off using the GEN_INDEX_PIX/VTX register. The sequencer is going to keep two counters, one for pixels and one for vertices. Every time a full vector of vertices or pixels is written to the GPRs the counter is incremented. Every time a RST_PIX_COUNT or RST_VTX_COUNT events are received, the corresponding counter is reset. While there is only one count broadcast to the GPRs, the LSB are hardwired to specific values making the index different for all elements in the vector. Since the count must be different for all pixels/vertices and the 4 LSBs (16 positions) are hardwired to the corresponding shader unit the SQ has two choices:

1) Maintain a 19 bit counter that counts the vectors of 64. In this case the phase must be appended to the count before the count is broadcast to the SPs:

Counter (19 bits)	Phase (2 bits)	Hardwired (4 bits)

2) Maintain a 21 bits counter that counts sub-vectors of 16. In this case only the counter is sent to the Sps:

Counter (21 bits) Hardwired (4 bits)

19.3.1 Vertex shaders

In the case of vertex shaders, if GEN INDEX VTX is set, the data will be put into the x field of the third register (it means that the compiler must allocate 3 GPRs in all multipass vertex shader modes).

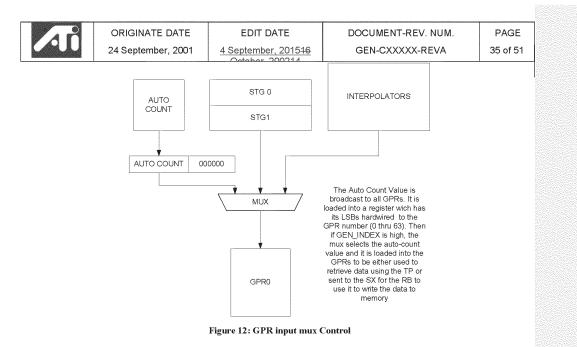
19.3.2 Pixel shaders

In the case of pixel shaders, if GEN_INDEX_PIX is set, the data will be put in the x field of the param_gen_pos+1 register.

Exhibit 2036.docR400_Sequencer.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044 0257795

ATI Ex. 2109 IPR2023-00922 Page 85 of 326



20. State management

Every clock, the sequencer will report to the CP the oldest states still in the pipe. These are the states of the programs as they enter the last ALU clause.

20.1 Parameter cache synchronization

In order for the sequencer not to begin a group of pixels before the associated group of vertices has finished, the sequencer will keep a 6 bit count per state (for a total of 8 counters). These counters are initialized to 0 and every time a vertex shader exports its data TO THE PARAMETER CACHE, the corresponding pointer is incremented. When the SC sends a new vector of pixels with the SC_SQ_new_vector bit asserted, the sequencer will first check if the count is greater than 0 before accepting the transmission (it will in fact accept the transmission but then lower its ready to receive). Then the sequencer waits for the count to go to one and decrements it. The sequencer can then issue the group of pixels to the interpolators. Every time the state changes, the new state counter is initialized to 0.

21. XY Address imports

The SC will be able to send the XY addresses to the GPRs. It does so by interleaving the writes of the IJs (to the IJ buffer) with XY writes (to the XY buffer). Then when writing the data to the GPRs, the sequencer is going to interpolate the IJ data or pass the XY data thru a Fix→float converter and expander and write the converted values to the GPRs. The Xys are currently SCREEN SPACE COORDINATES. The values in the XY buffers will wrap. See section 19.2 for details on how to control the interpolation in this mode.

21.1 Vertex indexes imports

In order to import vertex indexes, we have 16 8x96 staging registers. These are loaded one line at a time by the VGT block (96 bits). They are loaded in floating point format and can be transferred in 4 or 8 clocks to the GPRs.

22. Registers

Please see the auto-generated web pages for register definitions.

Exhibit 2036.docR400_Sequencer.doc 73119 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257796

ATI Ex. 2109 IPR2023-00922 Page 86 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 201516		36 of 51

23. Interfaces

23.1 External Interfaces

Whenever an x is used, it means that the bus is broadcast to all units of the same name. For example, if a bus is named SQ \rightarrow SPx it means that SQ is going to broadcast the same information to all SP instances.

23.2 SC to SP Interfaces

23.2.1 SC_SP#

There is one of these interfaces at front of each of the SP (buffer to stage pixel interpolators). This interface transmits the I,J data for pixel interpolation. For the entire system, two quads per clock are transferred to the 4 SPs, so each of these 4 interfaces transmits one half of a quad per clock. The interface below describes a half of a quad worth of data.

The actual data which is transferred per quad is

Ref Pix I => S4.20 Floating Point I value *4

Ref Pix J => S4.20 Floating Point J value *4

This equates to a total of 200 bits which transferred over 2 clocks and therefor needs an interface 100 bits wide

Additionally, X,Y data (12-bit unsigned fixed) is conditionally sent across this data bus over the same wires in an additional clock. The X,Y data is sent on the lower 24 bits of the data bus with faceness in the msb. Transfers across these interfaces are synchronized with the SC_SQ IJ Control Bus transfers.

The data transfer across each of these busses is controlled by a IJ_BUF_INUSE_COUNT in the SC. Each time the SC has sent a pixel vector's worth of data to the SPs, he will increment the IJ_BUF_INUSE_COUNT count. Prior to sending the next pixel vectors data, he will check to make sure the count is less than MAX_BUFER_MINUS_2, if not the SC will stall until the SQ returns a pipelined pulse to decrement the count when he has scheduled a buffer free. Note: We could/may optimize for the case of only sending only IJ to use all the buffers to pre-load more. Currently it is planned for the SP to hold 2 double buffers of I,J data and two buffers of X,Y data, so if either X,Y or Centers and Centroids are on, then the SC can send two Buffers.

In at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will increment buffer write address pointers correctly all the time. (We may revisit this for both the SX,SP,SQ and add a EndOfVector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of packet and multiple new vector signals should cause a partial vector and that this would not really be significant performance hit.)

Name	Bits	Description
SC SP# data	100	IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit)
		Type 0 or 1, First clock I, second clk J
		Field ULC URC LLC LRC
		Bits [63:39] [38:26] [25:13] [12:0]
		Format SE4M20 SE4M20 SE4M20 SE4M20
		Type 2
		Field Face X Y
		Bits [24] [23:12] [11:0]
		Format Bit Unsigned Unsigned
		Format Dit Onsigned Onsigned
SC SP# valid	1	Valid
SC SP# last guad data	1	This bit will be set on the last transfer of data per guad.
SC SP# type	2	0 -> Indicates centroids
		1 -> Indicates centers
		2 -> Indicates X.Y Data and faceness on data bus
		The SC shall look at state data to determine how many types to send for the

Exhibit 2036.docR400_Sequencer.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257797

ATI Ex. 2109 IPR2023-00922 Page 87 of 326

A ÎÎ	ORIGINATE DATE 24 September, 2001	EDIT DATE <u>4 September, 201546</u> Optober, 200214	DOCUMENT-REV. NUM. GEN-CXXXXX-REVA	PAGE 37 of 51
	i	nterpolation process.		

The # is included for clarity in the spec and will be replaced with a prefix of u#_ in the verilog module statement for the SC and the SP block will have neither because the instantiation will insert the prefix.

23.2.2 SC_SQ

This is the control information sent to the sequencer in order to synchronize and control the interpolation and/or loading data into the GPRs needed to execute a shader program on the sent pixels. This data will be sent over two clocks per transfer with 1 to 16 transfers. Therefore the bus (approx 108 bits) could be folded in half to approx 54 bits.

Name	Bits	Description
Name SC_SQ_data	Bits 46	Control Data sent to the SQ 1 clk transfers Event - valid data consist of event_id and state_id. Instruct SQ to post an event vector to send state id and event_id through request fifo and onto the reservation stations making sure state id and/or event_id gets back to the CP. Events only follow end of packets so no pixel vectors will be in progress. Empty Quad Mask – Transfer Control data consisting of pc_dealloc or new_vector. Receipt of this is to transfer pc_dealloc or new_vector without any valid quad data. New vector will always be posted to request fifo and pc_dealloc will be attached to any pixel vector outstanding or posted in request fifo if no valid quad outstanding. 2 clk transfers Quad Data Valid – Sending quad data with or without new_vector or pc_dealloc. New vector will be posted to request fifo with or without a pixel vector and pc_dealloc will be posted with a pixel vector unless none is in progress. In this case the pc_dealloc will be posted in the request queue. Filler quads will be transferred with
SC_SQ_valid	1	Filler quads will be transferred with The Quad mask set but the pixel corresponding pixel mask set to zero. SC sending valid data, 2 nd clk could be all zeroes

SC_SQ_data – first clock and second clock transfers are shown in the table below.

Name	BitField	Bits	Description
1 st Clock Transfer			
SC_SQ_event	0	1	This transfer is a 1 clock event vector Force quad_mask =
			new_vector=pc_dealloc=0
SC_SQ_event_id	[5:1]	4	This field identifies the event 0 => denotes an End Of State Event 1

Exhibit 2036.docR400_Sequences.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257798

ATI Ex. 2109 IPR2023-00922 Page 88 of 326

<u> </u>							
		ORIGINATE [DATE	E	DIT DATE	R400 Sequencer Specification	PAGE
	CAUL	24 September	2001		ember, 2015 16		38 of 51
Ч					=> TBD		L
	SC SQ state	id	[8:6]	3	State/constant	acintar (6*2+2)	
	SC SQ pc de		[11:9]	3		(en for the Parameter Cache	
			12	1		ait for Vertex shader done count > 0 a	nd offer
	SC_SQ_new_	vector	12			Pixel Vector the SQ will decrement the	
	SC SQ quad	mask	[16:13]	4	Quad Write ma	sk left to right SP0 => SP3	
	SC SQ end o		17	1	End Of the prim	itive	
	SC_SQ_pix_m		[33:18]	16	Valid bits for all	pixels SP0=>SP3 (UL,UR,LL,LR)	
	SC SQ provo	k vtx	[35:34]	2	Provoking verte	x for flat shading	
	SC SQ lod c	orrect 0	[44:36]	9	LOD correction	for quad 0 (SP0) (9 bits per quad)	
	SC SQ lod c	orrect 1	[53:45]	9		for guad 1 (SP1) (9 bits per guad)	
	2nd Clock Tra	nsfer					
	SC_SQ_lod_c	orrect_2	[8:0]	9	LOD correction	for quad 2 (SP2) (9 bits per quad)	
	SC_SQ_lod_c	orrect_3	[17:9]	9	LOD correction	for quad 3 (SP3) (9 bits per quad)	
	SC_SQ_pc_pt	rO	[28:18]	11	Parameter Cac	he pointer for vertex 0	
	SC_SQ_pc_pt	r1	[39:29]	11	Parameter Cacl	he pointer for vertex 1	
	SC_SQ_pc_pt	r2	[50:40]	11	Parameter Cacl	he pointer for vertex 2	
	SC_SQ_prim_	type	[53:51]	3	Stippled line an	d Real time command need to load te	c cords from
					alternate buffer		
					000: Sprite (poi	nt)	
					001: Line		
					010: Tri_rect		
					100: Realtime S		
					101: Realtime L		
					110: Realtime T	ri_rect	

Name	Bits	Description
SQ_SC_free_buff	1	Pipelined bit that instructs SC to decrement count of buffers in use.
SQ_SC_dec_cntr_cnt	1	Pipelined bit that instructs SC to decrement count of new vector and/or event
_		sent to prevent SC from overflowing SQ interpolator/Reservation request fifo.

The scan converter will submit a partial vector whenever:

1.) He gets a primitive marked with an end of packet signal.

2.) A current pixel vector is being assembled with at least one or more valid quads and the vector has been marked for deallocate when a primitive marked new_vector arrives. The Scan Converter will submit a partial vector (up to 16quads with zero pixel mask to fill out the vector) prior to submitting the new_vector marker\primitive.

(This will prevent a hang which can be demonstrated when all primitives in a packet three vectors are culled except for a one quad primitive that gets marked pc_dealloc (vertices maximum size). In this case two new_vectors are submitted and processed, but then one valid quad with the pc_dealloc creates a vector and then the new would wait for another vertex vector to be processed, but the one being waited for could never export until the pc_dealloc signal made it through and thus the hang.)

Exhibit 2036.doc/R400_Sequesn.cer.doc 73119 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257799

ATI Ex. 2109 IPR2023-00922 Page 89 of 326

^ 1î	ORIGINATE DATE 24 September, 2001	EDIT DATE <u>4 September, 201546</u> October, 200214	DOCUMENT-REV. NUM. GEN-CXXXXX-REVA	PAGE 39 of 51	
23.2.3 SQ to SX(SP): Interpolator hus					

3.2.3 SQ to SX(SP): Interpolator bus

Name	Direction	Bits	Description
SQ SPx interp flat vtx	SQ→SPx	2	Provoking vertex for flat shading
SQ_SPx_interp_flat_gouraud	SQ→SPx	1	Flat or gouraud shading
SQ_SPx_interp_cyl_wrap	SQ→SPx	4	Wich channel needs to be cylindrical wrapped
SQ_SPx_interp_param_gen	SQ→SPx	1	Generate Parameter
SQ_SPx_interp_prim_type	SQ→SPx	2	Bits [1:0] of primitive type sent by SC
SQ_SPx_interp_buff_swap	SQ→SPx	1	Swapp IJ buffers
SQ_SPx_interp_IJ_line	SQ→SPx	2	IJ line number
SQ_SPx_interp_mode	SQ→SPx	1	Center/Centroid sampling
SQ_SXx_pc_ptr0	SQ→SXx	11	Parameter Cache Pointer
SQ_SXx_pc_ptr1	SQ→SXx	11	Parameter Cache Pointer
SQ_SXx_pc_ptr2	SQ→SXx	11	Parameter Cache Pointer
SQ_SXx_rt_sel	SQ→SXx	1	Selects between RT and Normal data (Bit 2 of prim type)
SQ_SX0_pc_wr_en	SQ→SX0	8	Write enable for the PC memories
SQ_SX1_pc_wr_en	SQ→SX1	8	Write enable for the PC memories
SQ_SXx_pc_wr_addr	SQ→SXx	7	Write address for the PCs
SQ_SXx_pc_channel_mask	SQ→SXx	4	Channel mask
SQ_SXx_pc_ptr_valid	SQ→SXx	1	Read pointers are valid.
SQ_SPx_interp_valid	SQ→SPx	1	Interpolation control valid

23.2.4 SQ to SP: Staging Register Data

This is a broadcast bus that sends the VSISR information to the staging registers of the shader pipes.

Name	Direction	Bits	Description
SQ_SPx_vsr_data	SQ→SPx	96	Pointers of indexes or HOS surface information
SQ_SPx_vsr_double	SQ→SPx	1	0: Normal 96 bits per vert 1: double 192 bits per vert
SQ_SP0_vsr_valid	SQ→SP0	1	Data is valid
SQ_SP1_vsr_valid	SQ→SP1	1	Data is valid
SQ_SP2_vsr_valid	SQ→SP2	1	Data is valid
SQ_SP3_vsr_valid	SQ→SP3	1	Data is valid
SQ_SPx_vsr_read	SQ→SPx	1	Increment the read pointers

23.2.5 VGT to SQ : Vertex interface

23.2.5.1 Interface Signal Table

The area difference between the two methods is not sufficient to warrant complicating the interface or the state requirements of the VSISRs. <u>Therefore, the POR for this interface is that the VGT will transmit the data to the</u> <u>VSISRs (via the Shader Sequencer) in full, 32-bit floating-point format.</u> The VGT can transmit up to six 32-bit floating-point values to each VSISR where four or more values require two transmission clocks. The data bus is 96 bits wide. In the case where an event is sent the 5 LSBs of VGT_SQ_vsisr_data contain the eventID.

Name	Bits	Description				
VGT_SQ_vsisr_data	96	Pointers of indexes or HOS surface information				
VGT_SQ_event	1	VGT is sending an event				
VGT_SQ_vsisr_continued	1	0: Normal 96 bits per vert 1: double 192 bits per vert				
VGT_SQ_end_of_vtx_vect	1	Indicates the last VSISR data set for the current process vector (for double vector data, "end_of_vector" is set on the first vector)				
VGT_SQ_indx_valid	1	Vsisr data is valid				
VGT_SQ_state	3	Render State (6*3+3 for constants). This signal is guaranteed to be correct when "VGT_SQ_vgt_end_of_vector" is high.				
VGT_SQ_send	1	Data on the VGT_SQ is valid receive (see write-up for standard R400 SEND/RTR interface handshaking)				
SQ_VGT_rtr	1	Ready to receive (see write-up for standard R400 SEND/RTR interface handshaking)				

23.2.5.2 Interface Diagrams

Exhibit 2036.docR400_Sequences.doc 73119 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257800

ATI Ex. 2109 IPR2023-00922 Page 90 of 326

PROTECTIVE ORDER MATERIAL

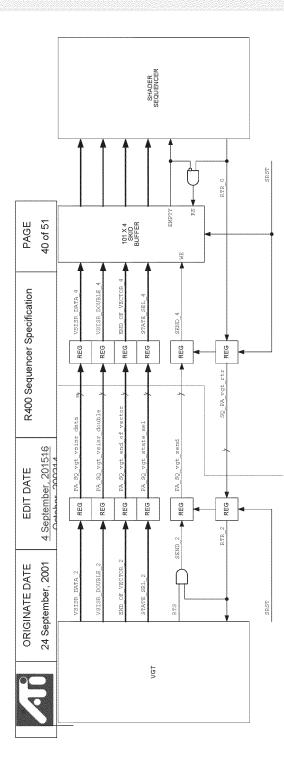
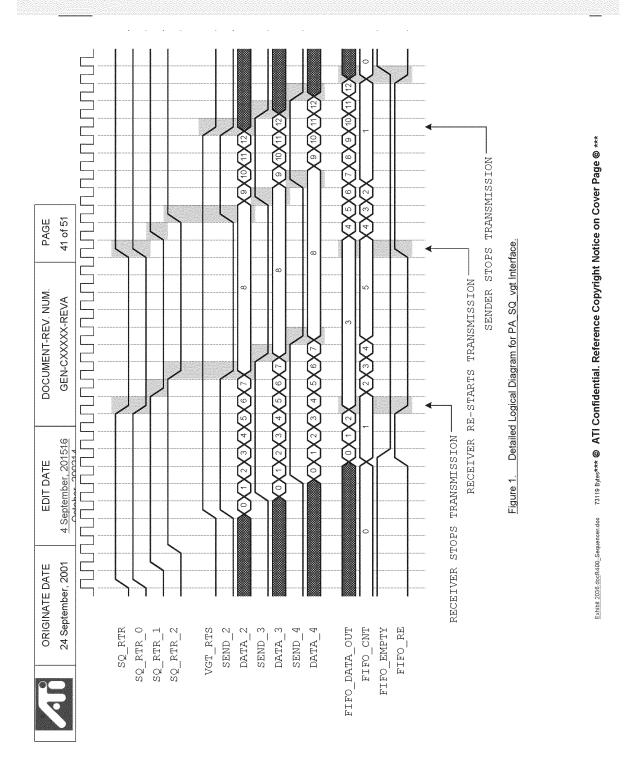


Exhibit 2005 docR400_Sequences doo 73119 Bytes*** @ ATI Confidential. Reference Copyright Notice on Cover Page @ ***

AMD1044_0257801

ATI Ex. 2109 IPR2023-00922 Page 91 of 326

PROTECTIVE ORDER MATERIAL



AMD1044_0257802

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
	24 September, 2001	4 September, 201516		42 of 51

23.2.6 SQ to SX: Control bus

Name	Direction	Bits	Description
SQ_SXx_exp_type	SQ→SXx	2	00: Pixel without z (1 to 4 buffers) 01: Pixel with z (1 to 4 buffers) 10: Position (1 or 2 results) 11: Pass thru (4,8 or 12 results aligned)
SQ_SXx_exp_number	SQ→SXx	2	Number of locations needed in the export buffer (encoding depends on the type see bellow).
SQ_SXx_exp_alu_id	SQ→SXx	1	ALU ID
SQ_SXx_exp_valid	SQ→SXx	1	Valid bit
SQ_SXx_exp_state	SQ→SXx	3	State Context
SQ_SXx_free_done	SQ→SXx	1	Pulse that indicates that the previous export is finished from the point of view of the SP. This does not necessarily mean that the data has been transferred to RB or PA, or that the space in export buffer for that particular vector thread has been freed up.
SQ_SXx_free_alu_id	SQ→SXx	1	ALU ID

Depending on the type the number of export location changes:

- Type 00 : Pixels without Z
 - 00 = 1 buffer
 - 01 = 2 buffers
 - 10 = 3 buffers
 - 11 = 4 buffer

۰

.

- Type 01: Pixels with Z
 - 00 = 2 Buffers (color + Z)
 - 01 = 3 buffers (2 color + Z)
 - 10 = 4 buffers (3 color + Z)
 - 11 = 5 buffers (4 color + Z)
- Type 10 : Position export
 - 00 = 1 position
 - 01 = 2 positions
 - 1X = Undefined
 - Type 11: Pass Thru
 - 00 = 4 buffers
 - 01 = 8 buffers
 - 10 = 12 buffers
 - 11 = Undefined

Below the thick black line is the end of transfer packet that tells the SX that a given export is finished. The report packet will always arrive either before or at the same time than the next export to the same ALU id.

23.2.7 SX to SQ : Output file control

Name	Direction	Bits	Description
SXx_SQ_exp_count_rdy	SXx→SQ	1	Raised by SX0 to indicate that the following two fields reflect the result of the most recent export
SXx_SQ_exp_pos_avail	SXx→SQ	2	Specifies whether there is room for another position. 00 : 0 buffers ready 01 : 1 buffer ready 10 : 2 or more buffers ready
SXx_SQ_exp_buf_avail	SXx→SQ	7	Specifies the space available in the output buffers. 0: buffers are full 1: 2K-bits available (32-bits for each of the 64

Exhibit 2036 docR400_Sequencer.doc 73119 Bytes***
C ATI Confidential. Reference Copyright Notice on Cover Page

AMD1044_0257803

ATI Ex. 2109 IPR2023-00922 Page 93 of 326

A ÎÎ	24 September, 2001 4 Septer		T DATE	DOCUMENT-REV. NUM. GEN-CXXXXX-REVA	PAGE 43 of 51
			 64: 128 64 pixe	n a clause) 8K-bits available (16 128-bit entries fo ≳ls) ∵ RESERVED	r each of

23.2.8 SQ to TP: Control bus

Once every clock, the fetch unit sends to the sequencer on which RS line it is now working and if the data in the GPRs is ready or not. This way the sequencer can update the fetch valid bits flags for the reservation station. The sequencer also provides the instruction and constants for the fetch to execute and the address in the register file where to write the fetch return data.

Name	Direction	Bits	Description
TPx_SQ_data_rdy	$TPx \rightarrow SQ$	1	Data ready
TPx_SQ_rs_line_num	$TPx \rightarrow SQ$	6	Line number in the Reservation station
TPx_SQ_type	TPx→ SQ	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_TPx_send	SQ→TPx	1	Sending valid data
SQ_TPx_const	SQ→TPx	48	Fetch state sent over 4 clocks (192 bits total)
SQ_TPx_instr	SQ→TPx	24	Fetch instruction sent over 4 clocks
SQ_TPx_end_of_group	SQ→TPx	1	Last instruction of the group
SQ_TPx_Type	SQ→TPx	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_TPx_gpr_phase	SQ→TPx	2	Write phase signal
SQ_TP0_lod_correct	SQ→TP0	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP0_pix_mask	SQ→TP0	4	Pixel mask 1 bit per pixel
SQ_TP1_lod_correct	SQ→TP1	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP1_pix_mask	SQ→TP1	4	Pixel mask 1 bit per pixel
SQ_TP2_lod_correct	SQ→TP2	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP2_pix_mask	SQ→TP2	4	Pixel mask 1 bit per pixel
SQ_TP3_lod_correct	SQ→TP3	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP3_pix_mask	SQ→TP3	4	Pixel mask 1 bit per pixel
SQ_TPx_rs_line_num	SQ→TPx	6	Line number in the Reservation station
SQ_TPx_write_gpr_index	SQ->TPx	7	Index into Register file for write of returned Fetch Data
SQ_TPx_ctx_id	SQ→TPx	3	The state context ID (needed for multisample resolves)

23.2.9 TP to SQ: Texture stall

The TP sends this signal to the SQ and the SPs when its input buffer is full.

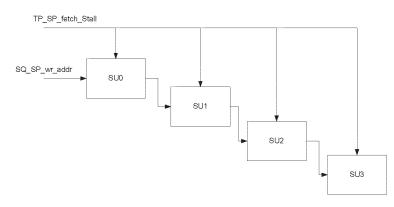


Exhibit 2036.docR400_Sequences.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257804

ATI Ex. 2109 IPR2023-00922 Page 94 of 326

	A Î	ORIGINATE DATE 24 September, 2001	EDIT DA <u>4 September,</u>	20154	6	R400 Sequencer Specification	PAGE 44 of 51
Name Direction TP_SQ_fetch_stall TP→ SQ					cription to send more texture request if asserted	d	

23.2.10 SQ to SP: Texture stall

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

23.2.11 SQ to SP: GPR and auto counter

Name	Direction	Bits	Description
SQ_SPx_gpr_wr_addr	SQ→SPx	7	Write address
SQ_SPx_gpr_rd_addr	SQ→SPx	7	Read address
SQ_SPx_gpr_rd_en	SQ→SPx	1	Read Enable
SQ_SP0_gpr_wr_en	SQ→SPx	4	Write Enable for the GPRs of SP0
SQ_SP1_gpr_wr_en	SQ→SPx	4	Write Enable for the GPRs of SP1
SQ_SP2_gpr_wr_en	SQ→SPx	4	Write Enable for the GPRs of SP2
SQ_SP3_gpr_wr_en	SQ→SPx	4	Write Enable for the GPRs of SP3
SQ_SPx_gpr_phase	SQ→SPx	2	The phase mux (arbitrates between inputs, ALU SRC
			reads and writes)
SQ_SPx_channel_mask	SQ→SPx	4	The channel mask
SQ_SPx_gpr_input_sel	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_auto_count	SQ→SPx	21	Auto count generated by the SQ, common for all shader pipes

Exhibit 2036.docR400_Sequencer.doc 73119 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257805

ATI Ex. 2109 IPR2023-00922 Page 95 of 326

	ORIGINATE DA	TE	EDIT I	DATE	DOC	UMENT-REV. NUM.	PAGE
	24 September, 2	001	4 September		<u>e</u> Ge	N-CXXXXX-REVA	45 of 51
3.2.12 SQ	to SPx: Insti	ructio					
Name		Direct			Description		
SQ_SPx_instr_	start	SQ→S SQ→S			nstruction start Fransferred over	4 avalaa	
SQ_SP_instr					D: SRC A Negate	e Argument Modifier 0:0 rgument Modifier 1:1 e 9:2 15:10	be 11 for all
						15:10	be 11 for all
						15:10	oe 11 for all
					3: Vector Opcod Scalar Opcod Vector Clamp Scalar Clamp Vector Write N Scalar Write N Unused	e 10:5 11:11 12:12 Mask 16:13 Mask 20:17 23:21	
SQ_SP0_pred	_override	SQ→S	\$P0			nel RGBA field (enables the nly pay attention to the 11 se	
SQ_SP1_pred		SQ→S			ogic, if not set o 1: Use GPR	nel RGBA field (enables the nly pay attention to the 11 se	eting).
SQ_SP2_pred	-	SQ→S			ogic, if not set o 1: Use GPR	nel RGBA field (enables the nly pay attention to the 11 se	eting).
SQ_SP3_pred_	-	SQ→S			ogic, if not set o I: Use GPR	nel RGBA field (enables the nly pay attention to the 11 se	
SQ_SPx_exp	id	SQ→S	SPx	1	GPR ID		

AMD1044_0257806

ATI Ex. 2109 IPR2023-00922 Page 96 of 326

	AI	ORIGINATE DATE EDIT DA 24 September, 2001 4 September, 2001 SQ_SPx_exporting SQ → SPx		201516	R400 Sequencer Specification	PAGE 46 of 51	
	SQ_SPx_e	xporting	SQ-	→SPx	1	0: Not Exporting	
1						1: Exporting	
	SQ_SPx_stall SQ_		→SPx 1 S		Stall signal		

23.2.13 SQ to SX: write mask interface (must be aligned with the SP data)

Name	Direction	Bits	Description
SQ_SX0_write_mask	SQ→SP0	8	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. This is for the data coming of SP0 and SP2.
SQ_SX1_ write_mask	SQ→SP1	8	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. This is for the data coming of SP1 and SP3.

23.2.14 SP to SQ: Constant address load/ Predicate Set/Kill set

Name	Direction	Bits	Description
SP0_SQ_const_addr	SP0→SQ	36	Constant address load / predicate vector load (4 bits only)/
			Kill vector load (4 bits only) to the sequencer
SP0_SQ_valid	SP0→SQ	1	Data valid
SP1_SQ_const_addr	SP1→SQ	36	Constant address load / predicate vector load (4 bits only)/
			Kill vector load (4 bits only) to the sequencer
SP1_SQ_valid	SP1→SQ	1	Data valid
SP2_SQ_const_addr	SP2→SQ	36	Constant address load / predicate vector load (4 bits only)/
			Kill vector load (4 bits only) to the sequencer
SP2_SQ_valid	SP2→SQ	1	Data valid
SP3_SQ_const_addr	SP3→SQ	36	Constant address load / predicate vector load (4 bits only)/
			Kill vector load (4 bits only) to the sequencer
SP3_SQ_valid	SP3→SQ	1	Data valid
SP0_SQ_data_type	SP→SQ	2	Data Type
			0: Constant Load
			1: Predicate Set
			2: Kill vector load

Because of the sharing of the bus none of the MOVA, PREDSET or KILL instructions may be coissued.

23.2.15 SQ to SPx: constant broadcast

Name	Direction	Bits	Description	100
SQ_SPx_const	SQ→SPx	128	Constant broadcast	

23.2.16 SQ to CP: RBBM bus

Name	Direction	Bits	Description
SQ_RBB_rs	SQ→CP	1	Read Strobe
SQ_RBB_rd	SQ→CP	32	Read Data
SQ_RBBM_nrtrtr	SQ→CP	1	Optional
SQ_RBBM_rtr	SQ→CP	1	Real-Time (Optional)

23.2.17 CP to SQ: RBBM bus

Name	Direction	Bits	Description	
rbbm_we	CP→SQ	1	Write Enable	
rbbm_a	CP→SQ	15	Address Upper Extent is TBD (16:2)	
rbbm_wd	CP→SQ	32	Data	
rbbm_be	CP→SQ	4	Byte Enables	
rbbm_re	CP→SQ	1	Read Enable	
rbb_rs0	CP→SQ	1	Read Return Strobe 0	

Exhibit 2036.docR400_Sequences.doc 73119 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257807

ATI Ex. 2109 IPR2023-00922 Page 97 of 326

			EDIT DATE			DOCUMENT-REV. NUM. GEN-CXXXXX-REVA	PAGE 47 of 51
rbb_rs1		CP→SQ		1	Read F	Return Strobe 1	
rbb_rd0		CP→SQ		32	Read I	Data 0	
rbb_rd1		CP→SQ		32	Read I	Data 0	
RBBM_S	Q_soft_reset	CP→SQ		1	Soft R	eset	

23.2.18 SQ to CP: State report

Name	Direction	Bits	Description	
SQ_CP_vs_event	SQ→CP	1	Vertex Shader Event	
SQ_CP_vs_eventid	SQ→CP	5	Vertex Shader Event ID	
SQ_CP_ps_event	SQ→CP	1	Pixel Shader Event	
SQ_CP_ps_eventid	SQ→CP	5	Pixel Shader Event ID	

23.3 Example of control flow program execution

We now provide some examples of execution to better illustrate the new design.

Given the program:

Alu 0 Alu 1 Tex 0 Tex 1 Alu 3 Serial Alu 4 Tex 2 Alu 5 Alu 6 Serial Tex 3 Alu 7 Alloc Position 1 buffer Alu 8 Export Tex 4 Alloc Parameter 3 buffers Alu 9 Export 0 Tex 5 Alu 10 Serial Export 2 Alu 11 Export 1 End

Would be converted into the following CF instructions:

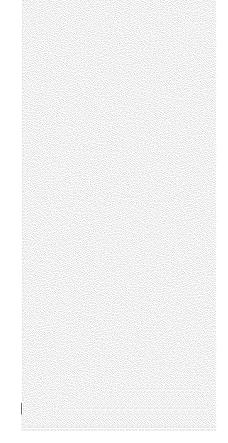
Execute 0 Alu 0 Alu 0 Tex 0 Tex 1 Alu 0 Alu 0 Tex 0 Alu 1 Alu 0 Tex Execute 0 Alu Alloc Position 1 Execute 0 Alu 0 Tex Alloc Param 3 Execute end 0 Alu 0 Tex 1 Alu 0 Alu

And the execution of this program would look like this:

Put thread in Vertex RS:

Control Flow Instruction Pointer (12 bits), (CFP) Execution Count Marker (3 or 4 bits), (ECM) Loop Iterators (4x9 bits), (LI) Call return pointers (4x12 bits), (CRP) Predicate Bits(4x64 bits), (PB) Export ID (1 bit), (EXID) GPR Base Ptr (8 bits), (GPR)

Exhibit 2036. doc R400_Sequencer.do 73119 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***



AMD1044_0257808

ATI Ex. 2109 IPR2023-00922 Page 98 of 326

	ORIGIN	ATE DATE		EDIT DA	TE	R40	0 Sequend	er Speci	fication	PAGE	
	24 Septe	ember, 2001		tember,						48 of 51	
Context	Export Base Ptr (7 bits), (EB) Context Ptr (3 bits).(CPTR) LOD correction bits (16x6 bits) (LOD)										
State Bits											
CFP I	CFP ECM LI CRP PB EXID GPR EB CPTR LOD										
0 (0 C) 0		0	0	(C	0	0	0	
Texture/ Texture Waiting d Allocatio 00 – 01 – 10 – Allocatio Position First three	Valid Thread (VALID) Texture/ALU engine needed (TYPE) Texture Reads are outstanding (PENDING) Waiting on Texture Read to Complete (SERIAL) Allocation Wait (2 bits) (ALLOC) 00 – No allocation needed 01 – Position export allocation needed (ordered export) 10 – Parameter or pixel export needed (ordered export) 11 – pass thru (out of order export) Allocation Size (4 bits) (SIZE) Position Allocated (POS_ALLOC) First thread of a new context (FIRST) Last (1 bit), (LAST)										
	TYPE	PENDING			ALLOC	SIZE	POS_AL	LOC	FIRST	LAST	
1	ALU	0	0		0	0	0		1	0	
Execute 0 It execut	e thread is pic Alu 0 Alu tes the first t urned to the F	u O Tex O ' wo ALU inst	Tex 1 A	lu O A	lu O Tex	0 Al	u 1 Alu	0 Tex	uest change	e. Here is the	

State Bit	s									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD	NORMA
0	2	0	0	0	0	0	0	0	0	

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	0	0	0	0	0	1	0

Then when the texture pipe frees up, the arbiter picks up the thread to issue the texture reads. The thread comes back in this state:

State Bits	State Bits												
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD				
0	4	0	0	0	0	0	0	0	0				

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	1	0	0	0	1	0

Because of the serial bit the arbiter must wait for the texture to return and clear the PENDING bit before it can pick the thread up. Lets say that the texture reads are complete, then the arbiter picks up the thread and returns it in this state:

Exhibit 2036.docR400_Sequences.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257809

ATI Ex. 2109 IPR2023-00922 Page 99 of 326

Ai		GINATE		EDIT [4 September October	er, 201546	DOCL	PAGE 49 of 51		
State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
0	6	S 0 0		0	0	0	0	0	0

Status Bit	s							
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	0	0	0	0	0	1	0

Again the TP frees up, the arbiter picks up the thread and executes. It returns in this state:

State Bits	State Bits												
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD				
0	7	0	0	0	0	0	0	0	0				

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	0	0	0	1	0

Now, even if the texture has not returned we can still pick up the thread for ALU execution because the serial bit is not set. The thread will however come back to the RS for the second ALU instruction because it has the serial bit set.

State Bits												
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD			
0	8	0	0	0	0	0	0	0	0			
0	0	0	0	0	0	0	0	0	0			

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	1	0	0	0	1	0

As soon as the TP clears the pending bit the thread is picked up and returns:

State Bits												
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD			
0	9	0	0	0	0	0	0	0	0			

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	0	0	0	0	0	1	0

Picked up by the TP and returns:

Exe	Execute 0 Alu												
State B													
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD				
1	0	0	0	0	0	0	0	0	0				

Exhibit 2036.doc R400_Sequences.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257810

ATI Ex. 2109 IPR2023-00922 Page 100 of 326

No. of Concession, Name		ORIGINATE DATE 24 September, 2001		EDIT E	r, 201546	R400	PAGE 50 of 51		
	Status Bits								
	VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
	1	ALU	1	0	0	0	0	1	0

Picked up by the ALU and returns (lets say the TP has not returned yet): Alloc Position 1 $\,$

State Bi	its								
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
2	0	0	0	0	0	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	01	1	0	1	0

If the SX has the place for the export, the SQ is going to allocate and pick up the thread for execution. It returns to the RS in this state:

Execute 0 Alu 0 Tex

State Bits										
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD	
3	1	0	0	0	0	0	0	0	0	

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	1	0	0	0	1	1	0

Now, since the TP has not returned yet, we must wait for it to return because we cannot issue multiple texture requests. The TP returns, clears the PENDING bit and we proceed:

Alloc Param 3

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
4	0	0	0	0	1	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	10	3	1	1	0

Once again the SQ makes sure the SX has enough room in the Parameter cache before it can pick up this thread.

Execute_end 0 Alu 0 Tex 1 Alu 0 Alu

State Bit										
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD	
5	1	0	0	0	1	0	100	0	0	

Exhibit 2036 doc R400_Sequencer.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257811

ATI Ex. 2109 IPR2023-00922 Page 101 of 326

Añ	ORIGINATE DATE 24 September, 2001		4 Septen	T DATE nber, 201546		DOCUMENT-RE			PAGE 51 of 51
Status Bits									
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LA	ST
1	TEX	1	0	0	0	1	1	0	

This executes on the TP and then returns:

State Bit	ts								
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
5	2	0	0	0	1	0	100	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	1	0	0	1	1	1

Waits for the TP to return because of the textures reads are pending (and SERIAL in this case). Then executes and does not return to the RS because the LAST bit is set. This is the end of this thread and before dropping it on the floor, the SQ notifies the SX of export completion.

24. Open issues

Need to do some testing on the size of the register file as well as on the register file allocation method (dynamic VS static).

Saving power?

Exhibit 2036 docR400_Sequences.doc 73119 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257812

ATI Ex. 2109 IPR2023-00922 Page 102 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 20157	GEN-CXXXXX-REVA	1 of 54
Author:	Laurent Lefebvre			l .
Issue To:		Copy No:		
		Сору но.		
	R400 S	equencer Spe	ecification	
		SQ		
		Version 2.0 <u>9</u> 8		
req		ted uses of the block. It a	cer block (SEQ). It provides an ov Iso describes the block interfaces,	
Document L		rforce\r400\doc_lib\design\b Sequencer Specification APPROVALS	olocks\sq\R400_Sequencer.doc Signature/Date	
Remarks:				
	NTIALLY DETRIME	NTAL TO THE INTE	NFORMATION THAT CO REST OF ATI TECHNOL E OR DISCLOSURE.	
work created in unpublished w confidential, pro	n 2001. The use of this copy ork. The copyright notice oprietary information and tra	yright notice is intended to is not an admission that ide secrets of ATI. No part	rial in this document constitutes an provide notice that ATI owns a cop publication has occurred. This w of this document may be used, re sion of ATI Technologies Inc."	oyright in this ork contains
Exhibit 2037.doc	:R400_Sequencer.doc 74373 Bytes*** ©	ATI Confidential. Referer	nce Copyright Notice on Cover Pa	ge © ***

ATI 2037 LG v. ATI IPR2015-00325

AMD1044_0257813

ATI Ex. 2109 IPR2023-00922 Page 103 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 20157		2 of 54

Table Of Contents	
1. OVERVIEW	07
I. OVERVIEW 1.1 Top Level Block Diagram	
1.2 Data Flow graph (SP).	1110
1.3 Control Graph	
2. INTERPOLATED DATA BUS	
3. INSTRUCTION STORE	
4. SEQUENCER INSTRUCTIONS	
5. CONSTANT STORES.	
5.1 Memory organizations	
5.2 Management of the Control Flow Constants	1615
5.3 Management of the re-mapping tables	
5.3.1 R400 Constant management	
5.3.2 Proposal for R400LE constant management	
5.3.3 Dirty bits	******
5.3.4 Free List Block	
5.3.5 De-allocate Block	
5.3.6 Operation of Incremental model	
5.4 Constant Store Indexing	
5.5 Real Time Commands	
5.6 Constant Waterfalling	2019
6. LOOPING AND BRANCHES	<u>2120</u>
6.1 The controlling state.	
6.2 The Control Flow Program	
6.2.1 Control flow instructions table	
6.3 Implementation	
6.4 Data dependant predicate instructions	
6.5 HW Detection of PV,PS	
6.6 Register file indexing	
6.7 Debugging the Shaders	
6.7.1 Method 1: Debugging registers	
6.7.2 Method 2: Exporting the values in the GPRs	
7. PIXEL KILL MASK	
8. MULTIPASS VERTEX SHADERS (HOS)	2726
9. REGISTER FILE ALLOCATION	
10. FETCH ARBITRATION	
11. ALU ARBITRATION	
12. HANDLING STALLS	
13. CONTENT OF THE RESERVATION STATION FIFOS	3029
14. THE OUTPUT FILE	3029
15. IJ FORMAT	
15.1 Interpolation of constant attributes	3029
16. STAGING REGISTERS	<u>3130</u>

Exhibit 2037.doc R400_Sequencer.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257814

ATI Ex. 2109 IPR2023-00922 Page 104 of 326

AA	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 20157	GEN-CXXXXX-REVA	3 of 54
17.1.1	Pixel exports:	*****		
17.1.2	Vertex exports:			
17.1.3			******	
9.1 Rea	al time commands			

3.1 Ext	ternal Interfaces			
23.2 SC	to SP Interfaces			
<u>23.2.1</u>	SC SP#			
23.2.2	SC SQ			
23.2.3	SQ to SX(SP): Interpo	lator bus		
23.2.4	SQ to SP: Staging Re	gister Data		
23.2.5	VGT to SQ : Vertex in	terface		
23.2.6				
23.2.7				
23.2.8				
23.2.9				
23.2.10				
23.2.11				
23.2.12				
		interface (must be alig	ned with the SP data)	
23.2.14	SP to SQ: Constant a	ddress load/ Predicate	Set/Kill set	
<u>23.2.15</u>	SQ to SPx: constant I	proadcast		
23.2.16	SQ to CP: RBBM bus			
23.2.17				
had I had . S. I				

Exhibit 2037.doc R400_Sequencer.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257815

100000

ATI Ex. 2109 IPR2023-00922 Page 105 of 326

AN	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
20	24 September, 2001	4 September, 20157		4 of 54
.3 Exa	mple of control flow p	ogram execution		
. OPEN	I ISSUES	******		5451
-999- 19 ADDEN N -		***********************************	***	7
				9

	RUCTION STORE			

				15
5.3.2	Proposal for R400LE	constant managemen	•	
5.3.3	Dirty bits	****		
5.3.4-	Free List Block			
5.3.5	De-allocate Block			
The C	ontrol Flow Program.		*****	
5.2.1	Control flow instruction	ns table		21
			*****	23
- Data d	dependant predicate i	nstructions		24
- HW D	etection of PV.PS			
Regis	ter file indexing			
	1			
ALU/	ARBITRATION	****		
	LING STALLS	******************************		
. CONT	ENT OF THE RESER	VATION STATION F	IFOS	
. THE C	OUTPUT FILE	*****		
.1 Inte	rpolation of constant a	ttributes		
THE F	PARAMETER CACHE	• • • • • • • • • • • • • • • • • • • •		
1 Eve	ort restrictions			32

Exhibit 2037.doc R400_Sequencer.doc 74373 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257816

ATI Ex. 2109 IPR2023-00922 Page 106 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
<u>~~~</u>	24 September, 2001	4 September, 20157	GEN-CXXXXX-REVA	5 of 54
17.1.1-	Pixel exports:		*****	
17.1.2	Vertex exports:	*****		
17.1.3	Pass thru exports:			
17.2 Arbi	tration restrictions	*****		
	ex-Shading Shading			
	AL INTERPOLATION			
1 <u>9.1 Rea</u>	I time commands			
	*			
	Pixel shaders			
	E MANAGEMENT		*******	
23. INTE		*****	*****	
	to SP Interfaces			36
23.2.1				
23.2.2 —	SC_SQ			
23.2.3-	-SQ to SX(SP): Interpo	plator bus		
23.2.4	SQ to SP: Staging Re	gister Data		
23.2.5	VGT to SQ : Vertex in	terface		
23.2.6	SQ to SX: Control bu	\$		
23.2.7	SX to SQ : Output file	control		
23 2 8	I.			43
23.2.9				Ла
23.2.11				
			A 1870	
23.2.13			Set/Kill set	
23.2.16	CP to SQ: RBBM bus			46
23.2.17	SQ to CP: State repo	nt		47
				4 ****

Exhibit 2037.doc R400_Sequencer.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257817

ATI Ex. 2109 IPR2023-00922 Page 107 of 326

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE	
	24 September, 2001	4 September, 20157		6 of 54	
Revision	Changes:				
Rev 0.1 (Laurent Lefebvre)First draft.Date: May 7, 2001First draft.					
Rev 0.2 (Laur Date : July 9, 1 Rev 0.3 (Laur Date : August Rev 0.4 (Laur Date : August Rev 0.5 (Laur Date : Septer Rev 0.6 (Laur Date : Septer Rev 0.7 (Laur Date : Octobe Rev 0.8 (Laur Date : Octobe	2001 ent Lefebvre) 6, 2001 ent Lefebvre) 24, 2001 ent Lefebvre) iber 7, 2001 ent Lefebvre) iber 24, 2001 ent Lefebvre) r 5, 2001 ent Lefebvre) r 8, 2001 ent Lefebvre) r 8, 2001 ent Lefebvre)	SP Revi Augu Addu file a flow Addu Chai arch Addu store data Chai flexil Inco	nged the interfaces to reflect the chang Added some details in the arbitration so ewed the Sequencer spec after the m ust 3, 2001. det the dynamic allocation method fo and an example (written in part by V of pixels/vertices in the sequencer. ed timing diagrams (Vic) nged the spec to reflect the ne itecture. Added interfaces. ed constant store management, in a management, control flow manager dependant predication. nged the control flow method to obe. Also updated the external interface rporated changes made in the 10/18/C	ection. eeting on r register ic) of the ew R400 nstruction ment and be more is. D1 control	
Date : Octobe Rev 1.0 (Lauro Date : Octobe Rev 1.1 (Lauro	ent Lefebvre) r 19, 2001 ent Lefebvre)	the regis Refii Addo	meeting. Added a NOP instruction, conditional_execute_or_jump. Added sters. ned interfaces to RB. Added state regis	d debug sters. ed delta	
Date : Octobe Rev 1.2 (Lauro Date : Novemi Rev 1.3 (Lauro Date : Novemi	ent Lefebvre) ber 16, 2001 ent Lefebvre)	Meth Inter	ision. Changed VGT→SP0 interface lods added. faces greatly refined. Cleaned up the s ed the different interpolation modes.	U	
Rev 1.4 (Lauro Date : Decemi Rev 1.5 (Lauro	ber 6, 2001 ent Lefebvre)	the ^v man Rem	ed the auto incrementing counters. /GT→SQ interface. Added content on agement. Updated GPRs. loved from the spec all interfaces that	constant at weren't	
Date : Decemi Rev 1.6 (Laur Date : January	ent Lefebvre)	cons sync Adde deta	hronization fields and explanation. ed more details on the staging registe il about the parameter caches. Cha	PA—SQ er. Added nged the	
Rev 1.7 (Lauro Date : Februa Rev 1.8 (Lauro Date : March o	ry 4, 2002 ent Lefebvre)	Add upd Add inter New clau	instruction to a Conditionnal_call in ed details on constant managem ted the diagram. ed Real Time parameter control in face. Updated the control flow section. interfaces to the SX block. Added th se modifier, removed the end o uctions.	ent and the SX ne end of	
Date : March 2	18, 2002 rent Lefebvre) 25, 2002 rent Lefebvre) , 2002 ent Lefebvre)	ensu Upda expo Adda spec	rangement of the CF instruction bits in irre byte alignement. ated the interfaces and added a se orting rules. ed CP state report interface. Last versi- with the old control flow scheme control flow scheme	ection on	
Exhibit 2037.docF	1400_Sequencer.doc 74373 Bytes***	ATI Confidential. Ref	erence Copyright Notice on Cover P	age © ***	

AMD1044_0257818

ATI Ex. 2109 IPR2023-00922 Page 108 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 20157	GEN-CXXXXX-REVA	7 of 54
Rev 2.01 (Lau	rent Lefebvre)		d slightly the control flow instruct	ions to
Date : May 2, 2	2002	allow for	rce jumps and calls.	
Rev 2.02 (Lau	rent Lefebvre)	Updated	I the Opcodes. Added type field	to the
Date : May 13	, 2002	constan	t/pred interface. Added Last field	to the
		SQ→SF	instruction load interface.	
Rev 2.03 (Lau	rent Lefebvre)	SP int	erface updated to include pred	dication
Date : July 15,	2002	optimiza	tions. Added the predicate no	o stall
		instructi	ons,	
Rev 2.04 (Lau	rent Lefebvre)	Docume	ented the new parameter generation s	scheme
Date :August 2	2, 2002	for XY c	oordinates points and lines STs.	
Rev 2.05 (Lau	rent Lefebvre)	Some	interface changes and an archi	tectural
Date : Septem	ber 10, 2002	change	to the auto-counter scheme.	
Rev 2.06 (Lau	rent Lefebvre)	Wideneo	d the event interface to 5 bits. Som	e other
Date : October	r 11, 2002	little type	os corrected.	
Rev 2.07 (Lau	rent Lefebvre)	Loops,	jumps and calls are now using a	13 bit
Date : October	r 14, 2002	address	which allows to jump and call ar	nd loop
		around	any control flow addresses (do	es not
		requires	to be even anymore).	
Rev 2.08 (Lau	rent Lefebvre)	Clarifica	tion updates after discussion with Cla	iy.
Date : October	r 16, 2002			
<u>Rev 2.09 (Lau</u>	rent Lefebvre)	Correcte	ed the SQ→SP staging register interfa	ace.
Date : January	<u>7, 2003</u>			

Exhibit 2037. doc R400_Sequences.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257819

ATI Ex. 2109 IPR2023-00922 Page 109 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 20157		8 of 54

1. Overview

The sequencer chooses two ALU threads and a fetch hread to execute, and executes all of the instructions in a block before looking for a new clause of the same type. Two ALU threads are executed interleaved to hide the ALU latency. The arbitrator will give priority to older threads. There are two separate reservation stations, one for pixel vectors and one for vertices vectors. This way a pixel can pass a vertex and a vertex can pass a pixel.

To support the shader pipe the sequencer also contains the shader instruction cache, constant store, control flow constants and texture state. The four shader pipes also execute the same instruction thus there is only one sequencer for the whole chip.

The sequencer first arbitrates between vectors of 64 vertices that arrive directly from primitive assembly and vectors of 16 quads (64 pixels) that are generated in the scan converter.

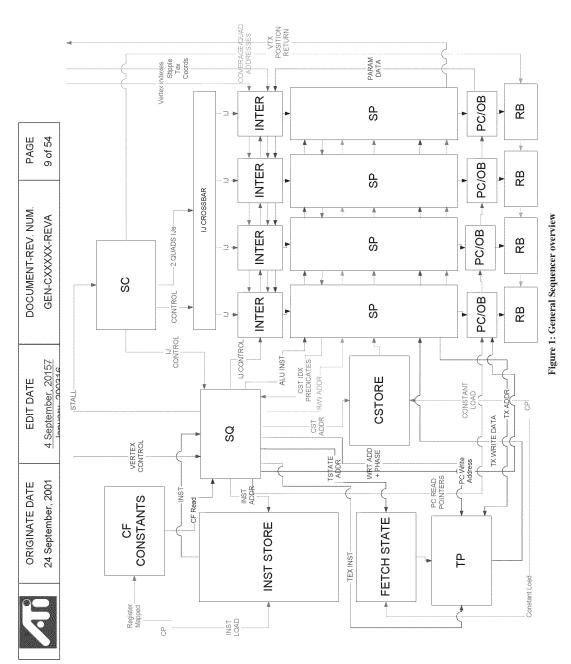
The vertex or pixel program specifies how many GPRs it needs to execute. The sequencer will not start the next vector until the needed space is available in the GPRs.

Exhibit 2037. doc/R400_Sequeencer.doc 74373 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257820

ATI Ex. 2109 IPR2023-00922 Page 110 of 326

PROTECTIVE ORDER MATERIAL





AMD1044_0257821

ATI Ex. 2109 IPR2023-00922 Page 111 of 326

 				_
ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE	
24 September, 2001	4 September, 20157		10 of 54	

1.1 Top Level Block Diagram

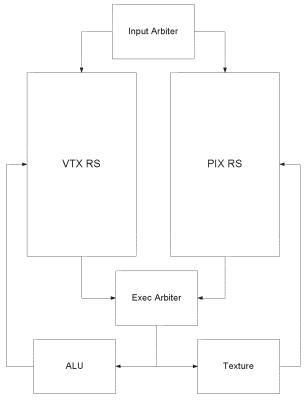


Figure 2: Reservation stations and arbiters

Under this new scheme, the sequencer (SQ) will only use one global state management machine per vector type (pixel, vertex) that we call the reservation station (RS).

Exhibit 2037. doc.R400_Sequences..doc 74373 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257822

ATI Ex. 2109 IPR2023-00922 Page 112 of 326

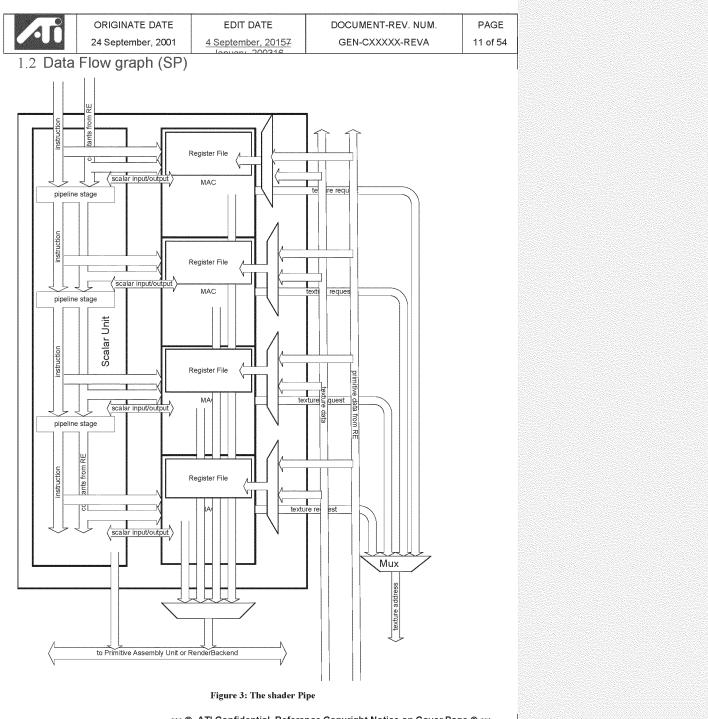


Exhibit 2037 docR400_Sequencer.doc 74373 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257823

ATI Ex. 2109 IPR2023-00922 Page 113 of 326

-					
ſ		ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
	<u>~</u>	24 September, 2001	4 September, 20157		12 of 54

The gray area represents blocks that are replicated 4 times per shader pipe (16 times on the overall chip).

1.3 Control Graph

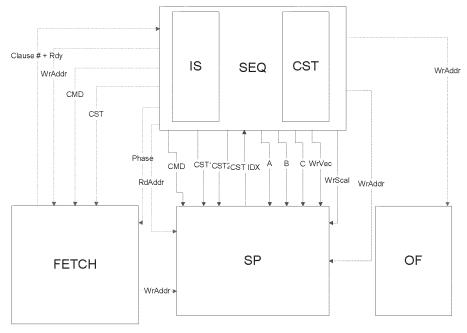


Figure 4: Sequencer Control interfaces

In green is represented the Fetch control interface, in red the ALU control interface, in blue the Interpolated/Vector control interface and in purple is the output file control interface.

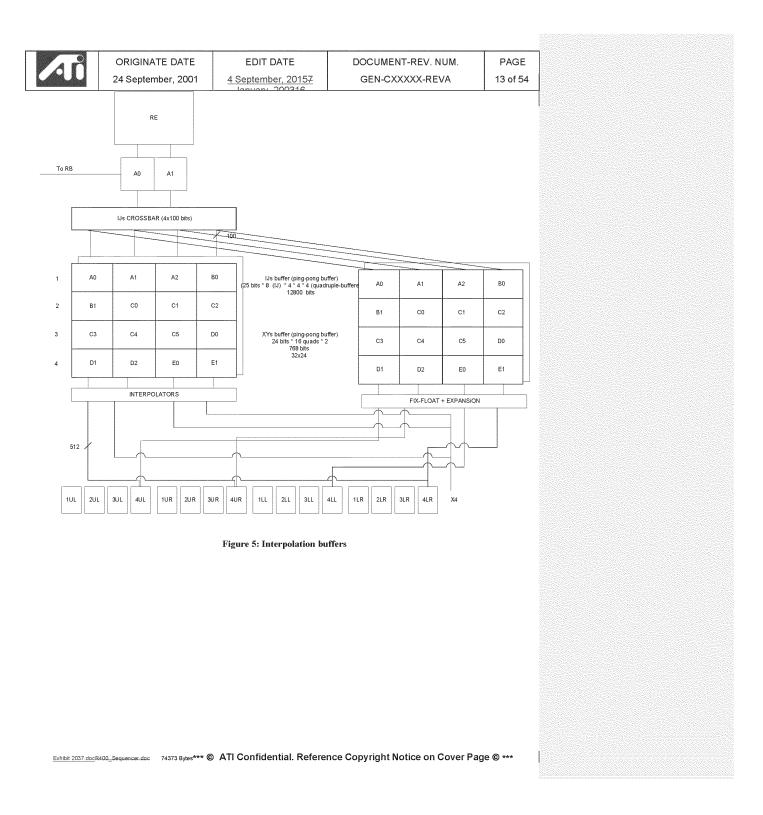
2. Interpolated data bus

The interpolators contain an IJ buffer to pack the information as much as possible before writing it to the register file.

Exhibit 2037. docR400_Sequencer.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257824

ATI Ex. 2109 IPR2023-00922 Page 114 of 326



ATI Ex. 2109 IPR2023-00922 Page 115 of 326

PROTECTIVE ORDER MATERIAL

		r23					5 4 2	52- 55	56- 59	> 🌣 🕄	
		T22 ⁻					V V 32-48- 35 51	အ ဗ္တိ ဒု	> 4 5	V V 44-60- 47 63	×
		T19 T20 T21 T22 T23					> 0 6	23 ² 0 <	24- 27	31 <mark>2</mark> 8 <	XL>
		T20					0-3 0-3	> 4-7	⇒ ∽ £	> 4 tî	
		T19							EO	E1	
		T18								8	
	4	T17			χΩ	XY E1		8	G	C3	
PAGE	14 of 54	T16			ы	ш				BO	Р2
	-	T15			В	ш	5	D2			Δ
ation		T14	גפ	×β		×۵	ប៊	5	C5		
ecifica		T13	δ	8		8	B1				
cer Sp		T12	Б	6		8	AO	A1	A2		
duen		T11	e E	≿2	ς5 ×		Į		ЮШ	μ	******
R400 Sequencer Specification		T10	10.0	64 C	C2	0	2			8	
		T9		C4	C5			ខ	5	C2	
	0157	Т8	≿ຬ	≿ଃ	≿ວ	ΣS				BO	d iamanan
EDIT DATE	4 September, 20157	11	ß	8	δ	3	Б	D2			À
EDIT	Septem	T6	ទ	ខ	δ	3	ប៊	5	C5		
	4	T5	≿ॼ			×8	B1				
ATE	2001	Τ4	Б.			BO	AO	A1	A2		
TE D/	nber, 2	13	8			BO	¥8- 51	55 52 -	59 - X	S°X	
ORIGINATE DATE	September, 2001	12	×۹	۶Ł	≿₹		35 32 ₹	% ဗို ဗိ	∑ 4 &	XY XY 44- 60- 47 63	\succ
OR	24 (F	Ao	A1	A2		19 1 6- 19 19	3 \$ ₹	24- 27		×
90		2	Ao	A1	A2		0-3 X	¥ 7-7	≍ ∾ ≿	15 72 ×2	
	3		d° o	ds –	SP SP	ი წ	dS o	d –	S P	а с	Constantial Constantia Const

Figure 6: Interpolation timing diagram

Exhibit 2027 doc/8400_Sequences doo 74573 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257826

ATI Ex. 2109 IPR2023-00922 Page 116 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
	24 September, 2001	4 September, 20157	GEN-CXXXXX-REVA	15 of 54	

Above is an example of a tile the sequencer might receive from the SC. The write side is how the data get stacked into the XY and IJ buffers, the read side is how the data is passed to the GPRs. The IJ information is packed in the IJ buffer 4 quads at a time or two clocks. The sequencer allows at any given time as many as four quads to interpolate a parameter. They all have to come from the same primitive. Then the sequencer controls the write mask to the GPRs to write the valid data in.

3. Instruction Store

There is going to be only one instruction store for the whole chip. It will contain 4096 instructions of 96 bits each.

It is likely to be a 1 port memory; we use 1 clock to load the ALU instruction, 1 clocks to load the Fetch instruction, 1 clock to load 2 control flow instructions and 1 clock to write instructions.

The instruction store is loaded by the CP thru the register mapped registers.

The VS_BASE and PS_BASE context registers are used to specify for each context where its shader is in the instruction memory.

For the Real time commands the story is quite the same but for some small differences. There are no wrap-around points for real time so the driver must be careful not to overwrite regular shader data. The shared code (shared subroutines) uses the same path as real time.

4. Sequencer Instructions

All control flow instructions instructions are handled by the sequencer only. The ALUs will perform NOPs during this time (MOV PV, PV, PS, PS) if they have nothing else to do.

5. Constant Stores

5.1 Memory organizations

A likely size for the ALU constant store is 1024x128 bits. The read BW from the ALU constant store is 128 bits/clock and the write bandwidth is 32 bits/clock (directed by the CP bus size not by memory ports).

The maximum logical size of the constant store for a given shader is 256 constants. Or 512 for the pixel/vertex shader pair. The size of the re-mapping table is 128 lines (each line addresses 4 constants). The write granularity is 4 constants or 512 bits. It takes 16 clocks to write the four constants. Real time requires 256 lines in the physical memory (this is physically register mapped).

The texture state is also kept in a similar memory. The size of this memory is 320x96 bits (128 texture states for regular mode, 32 states for RT). The memory thus holds 128 texture states (192 bits per state). The logical size exposes 32 different states total, which are going to be shared between the pixel and the vertex shader. The size of the re-mapping table to for the texture state memory is 32 lines (each line addresses 1 texture state lines in the real memory). The CP write granularity is 1 texture state lines (or 192 bits). The driver sends 512 bits but the CP ignores the top 320 bits. It thus takes 6 clocks to write the texture state. Real time requires 32 lines in the physical memory (this is physically register mapped).

The control flow constant memory doesn't sit behind a renaming table. It is register mapped and thus the driver must reload its content each time there is a change in the control flow constants. Its size is 320*32 because it must hold 8 copies of the 32 dwords of control flow constants and the loop construct constants must be aligned.

The constant re-mapping tables for texture state and ALU constants are logically register mapped for regular mode and physically register mapped for RT operation.

Exhibit 2037. doc/R400_Sequences..doc 74373 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044 0257827

ATI Ex. 2109 IPR2023-00922 Page 117 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 20157		16 of 54

5.2 Management of the Control Flow Constants

The control flow constants are register mapped, thus the CP writes to the according register to set the constant, the SQ decodes the address and writes to the block pointed by its current base pointer (CF_WR_BASE). On the read side, one level of indirection is used. A register (SQ_CONTEXT_MISC.CF_RD_BASE) keeps the current base pointer to the control flow block. This register is copied whenever there is a state change. Should the CP write to CF after the state change, the base register is updated with the (current pointer number +1)% number of states. This way, if the CP doesn't write to CF the state is going to use the previous CF constants.

5.3 Management of the re-mapping tables

5.3.1 R400 Constant management

The sequencer is responsible to manage two re-mapping tables (one for the constant store and one for the texture state). On a state change (by the driver), the sequencer will broadside copy the contents of its re-mapping tables to a new one. We have 8 different re-mapping tables we can use concurrently.

The constant memory update will be incremental, the driver only need to update the constants that actually changed between the two state changes.

For this model to work in its simplest form, the requirement is that the physical memory MUST be at least twice as large as the logical address space + the space allocated for Real Time. In our case, since the logical address space is 512 and the reserved RT space can be up to 256 entries, the memory must be of sizes 1280 and above. Similarly the size of the texture store must be of 32*2+32 = 96 entries and above.

5.3.2 Proposal for R400LE constant management

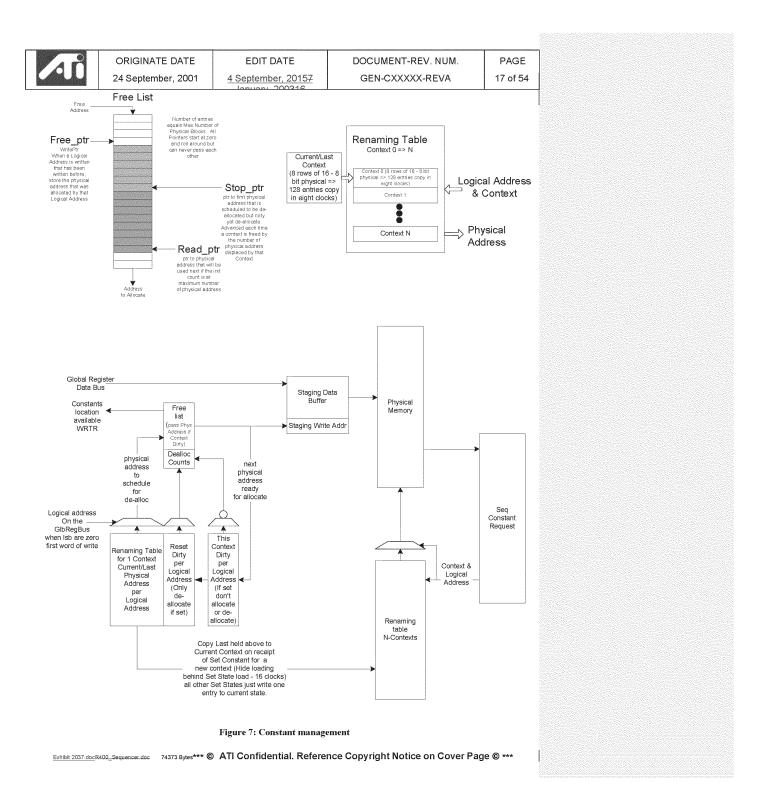
To make this scheme work with only 512+256 = 768 entries, upon reception of a CONTROL packet of state + 1, the sequencer would check for SQ_IDLE and PA_IDLE and if both are idle will erase the content of state to replace it with the new state (this is depicted in <u>Figure 8: De-allocation mechanismFigure 8: De-allocatin mechanismFigure 8: De-allocation mechanism</u>

The second path sets all context dirty bits that were used in the current state to 1 (thus allowing the new state to reuse these physical addresses if needed).

Exhibit 2037. doc R400_Sequences...doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044 0257828

ATI Ex. 2109 IPR2023-00922 Page 118 of 326



ATI Ex. 2109 IPR2023-00922 Page 119 of 326

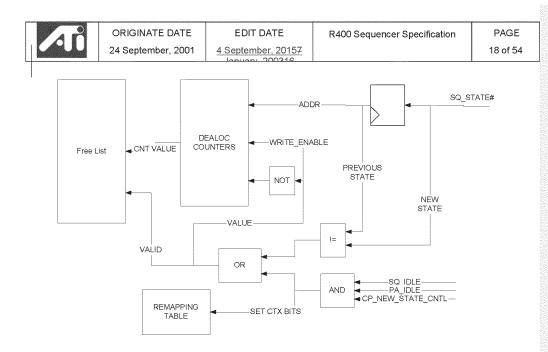


Figure 8: De-allocation mechanism for R400LE

5.3.3 Dirty bits

Two sets of dirty bits will be maintained per logical address. The first one will be set to zero on reset and set when the logical address is addressed. The second one will be set to zero whenever a new context is written and set for each address written while in this context. The reset dirty is not set, then writing to that logical address will not require de-allocation of whatever address stored in the renaming table. If it is set and the context dirty is not set, then the physical address is necessary to store the incoming data. If they are both set, then the data will be written into the physical address held in the renaming for the current logical address. No de-allocation or allocation takes place. This will happen when the driver does a set constant twice to the same logical address between context changes. NOTE: It is important to detect and prevent this, failure to do it will allow multiple writes to allocate all physical memory and thus hang because a context will not fit for rendering to start and thus free up space.

5.3.4 Free List Block

A free list block that would consist of a counter (called the IFC or Initial Free Counter) that would reset to zero and incremented every time a chunk of physical memory is used until they have all been used once. This counter would be checked each time a physical block is needed, and if the original ones have not been used up, us a new one, else check the free list for an available physical block address. The count is the physical address for when getting a chunk from the counter.

Storage of a free list big enough to store all physical block addresses.

Maintain three pointers for the free list that are reset to zero. The first one we will call write_ptr. This pointer will identify the next location to write the physical address of a block to be de-allocated. Note: we can never free more physical memory locations than we have. Once recording address the pointer will be incremented to walk the free list like a ring.

The second pointer will be called stop_ptr. The stop_ptr pointer will be advanced by the number of address chunks de-allocates when a context finishes. The address between the stop_ptr and write_ptr cannot be reused because they are still in use. But as soon as the context using then is dismissed the stop_ptr will be advanced.

The third pointer will be called read_ptr. This pointer will point will point to the next address that can be used for allocation as long as the read_ptr does not equal the stop_ptr and the IFC is at its maximum count.

Exhibit 2037.docR400_Sequencer.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257830

ATI Ex. 2109 IPR2023-00922 Page 120 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 20157	GEN-CXXXXX-REVA	19 of 54

5.3.5 De-allocate Block

This block will maintain a free physical address block count for each context. While in current context, a count shall be maintained specifying how many blocks were written into the free list at the write_ptr pointer. This count will be reset upon reset or when this context is active on the back and different than the previous context. It is actually a count of blocks in the previous context that will no longer be used. This count will be used to advance the write_ptr pointer to make available the set of physical blocks freed when the previous context was done. This allows the discard or de-allocation of any number of blocks in one clock.

5.3.6 Operation of Incremental model

The basic operation of the model would start with the write_ptr, stop_ptr, read_ptr pointers in the free list set to zero and the free list counter is set to zero. Also all the dirty bits and the previous context will be initialized to zero. When the first set constants happen, the reset dirty bit will not be set, so we will allocate a physical location from the free list counter is not at the max value. The data will be written into physical address zero. Both the additional copy of the renaming table and the context zeros of the big renaming table will be updated for the logical address that was written by set start with physical address of 0. This process will be repeated for any logical address that are not dirty until the context changes. If a logical address is hit that has its dirty bits set while in the same context, both dirty bits would be set, so the new data will be over-written to the last physical address assigned for this logical address. When the first draw command of the context is detected, the previous context stored in the additional renaming table will be copied to the larger renaming table in the current (new) context location. Then the set constant logical address it replaced in the renaming table would be entered at the write_ptr pointer location on the free list and the write_ptr will be incremented. The de-allocation counter for the previous context (eight) will be incremented. This as set states come in for this context one of the following will happen:

- 1.) No dirty bits are set for the logical address being updated. A line will be allocated of the free-list counter or the free list at read_ptr pointer if read_ptr != to stop_ptr .
- 2.) Reset dirty set and Context dirty not set. A new physical address is allocated, the physical address in the renaming table is put on the free list at write_ptr and it is incremented along with the de-allocate counter for the last context.
- 3.) Context dirty is set then the data will be written into the physical address specified by the logical address.

This process will continue as long as set states arrive. This block will provide backpressure to the CP whenever he has not free list entries available (counter at max and stop_ptr == read_ptr). The command stream will keep a count of contexts of constants in use and prevent more than max constants contexts from being sent.

Whenever a draw packet arrives, the content of the re-mapping table is written to the correct re-mapping table for the context number. Also if the next context uses less constants than the current one all exceeding lines are moved to the free list to be de-allocated later. This happens in parallel with the writing of the re-mapping table to the correct memory.

Now preferable when the constant context leaves the last ALU clause it will be sent to this block and compared with the previous context that left. (Init to zero) If they differ than the older context will no longer be referenced and thus can be de-allocated in the physical memory. This is accomplished by adding the number of blocks freed this context to the stop_ptr pointer. This will make all the physical addresses used by this context available to the read_ptr allocate pointer for future allocation.

This device allows representation of multiple contexts of constants data with N copies of the logical address space. It also allows the second context to be represented as the first set plus some new additional data by just storing the delta's. It allows memory to be efficiently used and when the constants updates are small it can store multiple context. However, if the updates are large, less contexts will be stored and potentially performance will be degraded. Although it will still perform as well as a ring could in this case.

5.4 Constant Store Indexing

In order to do constant store indexing, the sequencer must be loaded first with the indexes (that come from the GPRs). There are 144 wires from the exit of the SP to the sequencer (9 bits pointers x 16 vertexes/clock).

Exhibit 2037.docR400_Sequencesr.doc 74373 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257831

ATI Ex. 2109 IPR2023-00922 Page 121 of 326

		ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE		
		24 September, 2001	4 September, 20157		20 of 54		
Π	MOVA R1.X.I						

ADD R3,R4,C0[R2.X]// Uses the state from the sequencer to add R4 to C0[R2.X] into R3

Note that we don't really care about what is in the brackets because we use the state from the MOVA instruction. R2.X is just written again for the sake of simplicity and coherency.

The storage needed in the sequencer in order to support this feature is 2*64*9 bits = 1152 bits.

The address register is a signed integer, which ranges from -256 to 255.

5.5 Real Time Commands

The real time commands constants are written by the CP using the register mapped registers allocated for RT. It works is the same way than when dealing with regular constant loads BUT in this case the CP is not sending a logical address but rather a physical address and the reads are not passing thru the re-mapping table but are directly read from the memory. The boundary between the two zones is defined by the CONST_EO_RT control register. Similarly, for the fetch state, the boundary between the two zones is defined by the TSTATE_EO_RT control register.

5.6 Constant Waterfalling

In order to have a reasonable performance in the case of constant store indexing using the address register, we are going to have the possibility of using the physical memory port for read only. This way we can read 1 constant per clock and thus have a worst-case waterfall mode of 1 vertex per clock. There is a small synchronization issue related with this as we need for the SQ to make sure that the constants where actually written to memory (not only sent to the sequencer) before it can allow the first vector of pixels or vertices of the state to go thru the ALUs. To do so, the sequencer keeps 8 bits (one per render state) and sets the bits whenever the last render state is written to memory and clears the bit whenever a state is freed.

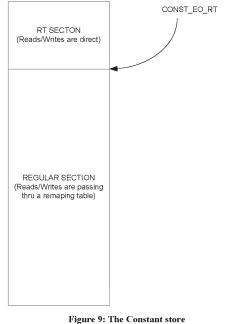


Exhibit 2037. docR400_Sequencer.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page 🕲 ***

AMD1044_0257832

ATI Ex. 2109 IPR2023-00922 Page 122 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 20157	GEN-CXXXXX-REVA	21 of 54

6. Looping and Branches

Loops and branches are planned to be supported and will have to be dealt with at the sequencer level. We plan on supporting constant loops and branches using a control program.

6.1 The controlling state.

The R400 controling state consists of:

Boolean[255:0] Loop_count[7:0][31:0] Loop_Start[7:0][31:0] Loop_Step[7:0][31:0]

That is 256 Booleans and 32 loops.

We have a stack of 4 elements for nested calls of subroutines and 4 loop counters to allow for nested loops.

This state is available on a per shader program basis.

6.2 The Control Flow Program

We'd like to be able to code up a program of the form:

1:	Loop	
2:	Exec	TexFetch
3:		TexFetch
4:		ALU
5:		ALU
6:		TexFetch
7:	End Lo	ор
8:	ALU E>	port

But realize that 3: may be dependent on 2: and 4: is almost certainly dependent on 2: and 3:. Without clausing, these dependencies need to be expressed in the Control Flow instructions. Additionally, without separate 'texture clauses' and 'ALU clauses' we need to know which instructions to dispatch to the Texture Unit and which to the ALU unit. This information will be encapsulated in the flow control instructions.

Each control flow instruction will contain 2 bits of information for each (non-control flow) instruction: a) ALU or Texture

b) Serialize Execution

(b) would force the thread to stop execution at this point (before the instruction is executed) and wait until all textures have been fetched. Given the allocation of reserved bits, this would mean that the count of an 'Exec' instruction would be limited to about 8 (non-control-flow) instructions. If more than this were needed, a second Exec (with the same conditions) would be issued.

Another function that relies upon 'clauses' is allocation and order of execution. We need to assure that pixels and vertices are exported in the correct order (even if not all execution is ordered) and that space in the output buffers are allocated in order. Additionally data can't be exported until space is allocated. A new control flow instruction:

Alloc <buffer select -- position, parameter, pixel or vertex memory. And the size required>.

would be created to mark where such allocation needs to be done. To assure allocation is done in order, the actual allocation for a given thread can not be performed unless the equivalent allocation for all previous threads is already completed. The implementation would also assure that execution of instruction(s) following the serialization due to the Alloc will occur in order -- at least until the next serialization or change from ALU to Texture. In most cases this will allow the exports to occur without any further synchronization. Only 'final' allocations or position allocations are

Exhibit 2037. docR400_Sequences.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257833

ATI Ex. 2109 IPR2023-00922 Page 123 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 20157		22 of 54

guaranteed to be ordered. Because strict ordering is required for pixels, parameters and positions, this implies only a single alloc for these structures. Vertex exports to memory do not require ordering during allocation and so multiple 'allocs' may be done.

6.2.1 Control flow instructions table

Here is the revised control flow instruction set.

Note that whenever a field is marked as RESERVED, it is assumed that all the bits of the field are cleared (0).

		NOP	100			
47 44	47 44 43 42 0					
0000	Addressing	RESERVED				

This is a regular NOP.

			Execute		
47 44	43	40 34	33 16	1512	11 0
0001	Addressing	RESERVED	Instructions type + serialize (9 instructions)	Count	Exec Address

	Execute_End								
47 44	43	40 34	33 16	1512	11 0				
0010	Addressing	RESERVED	Instructions type + serialize (9	Count	Exec Address				
			instructions)						

Execute up to 9 instructions at the specified address in the instruction memory. The Instruction type field tells the sequencer the type of the instruction (LSB) (1 = Texture, 0 = ALU and whether to serialize or not the execution (MSB) (1 = Serialize, 0 = Non-Serialized). If Execute_End this is the last execution block of the shader program.

	Conditional_Execute								
47 44	43	42	41 34	3316	15 12	11 0			
0011	Addressing	Condition	Boolean address	Instructions type + serialize (9 instructions)	Count	Exec Address			

	Conditional_Execute_End								
4	7 44	43	42	41 34	3316	15 12	11 0		
	0100	Addressing	Condition	Boolean	Instructions type + serialize (9	Count	Exec Address		
				address	instructions)				

If the specified Boolean (8 bits can address 256 Booleans) meets the specified condition then execute the specified instructions (up to 9 instructions). If the condition is not met, we go on to the next control flow instruction. If Conditional_Execute_End and the condition is met, this is the last execution block of the shader program.

	Conditional_Execute_Predicates									
47 44	43	42	41 36	35 34	3316	1512	11 0			
0101	Addressing	Condition	RESERVED	Predicate	Instructions	Count	Exec Address			
				vector	type + serialize					
					(9 instructions)					

Conditional_Execute_Predicates_End										
47 44	43	42	41 36	35 34	3316	1512	11 0			
0110	Addressing	Condition	RESERVED	Predicate vector	Instructions type + serialize	Count	Exec Address			
					(9 instructions)					

Check the AND/OR of all current predicate bits. If AND/OR matches the condition execute the specified number of instructions. We need to AND/OR this with the kill mask in order not to consider the pixels that aren't valid. If the

Exhibit 2037.doc/R400_Sequencer.doc 74373 Bytes**** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257834

ATI Ex. 2109 IPR2023-00922 Page 124 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 20157	GEN-CXXXXX-REVA	23 of 54

condition is not met, we go on to the next control flow instruction. If Conditional_Execute_Predicates_End and the condition is met, this is the last execution block of the shader program.

	Conditional_Execute_Predicates_No_Stall										
47 44	43	42	41 36	35 34	3316	1512	11 0				
1101	Addressing	Condition	RESERVED	Predicate vector	Instructions type + serialize (9 instructions)	Count	Exec Address				

	Conditional_Execute_Predicates_No_Stall_End										
47 44	43	42	41 36	35 34	3316	1512	11 0				
1110	Addressing	Condition	RESERVED	Predicate vector	Instructions type + serialize	Count	Exec Address				
					(9 instructions)						

Same as Conditionnal_Execute_Predicates but the SQ is not going to wait for the predicate vector to be updated. You can only set this in the compiler if you know that the predicate set is only a refinement of the current one (like a nested if) because the optimization would still work.

	Loop_Start								
47 44	43	42 21	20 16	1513	12 0				
0111	Addressing	RESERVED	loop ID	RESERVED	Jump address				

Loop Start. Compares the loop iterator with the end value. If loop condition not met jump to the address. Forward jump only. Also computes the index value. The loop id must match between the start to end, and also indicates which control flow constants should be used with the loop.

	Loop_End									
47 44	43	42	41 36	3534	33 22	21	20 16	1513	12 0	122
1000	Addressing	Cond	RESERVED	Predicate	RESERVED	Pred	loop ID	RESERVED	start	
				Vector		break			address	100

Loop end. Increments the counter by one, compares the loop count with the end value. If loop condition met, continue, else, jump BACK to the start of the loop. If predicate break != 0, then compares predicate vector n (specified by predicate Vector) to condition. If all bits meet condition then break the loop.

The way this is described does not prevent nested loops, and the inclusion of the loop id make this easy to do.

Conditionnal_Call							
47 44	43	42	41 34	33 14	13	12 0	
1001	Addressing	Condition	Boolean address	RESERVED	Force Call	Jump address	

If the condition is met, jumps to the specified address and pushes the control flow program counter on the stack. If force call is set the condition is ignored and the call is made always.

	Return							
47 44	43	42 0						
1010	Addressing	RESERVED	10002					

Pops the topmost address from the stack and jumps to that address. If nothing is on the stack, the program will just continue to the next instruction.

	Conditionnal_Jump								
47 44	43	42	41 34	33	32 14	13	12 0		
1011	Addressing	Condition	Boolean	FW only	RESERVED	Force Jump	Jump address		
			address						

Exhibit 2037.doc R400_Sequences.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257835

ATI Ex. 2109 IPR2023-00922 Page 125 of 326

Añ	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
MUY	24 September, 2001	4 September, 20157		24 of 54
If force jump is	s set the condition is igno		e always. If FW only is set then only	forward jumps
are allowed.	5			

Allocate						
47 44	43	4241	40 3	20		
1100	Debug	Buffer Select	RESERVED	Size		

Buffer Select takes a value of the following:

01 - position export (ordered export)

10 - parameter cache or pixel export (ordered export)

11 - pass thru (out of order exports).

Size field is only used to reserve space in the export buffer for pass thru exports. Valid values are 1 (1 line) thru 9 (9 lines). It should be determined by the compiler/assembler by taking max index used +1.

If debug is set this is a debug alloc (ignore if debug DB_ON register is set to off).

6.3 Implementation

The envisioned implementation has a buffer that maintains the state of each thread. A thread lives in a given location in the buffer during its entire life, but the buffer has FIFO qualities in that threads leave in the order that they enter. Actually two buffers are maintained -- one for Vertices and one for Pixels. The intended implementation would allow for:

16 entries for vertices 48 entries for pixels.

From each buffer, arbitration logic attempts to select 1 thread for the texture unit and 2 (interleaved) thread for the ALU unit. Once a thread is selected it is read out of the buffer, marked as invalid, and submitted to appropriate execution unit. It is returned to the buffer (at the same place) with its status updated once all possible sequential instructions have been executed. A switch from ALU to TEX or visa-versa or a Serialize_Execution modifier forces the thread to be returned to the buffer.

Each entry in the buffer will be stored across two physical pieces of memory - most bits will be stored in a 1 read port device. Only bits needed for thread arbitration will be stored in a highly multi-ported structure. The bits kept in the 1 read port device will be termed 'state'. The bits kept in the multi-read ported device will be termed 'status'.

'State Bits' needed include:

- 1. Control Flow Instruction Pointer (13 bits),
- 2. Execution Count Marker 4 bits),
- 3. Loop Iterators (4x9 bits),
- 4. Loop Counters (4x9 bits),
- 5. Call return pointers (4x13 bits),
- 6. Predicate Bits (64 bits),
- 7. Export ID (1 bit),
- 8. Parameter Cache base Ptr (7 bits),
- 9. GPR Base Ptr (8 bits),
- 10. Context Ptr (3 bits).
- 11. LOD corrections (6x16 bits)
- 12. Valid bits (64 bits)
- RT (1 bit) Signifies that this thread is a Real Time thread. This bit must be sent to the Constant store state machine when reading it.

Absent from this list are 'Index' pointers. These are costly enough that I'm presuming that they are instead stored in the GPRs. The first seven fields above (Control Flow Ptr, Execution Count, Loop Counts, call return ptrs, Predicate bits, PC base ptr and export ID) are updated every time the thread is returned to the buffer based on how much

Exhibit 2037.doc.R400_Sequencer.doc 74373 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257836

ATI Ex. 2109 IPR2023-00922 Page 126 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 20157	GEN-CXXXXX-REVA	25 of 54

progress has been mode on thread execution. GPR Base Ptr, Context Ptr and LOD corrections are unchanged throughout execution of the thread.

'Status Bits' needed include:

- Valid Thread
- Texture/ALU engine needed
- Texture Reads are outstanding
- Waiting on Texture Read to Complete
- Allocation Wait (2 bits)
- 00 No allocation needed
- 01 Position export allocation needed (ordered export)
- 10 Parameter or pixel export needed (ordered export)
- 11 pass thru (out of order export)
- Allocation Size (4 bits)
- Position Allocated
- Mem/Color Allocated
- First thread of a new context
- Event thread (NULL thread that needs to trickle down the pipe)
- Last (1 bit)
- Pulse SX (1 bit)

All of the above fields from all of the entries go into the arbitration circuitry. The arbitration circuitry will select a winner for both the Texture Engine and for the ALU engine. There are actually two sets of arbitration -- one for pixels and one for vertices. A final selection is then done between the two. But the rest of this implementation summary only considers the 'first' level selection which is similar for both pixels and vertices.

Texture arbitration requires no allocation or ordering so it is purely based on selecting the 'oldest' thread that requires the Texture Engine.

ALU arbitration is a little more complicated. First, only threads where either of Texture_Reads_outstanding or Waiting_on_Texture_Read_to_Complete are '0' are considered. Then if Allocation_Wait is active, these threads are further filtered based on whether space is available. If the allocation is position allocation, then the thread is only considered if all 'older' threads have already done their position allocation (position allocated bits set). If the allocation is parameter or pixel allocation, then the thread is only considered if it is the oldest thread. Also a thread is not considered if it is a parameter or pixel or position allocation, has its First_thread_of_a_new_context bit set and would cause ALU interleaving with another thread performing the same parameter or pixel or position allocation. Finally the 'oldest' of the threads that pass through the above filters is selected. If the thread needed to allocate, then at this time the allocation is done, based on Allocation_Size. If a thread has its "last" bit set, then it is also removed from the buffer, never to return.

If I now redefine 'clauses' to mean 'how many times the thread is removed from the thread buffer for the purpose of exection by either the ALU or Texture engine', then the minimum number of clauses needed is 2 -- one to perform the allocation for exports (execution automatically halts after an 'Alloc' instruction) (but doesn't performs the actual allocation) and one for the actual ALU/export instructions. As the 'Alloc' instruction could be part of a texture clause (presumably the final instruction in such a clause), a thread could still execute in this minimal number of 2 clauses, even if it involved texture fetching.

The Texture_Reads_Outstanding bit must be updated by the sequencer, based on keeping track of how many Texture Clauses have been executed by a given thread that have not yet had there data returned. Any number above 0 results in this bit being set. We could consider forcing synchronization such that two texture clauses for a given thread may not be outstanding at any time (that would be my preference for simplicity reasons and because it would require only very little change in the texture pipe interface). This would allow the sequencer to set the bit on execution of the texture clause, and allow the texture unit to return a pointer to the thread buffer on completion that clears the bit.

Exhibit 2037. doc R400_Sequencer.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257837

ATI Ex. 2109 IPR2023-00922 Page 127 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 20157		26 of 54

6.4 Data dependant predicate instructions

Data dependant conditionals will be supported in the R400. The only way we plan to support those is by supporting three vector/scalar predicate operations of the form:

PRED_SETE_PUSH - similar to SETE except that the result is 'exported' to the sequencer. PRED_SETNE_PUSH - similar to SETNE except that the result is 'exported' to the sequencer. PRED_SETGT_PUSH - similar to SETGT except that the result is 'exported' to the sequencer PRED_SETGTE_PUSH - similar to SETGTE except that the result is 'exported' to the sequencer

For the scalar operations only we will also support the two following instructions:

PRED_SETE PRED_SETNE PRED_SETGT PRED_SET_INV PRED_SET_POP PRED_SET_CLR PRED_SET_RESTORE

Details about actual implementation of these opcodes are in the shader pipe architectural spec.

The export is a single bit - 1 or 0 that is sent using the same data path as the MOVA instruction. The sequencer will maintain 1 set of 64 bits predicate vectors (in fact 2 sets because we interleave two programs but only 1 will be exposed) and use it to control the write masking. This predicate is maintained across clause boundaries.

Then we have two conditional execute bits. The first bit is a conditional execute "on" bit and the second bit tells us if we execute on 1 or 0. For example, the instruction:

P0_ ADD_# R0,R1,R2

Is only going to write the result of the ADD into those GPRs whose predicate bit is 0. Alternatively, P1_ADD_# would only write the results to the GPRs whose predicate bit is set. The use of the P0 or P1 without precharging the sequencer with a PRED instruction is undefined.

6.5 HW Detection of PV,PS

Because of the control program, the compiler cannot detect statically dependant instructions. In the case of nonmasked writes and subsequent reads the sequencer will insert uses of PV,PS as needed. This will be done by comparing the read address and the write address of consecutive instructions. For masked writes, the sequencer will insert detect wich channels to read from the GPRs and which ones to read from the PV/PS.

6.6 Register file indexing

Because we can have loops in fetch clause, we need to be able to index into the register file in order to retrieve the data created in a fetch clause loop and use it into an ALU clause. The instruction will include the base address for register indexing and the instruction will contain these controls:

Bit7	Bit 6	
0	0	'absolute register'
0	1	'relative register'
1	0	'previous vector'
1	1	'previous scalar'

In the case of an absolute register we just take the address as is. In the case of a relative register read we take the base address and we add to it the loop_index and this becomes our new address that we give to the shader pipe.

The sequencer is going to keep a loop index computed as such:

Index = Loop_iterator*Loop_step + Loop_start.

Exhibit 2037.docR400_Sequencer.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257838

ATI Ex. 2109 IPR2023-00922 Page 128 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 20157	GEN-CXXXXX-REVA	27 of 54

We loop until loop_iterator = loop_count. Loop_step is a signed value [-128...127]. The computed index value is a 10 bit counter that is also signed. Its real range is [-256,256]. The tenth bit is only there so that we can provide an out of range value to the "indexing logic" so that it knows when the provided index is out of range and thus can make the necessary arrangements.

6.7 Debugging the Shaders

In order to be able to debug the pixel/vertex shaders efficiently, we provide 2 methods.

6.7.1 Method 1: Debugging registers

Current plans are to expose 2 debugging, or error notification, registers:

- 1. address register where the first error occurred
- 2. count of the number of errors

The sequencer will detect the following groups of errors:

count overflow

- constant indexing overflow

register indexing overflow

Compiler recognizable errors:

jump errors

relative jump address > size of the control flow program

- call stack
 - call with stack full return with stack empty

With all the other errors, program can continue to run, potentially to worst-case limits.

If indexing outside of the constant or the register range, causing an overflow error, the hardware is specified to return the value with an index of 0. This could be exploited to generate error tokens, by reserving and initializing the 0th register (or constant) for errors.

{ISSUE : Interrupt to the driver or not?}

6.7.2 Method 2: Exporting the values in the GPRs

1) The sequencer will have a debug active, count register and an address register for this mode.

Under the normal mode execution follows the normal course.

Under the debug mode it is assumed that the program is always exporting n debug vectors and that all other exports to the SX block (but for position) will be turned off (changed into NOPs) by the sequencer (even if they occur before the address stated by the ADDR debug register).

7. Pixel Kill Mask

A vector of 64 bits is kept by the sequencer per group of pixels/vertices. Its purpose is to optimize the texture fetch requests and allow the shader pipe to kill pixels using the following instructions:

MASK_SETE MASK_SETNE MASK_SETGT MASK_SETGTE Multipass-vertex shaders (HOS)

Multipass vertex shaders are able to export from the 6 last clauses but to memory ONLY.

Exhibit 2037. docR400_Sequences.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257839

ATI Ex. 2109 IPR2023-00922 Page 129 of 326

Γ	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE	
	24 September, 2001	4 September, 20157		28 of 54	Formatted: Bullets and Numbering
	starfile allocatio			•	F ()

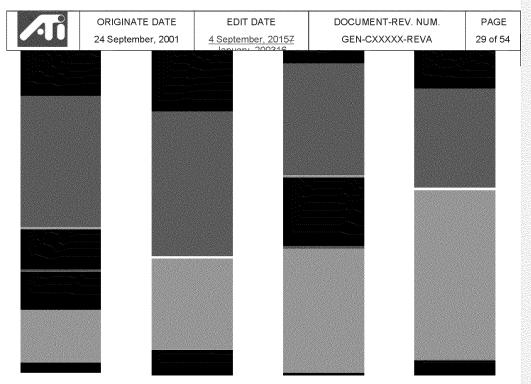
9.8. Register file allocation

The register file allocation for vertices and pixels can either be static or dynamic. In both cases, the register file in managed using two round robins (one for pixels and one for vertices). In the dynamic case the boundary between pixels and vertices is allowed to move, in the static case it is fixed to 128-VERTEX_REG_SIZE for vertices and PIXEL_REG_SIZE for pixels.

Exhibit 2037.doc.R400_Sequencer.doc 74373 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257840

ATI Ex. 2109 IPR2023-00922 Page 130 of 326



Above is an example of how the algorithm works. Vertices come in from top to bottom; pixels come in from bottom to top. Vertices are in orange and pixels in green. The blue line is the tail of the vertices and the green line is the tail of the pixels. Thus anything between the two lines is shared. When pixels meets vertices the line turns white and the boundary is static until both vertices and pixels share the same "unallocated bubble". Then the boundary is allowed to move again. The numbering of the GPRs starts from the bottom of the picture at index 0 and goes up to the top at index 127.

10.9. Fetch Arbitration

The fetch arbitration logic chooses one of the n potentially pending fetch clauses to be executed. The choice is made by looking at the Vs and Ps reservation stations and picking the first one ready to execute. Once chosen, the clause state machine will send one 2x2 fetch per clock (or 4 fetches in one clock every 4 clocks) until all the fetch instructions of the clause are sent. This means that there cannot be any dependencies between two fetches of the same clause.

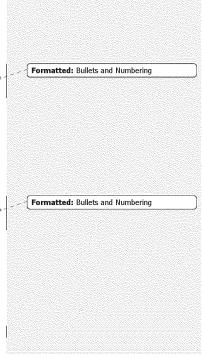
The arbitrator will not wait for the fetches to return prior to selecting another clause for execution. The fetch pipe will be able to handle up to X(?) in flight fetches and thus there can be a fair number of active clauses waiting for their fetch return data.

11.10. ALU Arbitration

ALU arbitration proceeds in almost the same way than fetch arbitration. The ALU arbitration logic chooses one of the n potentially pending ALU clauses to be executed. The choice is made by looking at the Vs and Ps reservation stations and picking the first one ready to execute. There are two ALU arbitrers, one for the even clocks and one for the odd clocks. For example, here is the sequencing of two interleaved ALU clauses (E and O stands for Even and Odd sets of 4 clocks):

Einst0 Oinst0 Einst1 Oinst1 Einst2 Oinst2 Einst0 Oinst3 Einst1 Oinst4 Einst2 Oinst0...

Exhibit 2037.docR400_Sequencer.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***



AMD1044_0257841

ATI Ex. 2109 IPR2023-00922 Page 131 of 326

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE	A Contraction Contraction
	24 September, 2001	4 September, 20157		30 of 54	10000000
D 11 11		CO 1 1 CU ALL	AL		

Proceeding this way hides the latency of 8 clocks of the ALUs. Also note that the interleaving also occurs across clause boundaries.

12.11. Handling Stalls

When the output file is full, the sequencer prevents the ALU arbitration logic from selecting the last clause (this way nothing can exit the shader pipe until there is place in the output file. If the packet is a vertex packet and the position buffer is full (POS_FULL) then the sequencer also prevents a thread from entering an exporting clause. The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU arbiter will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

13.12. Content of the reservation station FIFOs

The reservation FIFOs contain the state of the vector of pixels and vertices. We have two sets of those: one for pixels, and one for vertices. They contain 3 bits of Render State 7 bits for the base address of the GPRs, some bits for LOD correction and coverage mask information in order to fetch fetch for only valid pixels, the quad address.

14.13. The Output File

The output file is where pixels are put before they go to the RBs. The write BW to this store is 256 bits/clock. Just before this output file are staging registers with write BW 512 bits/clock and read BW 256 bits/clock. The staging registers are 4x128 (and there are 16 of those on the whole chip).

15.14. IJ Format

The IJ information sent by the PA is of this format on a per quad basis:

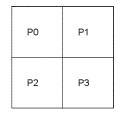
We have a vector of IJ's (one IJ per pixel at the centroid of the fragment or at the center of the pixel depending on the mode bit). All pixel's parameters are always interpolated at full 20x24 mantissa precision.

$$P0 = A + I(0) * (B - A) + J(0) * (C - A)$$

$$P1 = A + I(1) * (B - A) + J(1) * (C - A)$$

$$P2 = A + I(2) * (B - A) + J(2) * (C - A)$$

$$P3 = A + I(3) * (B - A) + J(3) * (C - A)$$



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

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

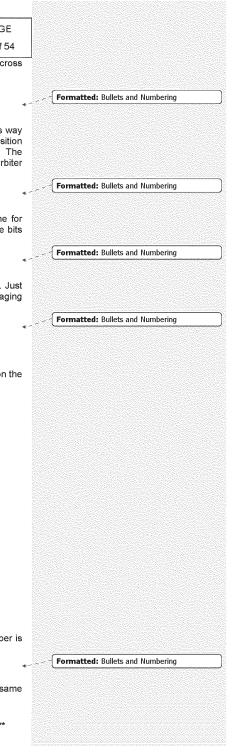
Total number of bits : 20*8 + 4*8 + 4*2 = 200.

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 +/- 1024.

15.114.1 Interpolation of constant attributes

Because of the floating point imprecision, we need to take special provisions if all the interpolated terms are the same or if two of the terms are the same.

Exhibit 2037.docR400_Sequencer.doc 74373 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***



AMD1044_0257842

ATI Ex. 2109 IPR2023-00922 Page 132 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	1
24 September, 2001	4 September, 20157	GEN-CXXXXX-REVA	31 of 54	

Formatted: Bullets and Numbering

16-15. Staging Registers

In order for the reuse of the vertices to be 14, the sequencer will have to re-order the data sent IN ORDER by the VGT for it to be aligned with the parameter cache memory arrangement. Given the following group of vertices sent by the VGT:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 || 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 || 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 || 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

The sequencer will re-arrange them in this fashion:

0 1 2 3 16 17 18 19 32 33 34 35 48 49 50 51 || 4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 || 8 9 10 11 24 25 26 27 40 41 42 43 56 57 58 59 || 12 13 14 15 28 29 30 31 44 45 46 47 60 61 62 63

The || markers show the SP divisions. In the event a shader pipe is broken, the SQ is responsible to insert padding to account for the missing pipe. For example, if SP1 is broken, vertices 4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 will not be sent by the VGT to the SQ AND the SQ is responsible to "jump" over these vertices in order for no valid vertices to be sent to an invalid SP.

The most straightforward, *non-compressed* interface method would be to convert, in the VGT, the data to 32-bit floating point prior to transmission to the VSISRs. In this scenario, the data would be transmitted to (and stored in) the VSISRs in full 32-bit floating point. This method requires three 24-bit fixed-to-float converters in the VGT. Unfortunately, it also requires and additional 3,072 bits of storage across the VSISRs. This interface is illustrated in Figure 11Figure-11Figure-11. The area of the fixed-to-float converters and the VSISRs for this method is roughly estimated as 0.759sqmm using the R300 process. The gate count estimate is shown in Figure 10Figure-40Figure-40.

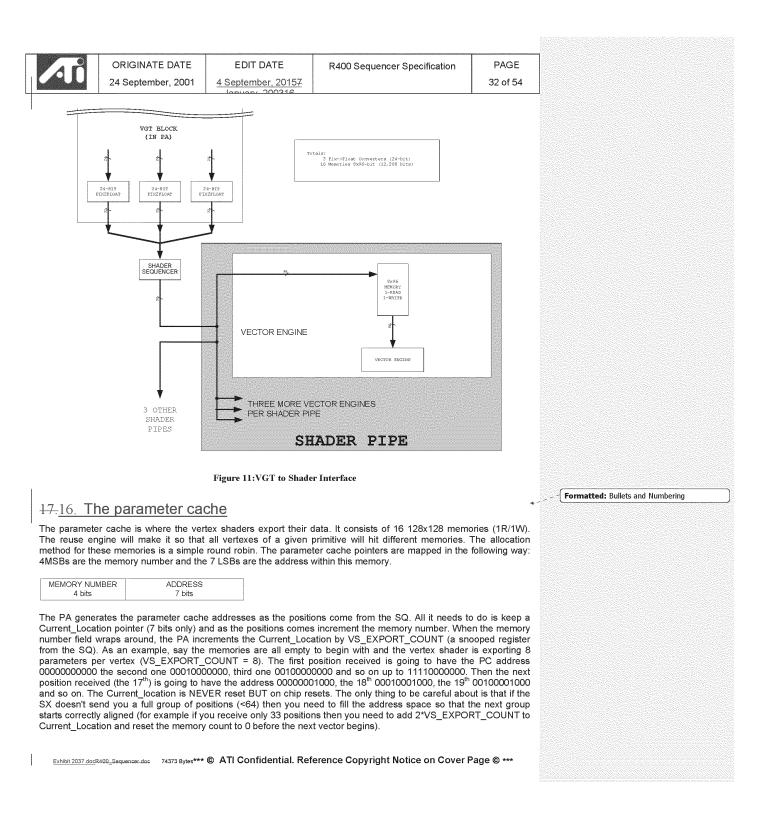
Basis for 8-deep Latch Memory (fror	n R300)		
8x24-bit	11631	μ^2	$60.57813\mu^2\text{per}$ bi
Area of 96x8-deep Latch Memory	46524	μ^2	
Area of 24-bit Fix-to-float Converter	4712	μ^2 per conve	erter
Method 1	Block	Quantity	Area
	F2F	3	14136
	8x96 Latch	16	744384
			758520 µ ²

Figure 10: Area Estimate for VGT to Shader Interface

Exhibit 2037.docR400_Sequencer.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257843

ATI Ex. 2109 IPR2023-00922 Page 133 of 326



ATI Ex. 2109 IPR2023-00922 Page 134 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE				
	24 September, 2001	4 September, 20157	GEN-CXXXXX-REVA	33 of 54	(<u></u>			
17.1 16.1	Export restriction			*	< Formatted: Bullets and Numbering			
17.1.1 16.	<u>1.1_</u> Pixel exports:							
Pixels can exp ordered to the		to the SX(+z). The exports	will be done in order. The exports w	ill always be				
<u>17.1.2</u> 16.	1.2_Vertex exports:			*	- Formatted: Bullets and Numbering			
posistion as s placed betwee	soon as possible. Position I	has to be exported in a sin cache exports can be done i	e shader program. It is always bett gle export block (no texture instruct in any order with texture instructions	ions can be				
17.1.3<u>16.</u>	1.3 Pass thru expo	rts:		*-	- Formatted: Bullets and Numbering			
Pass thru exp	orts have to be done in grou	ups of the form:		I				
	thru 5 (max export of U(ADDR) ALU(DATA) ALU		e if using EM4 alloc size 5	<u>)</u> (or_8)				
	ng to more than EMO, one s used to initialize the buffer		e write may be predicated if you do	<u>n't need the</u>				
There cannot	t be any serialize bits set (OR texture Reads between	the EA and the last EM.		- Formatted			
Memory expo	rts will be surfaced using a r	macro extension; here is wh	at needs to happen inside the macro	<u>í</u>				
The macro ne	eds to create a special cons	stant of the form:						
always repres take the 23 bit	Stream ID constant: .x = Integer that holds BaseAddressInBytes/4 in bits (29:0). Bits 31:30 should be 0b01. .y = 2**23 .z = Integer that holds register field data. Note that this data must be organized so that it always represents a 'valid' floating point number, with the relevant bits in (23 - 0); One way of doing this would be to take the 23 bits and add 2**23. .w = max index value + 2**23							
Output to EXa	address:							
	.x = Base of array (in low 30 bits)/4 .y = Index value (in low 23 bits) .z = Register Field data (in low 23 bits) .w = Max Index value (in low 23 bits)							
Also Assume	Also Assume that CO:							
	<u>.x = 0.0</u> .y = 1.0							
The Macro ex	pansion would be as follows	<u>S:</u>						
		EA = Rindex.xxxx,C0.xyxx,C EMx (x = 0 thru 4) = Rdata;	<u>streamID;</u>					
The SX will ch	neck for invalid writes and m	ask out the data so it won't	t be written to memory. Invalid writes	are:	- Formatted			
	value >= Max Index value != 0 (negative index)			•	- Formatted: Bullets and Numbering			
Exhibit 2037.doc	3400_Sequencer.doc 74373 Bytes*** @) ATI Confidential. Referen	nce Copyright Notice on Cover Pa	ge © ***				

ATI Ex. 2109 IPR2023-00922 Page 135 of 326

	DATE EDIT DATE	R400 Sequencer Specification	PAGE	
24 September	January 200216		34 of 54	
<u>3) bits [30:23] != 23 + IE</u>	EE_EXP_BIAS (127) (meaning the	index was too big to be represented us	ing 23 bits)	
They cannot have texture in ordered.	structions interleaved in the expo	t block. These exports are not guar	anteed to be	
export is used to synchronize the shader must still do either	the chip when doing a transition fro a position and PC export (if Vertex r program and thus can be used to	orted AFTER all pass thru exports. om pass thru shader to regular shader a) or a color export (if Pixel). The pass th debug. There can be any number of pa	ind vice versa iru export can	
17.216.2 Arbitration	n restrictions		alter.	
Here are the Sequencer arbitr	ration restrictions:			
 Cannot allocate positi Cannot have more that If last thread is marked thread also marked la a. Both threads b. Must turn off Cannot execute a text Cannot execute last if 		ated position y, position and Color. st and we are about to execute the sec nnot allow a first thread) scond thread ding	cond to oldest	
18.17. Export Type	20		4.1	Formatted: Bullets and Numbering
	on where the data should be put) i	s specified using the destination addre	ss field in the	
18.117.1 Vertex Sh	nading			
16:31 - Em 32 - Ext 33:37 - 5 ve 38:476 - Em 47 - Deb 48:52 - 5 de 53:59 - Em 60 - exp 61 - Em 62 - posi 63 - sprit (X=	ug Address ebug export (interpret as normal me <u>oty</u> ort addressing mode oty ition te size export that goes with positio	mory export)	it other than	
sign means VtxKill.)				
18.217.2 Pixel Sha			4 -≁	
1 - Colo 2 - Colo 3 - Colo 4:15 - Emj 16 - Buff	or for buffer 0 (primary) or for buffer 1 or for buffer 2 or for buffer 3 oty fer 0 Color/Fog (primary) fer 1 Color/Fog			
Exhibit 2037 docR400_Sequencer.doc	74373 Bytes*** © ATI Confidential. Re	ference Copyright Notice on Cover P	age © ***	

ATI Ex. 2109 IPR2023-00922 Page 136 of 326

24 18 19 20: 32: 33: 38: 47: 60 61 62: 19.18. Speci 19.18.1 Real We are unable to unded three 4 onto the register but onto th	- export addressii - Z for primary bu 63 - Empty al Interpolation al time comma 6x128 4x128 memorie s and written by type 0 o address the reatime ould be able able to the her is rasterized with.	og iltipass pixel shaders. (interpret as normal memo ng mode ffer (Z exported to 'alpha' co <u>n modes</u> ands hory since there is no way f (s) (one for each of three ver () packets, and output to the e parameter memory as w view them as two-banks-of	/	ll be mapped er and/or PA e. For higher	• Formatted: Bullets and Numbering
18 19 20: 32 33: 	Buffer 2 Color/F Buffer 3 Color/F Buffer 3 Color/F Buffer 3 Color/F Suffer 3 Color/F Empty - Export Address - 5 exports for mu - 5 export s for mu - 2 bebug Address - 5 debug exports - export addressi - 2 for primary bu - 2 for primary bu - 2 for primary bu - 3 - Empty - 2 Interpolation - 3 - Empty - 3 - Empty - 3 - Empty - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4	a lanuary 200216 og illtipass pixel shaders. (interpret as normal memo ng mode ffer (Z exported to 'alpha' co n modes ands ands hory since there is no way f as (one for each of three ver) packets, and output to the e parameter memory as w view them as two banks of	ry export) omponent) for a command stream to write into in rtices x 16 <u>4</u> interpolants). These will the parameter busses (the sequenc rell as the regular parameter store	t. Instead we li be mapped er and/or PA e. For higher	- Formatted: Bullets and Numbering
19 20: 32 33: 33: 47 48: 60 61 62: 19.18. Speci 19.18. Rea We are unable to u need to add three 4 onto the register bu	- Buffer 3 Color/F 31 - Empty - Export Address 37 - 5 exports for mu 476 - Empty - Debug Address 52 - 5 debug exports - export addressi - Z for primary bu 63 - Empty al Interpolation al time comma se the parameter men 6x128 4x128 memories s and written by type 0 o address the reatime ould be able able to -	og og iltipass pixel shaders. 6 (interpret as normal memo ng mode ffer (Z exported to 'alpha' co <u>n modes</u> ands hory since there is no way f ps (one for each of three ver p packets, and output to the e parameter memory as w view them as two banks of	or a command stream to write into it trices x 16- <u>4</u> interpolants). These wil the parameter busses (the sequenc rell as the regular parameter store	ll be mapped er and/or PA e. For higher	• Formatted: Bullets and Numbering
19.18. Speci 19.118.1 Rea We are unable to u need to add three 4 onto the register bu	al Interpolation al time comma se the parameter men 6x128 <u>4x128</u> memorie s and written by type 0 o address the reatime ould be able able to ther is rasterized with.	ands hory since there is no way f es (one for each of three ver) packets, and output to the e parameter memory as w view them as two banks of	rtices x $46 \cdot 4$ interpolants). These will the parameter busses (the sequenc rell as the regular parameter store	ll be mapped er and/or PA e. For higher	Formatted: Bullets and Numbering
19.118.1 Real We are unable to u need to add three 4 onto the register bu	al time comma se the parameter men $6x128 \frac{4x128}{4x128}$ memoria s and written by type C o address the reatime ould be able able to her is rasterized with.	ands hory since there is no way f es (one for each of three ver) packets, and output to the e parameter memory as w view them as two banks of	rtices x $46 \cdot 4$ interpolants). These will the parameter busses (the sequenc rell as the regular parameter store	ll be mapped er and/or PA e. For higher	
We are unable to u need to add three 4 onto the register bu	se the parameter men 6x128 <u>4x128</u> memorie s and written by type C address the reatime ould be able able to v her is rasterized with.	nory since there is no way f es (one for each of three ver) packets, and output to the e parameter memory as w view them as two-banks-of	rtices x $46 \cdot 4$ interpolants). These will the parameter busses (the sequenc rell as the regular parameter store	ll be mapped er and/or PA e. For higher	
need to add three 4 onto the register bu	6x128- <u>4x128</u> memorie s and written by type 0 address the reatime ould be able able to v her is rasterized with.	es (one for each of three vel) packets, and output to the e parameter memory as w view them as two banks of	rtices x $46 \cdot 4$ interpolants). These will the parameter busses (the sequenc rell as the regular parameter store	ll be mapped er and/or PA e. For higher	
performance we sh loaded, while the of be to restrict the m with this is, if we w stream to the realti address 32 vectors	iew support for 16 ve me stream), then the of parameters instead in the SX blocks. The	k64 allowing only two interp ector-4 interpolants importa PA/sequencer need to sup of 16. This mode is trigger	need 2 or 4 scalar coordinates, one- polated scalars per cycle, the only p nt (true only if we map Microsoft's port a realtime specific mode where ed by the primitive type: REAL TIME are hooked on the RBBM bus and a	option might roblem I see high priority we need to E. The actual	
<u>19.218.2</u> Spi	ites/ XY scree	en coordinates/ FE	3 information	*-	
register (in SQ) in o the faceness inform	ites may be needed ir conjunction with the SI lation (for OGL front/b des and how they inte	sible to send			
The Data is going to	be written in the regi	ster specified by the param_	_gen_pos register.		
Param_Gen_I0 disa Param_Gen_I0 ena		No modification Sign(faceness)garbage,(Sig	n Point)garbage,Sign(Line)s, t n Point)screenY,Sign(Line)s, t		
X,Y,Š,T		D, Y in GREEN, S in BLUE a	and T in ALPHA): ader use of them should use the AB	S function	
(as their sig SignX = Ba SignY = Po SignS = Lir	n bits will now be used ckFacing int Primitive	d for flags).			
If !Point & !	_ine, then it is a Poly.				
stipple and if(Y<0) {	AA for the driver would		neric texture lookup (using 3D maps)	for poly	
R =	0.0 (Point)				
Exhibit 2037.docR400_Sec	uencer.doc 74373 Bytes*** C	ATI Confidential. Refere	nce Copyright Notice on Cover Pa	ıge © ***	

ATI Ex. 2109 IPR2023-00922 Page 137 of 326

	12	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE	
1		24 September, 2001	4 September, 20157		36 of 54	
	} else	e if (S < 0) {	lanuary 200216			
1	}else	R = 1.0 (Line)				
	, , , , , , , , , , , , , , , , , , , ,	R = 2.0 (Poly)				
1	}					
	<u>19.318.3</u>	_Auto generated	counters		4~	,
'	In the cases both use this	we are dealing with multip count to write the 1 st pass	ass shaders, the sequence data to memory and the	er is going to generate a vector coun n use the count to retrieve the data or	t to be able to the 2 nd pass.	
				o the shader in a slightly different way the GEN_INDEX_PIX/VTX register. T		
	is going to k	eep two counters, one for	pixels and one for vertic	es. Every time a full vector of vertice	es or pixels is	
				ST_PIX_COUNT or RST_VTX_COUI ly one count broadcast to the GPRs.		
	hardwired to	specific values making the	index different for all elen	nents in the vector. Since the count mu d to the corresponding shader unit th	ist be different	
	choices:	venices and the 4 LODS (1	o posicions) are naruwire	to the corresponding shader unit th	e SQ has two	
	1) Main	tain a 19 bit counter that co	ounts the vectors of 64 In	this case the phase must be appende	d to the count	
		re the count is broadcast to		···· ··· p···· ·· ···		
	(Counter (19 bits)	Phase (2 bits)	Hardwired (4 bits)		
	2) Main	tain a 21 bits counter that c	ounts sub-vectors of 16. I	n this case only the counter is sent to	the Sps:	
	Cour	nter (21 bits)	Hard	vired (4 bits)		
1	10 2 1 10	21.14.7				
	19.3.118	<u>.3.1</u> Vertex shaders	S			·
1		of vertex shaders, if GEN_I ne compiler must allocate 3		ata will be put into the x field of the th rtex shader modes).	nird register (it	
	means that th				nird register (it	Formatted: Bullets and Numbering
	means that the $19.3.218$ In the case of	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver		•	Formatted: Bullets and Numbering
	means that the the second seco	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver	rtex shader modes).	•	(Formatted: Bullets and Numbering
	means that the $19.3.218$ In the case of	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver	rtex shader modes).	•	Formatted: Bullets and Numbering
	means that the $19.3.218$ In the case of	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver	rtex shader modes).	•	Formatted: Bullets and Numbering
	means that the $19.3.218$ In the case of	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver	rtex shader modes).	•	• Formatted: Bullets and Numbering
	means that the $19.3.218$ In the case of	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver	rtex shader modes).	•	• • Formatted: Bullets and Numbering
	means that the $19.3.218$ In the case of	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver	rtex shader modes).	•	Formatted: Bullets and Numbering
	means that the $19.3.218$ In the case of	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver	rtex shader modes).	•	• Formatted: Bullets and Numbering
	means that the $19.3.218$ In the case of	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver	rtex shader modes).	•	• Formatted: Bullets and Numbering
	means that the $19.3.218$ In the case of	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver	rtex shader modes).	•	Formatted: Bullets and Numbering
	means that the $19.3.218$ In the case of	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver	rtex shader modes).	•	Formatted: Bullets and Numbering
	means that the $19.3.218$ In the case of	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver	rtex shader modes).	•	Formatted: Bullets and Numbering
	means that the $19.3.218$ In the case of	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver	rtex shader modes).	•	Formatted: Bullets and Numbering
	means that the $19.3.218$ In the case of	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver	rtex shader modes).	•	Formatted: Bullets and Numbering
	means that the $19.3.218$ In the case of	ne compiler must allocate 3 . <u>3.2</u> Pixel shaders	GPRs in all multipass ver	rtex shader modes).	•	Formatted: Bullets and Numbering
	means that the 19.3.218 In the case of register.	ne compiler must allocate 3 . <u>3.2</u> <i>Pixel shaders</i> of pixel shaders, if GEN_IN	GPRs in all multipass ver	rtex shader modes). a will be put in the x field of the para	n_gen_pos+1	• • • • • • • • • • • • • • • • • •
II.	means that the 19.3.218 In the case of register.	ne compiler must allocate 3 . <u>3.2</u> <i>Pixel shaders</i> of pixel shaders, if GEN_IN	GPRs in all multipass ver	rtex shader modes).	n_gen_pos+1	Tormatted: Bullets and Numbering

ATI Ex. 2109 IPR2023-00922 Page 138 of 326

	I	1	I		
A n	ORIGINATE DATE EDIT DATE		DOCUMENT-REV. NUM.	PAGE	
	24 September, 2001	4 September, 20157	GEN-CXXXXX-REVA	37 of 54	
	AUTO COUNT	STG 0 STG1	INTERPOLATORS		
			The Auto Count Value is broadcast to all GPRs. It is loaded into a register wich has its LSBs hardwired to the GPR number (0 thru 63). Then if GEN_INDEX is high, the mux selects the auto-count value and it is loaded into the GPRs to be either used to retrieve data using the TP or sent to the SX for the RB to use it to write the data to memory		
		Figure 12: GPR input mux (Control		
20. 19. St	ate management			* 1	
	the sequencer will report the sequencer will report the last ALU claused and the last ALU claused and the sequence of the sequ		s still in the pipe. These are th	e states of the	
	Parameter cache				
In order for th sequencer wil time a vertex When the SC the count is g ready to recei	- he sequencer not to begin Il keep a 6 bit count per sta shader exports its data T sends a new vector of pixe reater than 0 before accept ive). Then the sequencer w	a group of pixels before the ate (for a total of 8 counters O THE PARAMETER CAC Is with the SC_SQ_new_ve ing the transmission (it will i <i>r</i> aits for the count to go to c	e associated group of vertices h s). These counters are initialized CHE, the corresponding pointer i ctor bit asserted, the sequencer v in fact accept the transmission bu one and decrements it. The sequ inges, the new state counter is ini	to 0 and every s incremented. vill first check if it then lower its encer can then	
21-20 X	Y Address import	*-	Formatted: Bullets and Numbering		
The SC will be buffer) with X interpolate the the GPRs. Th	e able to send the XY addr (Y writes (to the XY buffe b IJ data or pass the XY dat ne Xys are currently SCRE	ー esses to the GPRs. It does r). Then when writing the a thru a Fix→float converter	so by interleaving the writes of the data to the GPRs, the sequentian and write the converse of the values in the XY buffers this mode.	cer is going to rerted values to	
21.1 20.1	Vertex indexes ir	*-			
In order to im	- port vertex indexes, we hav). They are loaded in floating				
, ,				*	Formatted: Bullets and Numbering
22.21. Re		eb pages for register definiti	ions		
Fieds	e see the auto-yenerated w	es pages ior register dell'Illi			
Exhibit 2037.docf	3400_Sequencer.doc 74373 Bytes*** 🤅) ATI Confidential. Refere	nce Copyright Notice on Cover	Page © ***	

ATI Ex. 2109 IPR2023-00922 Page 139 of 326

ORGINATE DATE EDIT DATE R400 Sequencer Specification PAGE 24 September, 2001 4 September, 2015Z 38 of 54 23.22. Interfaces 38 of 54 23.22. Interfaces 500005 23.42.2.1 External Interfaces 500005 Whenever an x is used. it means that the bus is broadcast to all units of the same name. For example, if a bus is named SQSPx it means that SQ is going to broadcast the same information to all SP instances. 52.4.22.2. SC to SP Interfaces 23.4.22.2.1 SC _SC SP# Formatted: Bulk There is one of these interfaces at front of each of the SP (buffer to stage pixel interpolators). This interface transmits the 1/J data for pixel interpolation. For the entire system, two guads per clock are transferred to the 4 SPs, so each of these 4 interfaces transmits one fall of a quad per clock. The interface below describes a half of a quad worth of data. The set out 1/2 SQD liss which transferred over 2 clocks and therefor needs an interface 100 bits wide Additionally, XY data (12-bit unsigned fixed) is conditionally sent across this data bus over the same wires in an additional clock. The XY data is sent on the lower 24 bits of the data bus with faceness in the meb. Transfer across are sprohronized with the SC_SO 10 control Bus MAX_BUFERMINUS_2. To its before the SPs, he will increment the U_BUF_INUSE_COUNT count. Prior to sending the nacty, he so returns a pelicied pluse to decriment the count is less than MAX_BUFERMINUS_2. To its before the SPs to hol 2 double buffers of 1/J data and two buffers of X, Y data, so if etther X, Y or Centers and End Counters and the	
23-22. Interfaces Fermatted: Bulk 23-122. I External Interfaces Emanues 100332 23-122. I External Interfaces Status is broadcast to all units of the same name. For example, if a bus is named SQ-SPx it means that SQ is going to broadcast the same information to all SP instances. 23-22.2. SC to SP Interfaces Status is broadcast the same information to all SP instances. 23-22.2.2.SC to SP Interfaces Status is broadcast the same information to all SP instances. 23-2.122.1 SC_SP# There is one of these interfaces at front of each of the SP (buffer to stage pixel interpolators). This interface transmits the 1.4 data for pixel interpolation. For the entire system, two quads per dock are transitered to the 4 SPs, see each of these 4 interfaces transmits one that of a quad per clock. The interface below describes a half of a quad worth of data and which is transferred per quad is the SA 20 Floating Point 1 value '4 Ref Pix I => 54.20 Floating Point 1 value '4 Ref Pix I => 54.20 Floating Point 1 value '4 The data transfer across each of these busses is controlled by a 1_BUF_INUSE_COUNT in the SC. Each time the SC has earn a pixel vector's worth of data to the SPs, he will increment the 1_BUF_INUSE_COUNT in the SC. Each time the SC has earn a pixel vector's worth of data to the SPs, he will increment the 1_BUF_INUSE_COUNT in the SC. Each time the SC has earn a pixel vector's worth of data to the SPs, he will increment the 1_BUF_INUSE_COUNT in the SC. Cach time the SC has earn a pixel vector's worth of data to the SPs, he will increment the 1_BUF_INUSE_COUNT in the SC. Each time the SC has earn and two Buffers. In a teast	
23.122_1 External Interfaces Wherever an x is used, it means that the bus is broadcast to all units of the same name. For example, if a bus is named SQ-SPX it means that SQ is going to broadcast the same information to all SP instances. Image: Content of the set of t	ts and Numbering
Whenever an x is used, it means that the bus is broadcast to all units of the same name. For example, if a bus is named SQ_SPX it means that SQ is going to broadcast the same information to all SP instances. 23.222.2 SC to SP Interfaces 23.221.2 SC to SP interfaces 23.221.2 SC to SP interfaces 23.221.3 SC interfaces at front of each of the SP (buffer to stage pixel interpolators). This interface transmits near half of a quad worth of the actual data which is transferred per quad is Ref Pix i => S4.20 Floating Point I value '4 Ref Pix i => S4.20 Floating Point I value '4 The equates to a total of 200 bits which transferred over 2 clocks and ditionally. X/Y data (12-bit uneigned fixed) is controllous by a U_BUF_INUSE_COUNT in the SC. Each time the SC has sent a pixel vector's worth of data to the SPs, he will increment the Curit leaves than MAX_BUFER_MINUS_2, if not the SC has and therefaces to quit acity vise send work buffers of X/Y data, so if either X,Y or Centers and Zentrola we curit with so buffers. Net: We could fund work with transferred over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Yeix We ould for the exate send work buffers of X,Y data, so if either X,Y or Centers and Zentrola and the wetler besch we buffers of X,Y data, so if either X,Y or Centers and Zentrola are on, then the SC chall send 16 quads per pixel vector even if the vecto	
Nhenever an x is used, it means that the bus is broadcast to all units of the same name. For example, if a bus is name SQ_SPE it means that SQ is going to broadcast the same information to all SP instances. 23_2222_2_SC to SP Interfaces 23_2212_2_SC_to SP interfaces 23_2212_2_SC_SC to SP interfaces 23_2212_2_SC_SC to SP interfaces 23_2212_2_SC_SC to SP interfaces 23_2212_2_SC_SC to SP interfaces 23_2212_2_SC_SC SP# There is one of these interfaces at front of each of the SP (buffer to stage pixel interpolators). This interface transmits he IJ data for pixel interpolaton. For the entire system, two quads per dock are transferred to the 4 SPs, so each of these 4 interfaces transmits on half of a quad worth of data. The actual data which is transferred per quad is Ref Pix I => S4.20 Floating Point I value '4 Ref Pix I => S4.20 Floating Point I value '4 Ref Pix I => S4.20 Floating Point I value '4 The equates to a total of 200 bits which transferred over 2 clocks Interface are synchronized with the SC_S0 LUC Control Bus transfers. The data transfer across seath on the lower 24 bib of the data bus over the same wires in an idditional (x) votat is econtrol loby a U_LEF_INUSE_COUNT in the SC_ Each time the SC has a new the ourtil seles that MAX_DEFER_MINUS_2, if not to over We outdating applicint of upset of the case of only sending on U_I to use at the haddress pointers correctly at the tom bount is less that MAX_DEFER_MINUS_2, if not to over We outdate the will horder to make sume the outrul is less to that SS, SP_C and add a for over signal	
named SQ-SPR it means that SQ is going to broadcast the same information to all SP instances. Permatted: Bulk D3-222_2_SC to SP Interfaces Formatted: Bulk D3-222_2_SC to SP Interfaces Formatted: Bulk Pare is one of these interfaces at fort of each of the SP (buffer to stage pixel interpolators). This interface transmits he .J data for pixel interpolation. For the entire system, two guads per clock are transferred to the 4 SPs, so each of hese 4 interfaces transmits one half of a quad per clock. The interface below describes a half of a quad worth of lata. The actual data which is transferred per quad is RefPix J => 54.20 Floating Point I value '4 Permatted: Bulk The actual data which is transferred over 2 clocks and therefore needs an interface 100 bits wide Mile sequence to a total of 200 bits which transferred over 2 clocks and therefor needs an interface are synchronized with the SC_SQ IJ Control Bus transfers. The data transfer across each of these buseses is controlled by a J_BUF_INUSE_COUNT out. Pror to be SC and a pixel vector's worth of data to the SC_SQ IJ Locontol Bus transfers. The data transfer across each of these buseses is controlled by a J_BUF_INUSE_COUNT count. Pror to be SC and a pixel vector's worth of data to the SC and VL Buffer. The SC will stall until the SC results and two buffers of X,Y data, so if either X,Y or Centers and cantrols are on, then the SC can seed two Buffers. The data transfer across each of these buseses to control bus transfers. The data transfer across each of the case of Only sending on buffer SC, and the buffer so I, J data and	
H-222.2_SC to SP Interfaces Permatted: Bulk Part is one of these interfaces at front of each of the SP (buffer to stage pixel interpolators). This interface transmits he i, J data for pixel interpolation. For the entire system, two quads per clock are transferred to the 4 SPs, so each of hese 4 interfaces transmits one half of a quad per clock. The interface below describes a half of a quad worth of lata. The eactual data which is transferred per quad is Ref Pix I => S4.20 Floating Point I value '4 The equates to a total of 200 bits which transferred over 2 clocks ind therefor needs an interface 100 bits wide Value '4 The equates to a total of 200 bits which transferred over 2 clocks ind therefor needs an interface 100 bits wide Value '4 The equates to a total of 200 bits which transferred over 2 clocks ind therefor needs an interface 100 bits wide Value 2.0 Control Bus transfers Transfers across these interfaces are synchronized with the SC_SQ U Control Bus transfers. The data transfer across each of these buseses is controlled by a U_BUF_INUSE_COUNT out. Third to be control with the SO relums a pipelined plate bot dearment the count when the has scheduled a buffer free. tots: We couldmay optimize to the case of only sending only 1b use all the buffers of X.Y data, so if either X.Y or Centers and entrols are on the SD relums a pipelined plate buffers of X.Y data, so if either X.Y or Centers and cantrols are on, then the SC can send two Buffers. n at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will reterment buffer write address pointers correc	
23.2.122.1_SC_SP# There is one of these interfaces at front of each of the SP (buffer to stage pixel interpolations). This interface transmits one half of a quad per clock. The interface below describes a half of a quad worth of lata. The actual data which is transferred per quad is Ref Pix 1=> 54.20 Floating Point 1 value '4 Ref Pix 1=> 54.20 Floating Point 1=> 71.20 Floating Poi	ts and Numbering
23-22-12_2_1_SC_SP# There is one of these interfaces at front of each of the SP (buffer to stage pixel interpolators). This interface transmits the 1, J data for typice interpolation. For the entire system, two quads per clock are transferred to the 4 SPs, so each of hese 4 interfaces transmits one half of a quad per clock. The interface below describes a half of a quad worth of lata. The actual data which is transferred per quad is Ref Pix J => S4.20 Floating Point J value '4 Ref Pix J => S4.20 Floating Point J value '4 Ref Pix J => S4.20 Floating Point J value '4 This equates to a total of 200 bits which transferred over 2 clocks in the therefore needs an interface 100 bits wide Note the same wires in an indiverse in the lower 24 bits of the data bus with faceness in the mesb. Transfers across these interfaces are synchronized with the SC_SQ IJ Control Bus transfers. The data transfer across each of these busses is controlled by a J_BUF_INUSE_COLNT count. Prior to ending the next pixel vectors data, he will check to make sure the count when the has scheduled a buffer free. Note: Sculemas a pixeline public J double buffers of J data and two buffers of X,Y data, so if either X,Y or Centers and 2entrolids are on, then the SC can send two Buffers. na t least the initial version, the SC shall send 16 quads per pixel vector avent if the vector is not full. This will thereface that and the solution ally set of the signal of the SP by hold 2 double buffers of J, data and two buffers of X,Y data, so if either X,Y or Centers and 2entrolids are on, then the SC shall send 16 quads per pixel vector avent if the vector is not full. This will thereface transmite Signals should cause a partial vector and	
There is one of these interfaces at front of each of the SP (buffer to stage pixel interpolators). This interface transmits he I.J data for pixel interpolation. For the entire system, two quads per clock are transferred to the 4 SPs, so each of heas 4 interfaces transmits one half of a quad per clock. The interface below describes a half of a quad worth of lata. The actual data which is transferred per quad is Ref Pix I => S4.20 Floating Point I value *4 Ref Pix I => S4.20 Floating Point J value *4 This equates to a total of 200 bits which transferred over 2 clocks In therefor needs an interface 100 bits wide value *4 This equates to a total of 200 bits which transferred over 2 clocks interpolation. For the other 2 bits of the data bus with faceness in the mab. Transfers across these interfaces are synchronized with the SC_SQ JI Control Bus transfers. The data transfer across these interfaces are synchronized with the SC_SQ JI Control Bus transfers. The data transfer across data, he will check to make sure the count Max BUFER_MINUSE_COUNT count. Prior to tending the next pixel vector's worth of data to the SPs, he will increment the L_BUF_INUSE_COUNT count. Prior to the SC will stall until the SQ returns a pipelined pulse to decrement the count when he has scheduled a buffer free. Viet: We could/may optimize for the case of only sending only JI to use all the buffers to pre-load more. Currently is planned for the SP to hold 2 double buffers of 1, J data and two buffers of X,Y data, so if either X,Y or Centers and 2 banfor three. Network we vector signals should cause a partial vector even if the vector is not full. This will normer th buffe	ts and Numbering
he I.J data for pixel interpolation. For the entire system, two quads per clock are transferred to the 4 SPs, so each of hease 4 interfaces transmits one half of a quad per clock. The interface below describes a half of a quad worth of lata. hee actual data which is transferred per quad is Ref Pix I => S4.20 Floating Point I value *4 Ref Pix I => S4.20 Floating Point J value *4 Ref Pix I => S4.20 Floating Point J value *4 This equates to a total of 200 bits which transferred over 2 clocks ind therefor needs an interface 100 bits wide Note that the second the secon	
iata The actual data which is transferred per quad is Ref Pix I => S4.20 Floating Point I value *4 Ref Pix J => S4.20 Floating Point I value *4 This equates to a total of 200 bits which transferred over 2 clocks ind therefor needs an interface 100 bits wide Validitionally, X,Y data (12-bit unsigned fixed) is conditionally sent across this data bus over the same wires in an idditional clock. The X,Y data is sent on the lower 24 bits of the data bus with faceness in the msb. Transfers across these interfaces are synchronized with the SC_SQ IJ Control Bus transfers. The data transfer across each of these busses is controlled by a JJ_BUF_INUSE_COUNT in the SC. Each time the SC will stall until the SQ returns a pipelined pulse to decrement the LJ_BUF_INUSE_COUNT count. Prior to rending the next pixel vector's worth of data to the SPs, he will increment the LJ_BUF_INUSE_COUNT count. Prior to rending the next pixel vector's data, he will check to make sure the count when he has scheduled a buffer free. Vote: We could/may optimize for the case of only sending only U to use all the buffers to pre-load more. Currently is planned for the SP to hold 2 double buffers of L, J data and two buffers of X, Y data, so if either X, Y or Centers and 2entroids are on, then the SC shall send 16 quads per pixel vector even if the vector is not full. This will reformance htuffer write address pointers correctly all the time. (We may revisit this for both the SX,SP SQ and add a findOVector signal on all interfaces to qui early. We opted for the simple mode first with a belief that only the end of facket and multiple new vector signals should cause a partial vector and that this would not really be significant verformats E4MA20 SE4MA20 SE4MA20 SE4MA20 SE4MA20 Type 2 Siss [22,3] [24] [23:12] [11:	
Ref Pix I => 54.20 Floating Point J value *4 Ref Pix J => 54.20 Floating Point J value *4 This equates to a total of 200 bits which transferred over 2 clocks and therefor needs an interface 100 bits wide Additionally, X,Y data (12-bit unsigned fixed) is conditionally sent across this data bus over the same wires in an additional clock. The X,Y data is sent on the lower 24 bits of the data bus with faceness in the msb. Fransfers across these interfaces are synchronized with the SC_SQ IJ Control Bus transfers. Ch data transfer across each of these busses is controlled by a JJ_BUF_INUSE_COUNT count. Prior to sending the next pixel vectors data, he will check to make sure the count is less than MAX_BUFER_MINUS_2, if not he SC will stall until the SQ returns a pipelined pulse to decrement the LJ_BUF_INUSE_COUNT count. Prior to sending the next pixel vectors data, he will check to make sure the count when he has scheduled a buffer free. Note: We could/may optimize for the case of only sending only 1b to use all the buffers to pre-load more. Currently is planned for the SP to hold 2 double buffers of 1,J data and two buffers of X,Y data, so if either X,Y or Centers and Zentroids are on, then the SC can send two Buffers. n at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will necrement buffer write address pointers correctly all the time. (We may revisit this for both the SX,SP,SQ and add andOtvector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of acaket and multiple new vector signals should cause a partial vector and that this would not really be significant performance hit.) Name Bits Description Format SE4MAVO SE4MA2O SE4MA2O SE4MA2O	
This equates to a total of 200 bits which transferred over 2 clocks Indit therefor needs an interface 100 bits wide Additional clock. The X,Y data is sent on the lower 24 bits of the data bus with faceness in the meb. Transfers across these interfaces are synchronized with the SC_SQ IJ Control Bus transfers. The data transfer across each of these busses is controlled by a IJ_BUF_INUSE_COUNT in the SC. Each time the SC has sent a pixel vector's worth of data to the SPs, he will increment the IJ_BUF_INUSE_COUNT count. Prior to sending the next pixel vector's worth of data to the SPs, he will increment the COUNT when he has scheduled a buffer free. Note: We could/may optimize for the case of only sending only IJ to use all the buffers to pre-load more. Currently is planned for the SP to hold 2 double buffers of I,J data and two buffers of X,Y data, so if either X,Y or Centers and Centroids are on, then the SC can send two Buffers. In at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will norement buffer write address pointers correctly all the time. (We may revisit this for both the SX,SP,SQ and add a EndON/ector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of acket and multiple new vector signals should cause a partial vector and that this would not really be significant verformance hit.) Name Bits Description SC_SP#_data 100 I information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 2 Field ULC URC LLC LRC Bits [53.39] [38.26] [25.13] [12:0] Format SE4MAVO. SE4MA20. SE4MA20 Type 2 Field Face X Y Bits [24] [23:12] [11:0] Format SE4MAVO. SE4MA20. SE4MA20 SE4MA20. SE4MA20. SE4MA20. SE4MA20. SE4M	
Additionally, X,Y data (12-bit unsigned fixed) is conditionally sent across this data bus over the same wires in an idditional clock. The X,Y data is sent on the lower 24 bits of the data bus with faceness in the msb. Transfers across these interfaces are synchronized with the SC_SQ IJ Control Bus transfers. The data transfer across each of these busses is controlled by a IJ_BUF_INUSE_COUNT in the SC. Each time the SC has sent a pixel vector's worth of data to the SPs, he will increment the IJ_BUF_INUSE_COUNT count. Prior to the ending the next pixel vectors data, he will check to make sure the count is less than MAX_BUFER_MINUS_2, if not he SC will stall until the SQ returns a pipelined pulse to decrement the count when he has scheduled a buffer free. Oit: We could/may optimize for the case of only sending only IJ to use all the buffers to pre-load more. Currently is planned for the SP to hold 2 double buffers of I,J data and two buffers of X,Y data, so if either X,Y or Centers and zentroids are on, then the SC can send two Buffers. In at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will normative sourced will the time. (We may revisit this for both the SX,SP,SQ and add a indOfVector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of vacket and multiple new vector signals should cause a partial vector and that this would not really be significant verformance hit.) Name Bits Description SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J Field Field	
Additionally, X,Y data (12-bit unsigned fixed) is conditionally sent across this data bus over the same wires in an idditional clock. The X,Y data is sent on the lower 24 bits of the data bus with faceness in the msb. Transfers across these interfaces are synchronized with the SC_SQ IJ Control Bus transfers. The data transfer across each of these busses is controlled by a IJ_BUF_INUSE_COUNT in the SC. Each time the SC has sent a pixel vector's worth of data to the SPs, he will increment the IJ_BUF_INUSE_COUNT count. Prior to ending the next pixel vectors data, he will check to make sure the count is less than MAX_BUFER_MINUS_2, if not he SC will stall until the SQ returns a pipelined pulse to decrement the count when he has scheduled a buffer free. Net Will the SQ returns a pipelined pulse to decrement the count is less than MAX_BUFER_MINUS_2, if not he SQ will stall until the SQ returns a pipelined pulse to decrement the count when he has scheduled a buffer free. Net Will the SQ returns a pipelined pulse to decrement the count is less than MAX_BUFER_MINUS_2, if not he SQ to hold 2 double buffers of I,J data and two buffers of X,Y data, so if either X,Y or Centers and 2-entroids are on, then the SC can send two Buffers. n at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will ind/OVector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of racket and multiple new vector signals should cause a partial vector and that this would not really be significant serformance hit.) Name Bits Description SC_SP#_data 100 I information sent over 2 clocks (or X,Y in 24 LSBs with faceness	
Idditional clock. The X,Y data is sent on the lower 24 bits of the data bus with faceness in the msb. Transfers across these interfaces are synchronized with the SC_SQ IJ Control Bus transfers. The data transfer across each of these busses is controlled by a IJ_BUF_INUSE_COUNT in the SC. Each time the SC has ent a pixel vector's worth of data to the SPs, he will increment the IJ_BUF_INUSE_COUNT count. Prior to brending the next pixel vectors data, he will check to make sure the count is less than MAX_BUFER_MINUS_2, if not the SC will stall until the SQ returns a pipelined pulse to decrement the count when he has scheduled a buffer free. Note: We could/may optimize for the case of only sending only IJ to use all the buffers to pre-load more. Currently is planned for the SP to hold 2 double buffers of I.J data and two buffers of X,Y data, so if either X,Y or Centers and 2entroids are on, then the SC shall send 16 quads per pixel vector even if the vector is not full. This will norement buffer write address pointers correctly all the time. (We may revisit this for both the SX,SP,SQ and add a indO/Vector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of backet and multiple new vector signals should cause a partial vector and that this would not really be significant thereforemance hit.) Name Bits Description SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J Field ULC LRC Bits Description SC_SP#_121_[11:0] Format SE4M20 SE4M20 Field <	
Transfers across these interfaces are synchronized with the SC_SQ IJ Control Bus transfers. The data transfer across each of these busses is controlled by a IJ_BUF_INUSE_COUNT in the SC. Each time the SC has sent a pixel vector's worth of data to the SPs, he will increment the IJ_BUF_INUSE_COUNT count. Prior to sending the next pixel vectors data, he will check to make sure the count is less than MAX_BUFER_MINUS_2, if not the SC will stall until the SQ returns a pipelined pulse to decrement the count when he has scheduled a buffer free. Note: We could/may optimize for the case of only sending only IJ to use all the buffers to pre-load more. Currently to is planned for the SP to hold 2 double buffers of I, J data and two buffers of X,Y data, so if either X,Y or Centers and Centroids are on, then the SC can send two Buffers. n at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will norement buffer write address pointers correctly all the time. (We may revisit this for both the SX,SP,SQ and add a EndOfVector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of packet and multiple new vector signals should cause a partial vector and that this would not really be significant vectormance hit.) Name Bits Description SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J Field Hace X Y Heid SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J	
SC has sent a pixel vector's worth of data to the SPs, he will increment the IJ_BUF_INUSE_COUNT count. Prior to sending the next pixel vectors data, he will check to make sure the count is less than MAX_BUFER_MINUS_2, if not he SC will stall until the SQ returns a pipelined pulse to decrement the count when he has scheduled a buffer free. Note: We could/may optimize for the case of only sending only IJ to use all the buffers to pre-load more. Currently is planned for the SP to hold 2 double buffers of I,J data and two buffers of X,Y data, so if either X,Y or Centers and Centroids are on, then the SC can send two Buffers. n at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will norement buffer write address pointers correctly all the time. (We may revisit this for both the SX,SP,SQ and add a EndOfVector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of packet and multiple new vector signals should cause a partial vector and that this would not really be significant serformance hit.) Name Bits Description SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J Field ULC URC LLC LRC Bits [63:39] [38:26] [25:13] [12:0] Format Bits [24] [23:12] [11:0] Type 2 Field Face X Y Bits [24] [23:12] [11:0] <td></td>	
sending the next pixel vectors data, he will check to make sure the count is less than MAX_BUFER_MINUS_2, if not he SC will stall until the SQ returns a pipelined pulse to decrement the count when he has scheduled a buffer free. Note: We could/may optimize for the case of only sending only J to use all the buffers to pre-load more. Currently t is planned for the SP to hold 2 double buffers of 1,J data and two buffers of X,Y data, so if either X,Y or Centers and Centroids are on, then the SC can send two Buffers. In at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will nerement buffer write address pointers correctly all the time. (We may revisit this for both the SX,SP,SQ and add a EndOfVector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of packet and multiple new vector signals should cause a partial vector and that this would not really be significant berformance hit.) Name Bits Description SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock 1, second clk J Field ULC URC LLC LRC Bits [63:39] [38:26] [23:13] [12:0] Format SE4M20 SE4M20 SE4M20 Type 2 Field Face X Y Bits [24] [23:12] [11:0] Format Bit Unsigned Unsigned SC_SP#_valid 1 Valid SC_SP#_last_quad_data 1 This bit will be set on the last transfer of data per quad. SC_SP#_type 2 0 -> Indicates centers	
he SC will stall until the SQ returns a pipelined pulse to decrement the count when he has scheduled a buffer free. Note: We could/may optimize for the case of only sending only IJ to use all the buffers to pre-load more. Currently tis planned for the SP to hold 2 double buffers of I,J data and two buffers of X,Y data, so if either X,Y or Centers and Centroids are on, then the SC can send two Buffers. n at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will nerement buffer write address pointers correctly all the time. (We may revisit this for both the SX,SP,SQ and add a EndOfVector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of packet and multiple new vector signals should cause a partial vector and that this would not really be significant beerformance hit.) Name Bits Description SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J Field ULC URC LRC Bits Description SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J Field ULC URC LRC Bits [23:13] [12:0] Format El4M20 SE4M20 SE4M20 Type 2 Field Face <td< td=""><td></td></td<>	
t is planned for the SP to hold 2 double buffers of I,J data and two buffers of X,Y data, so if either X,Y or Centers and Centroids are on, then the SC can send two Buffers. n at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will ncrement buffer write address pointers correctly all the time. (We may revisit this for both the SX,SP,SQ and add a EndOVector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of backet and multiple new vector signals should cause a partial vector and that this would not really be significant Name Bits Description SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J Field ULC URC LLC LRC Bits [63:39] [38:26] [25:13] [12:0] Format SE4M20 SE4M20 SE4M20 SE4M20 SE4M20 SE4M20 SC_SP#_valid 1 Valid SC_SP#_valid 1 Valid SC_SP#_valid 1 Valid SC_SP#_last_quad_data 1 This bit will be set on the last transfer of data per quad. SC_SP#_type 2 0 -> Indicates centers	
Centroids are on, then the SC can send two Buffers. n at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will nerement buffer write address pointers correctly all the time. (We may revisit this for both the SX,SP,SQ and add a EndOfVector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of backet and multiple new vector signals should cause a partial vector and that this would not really be significant berformance hit.) Name Bits Description SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J Field ULC URC LLC Bits [63:39] [38:26] [25:13] Field ULC URC LRC Bits [63:39] [38:26] [25:13] Field Field [12:0] Format Bit [13:17] Field Face X Y Bits [24] [23:12] Field Face X Y Bits [24] [23:12] [11:0] Format Bit Unsigned Unsigned SC_SP#_valid 1 Valid	
Increment buffer write address pointers correctly all the time. (We may revisit this for both the SX,SP,SQ and add a EndOfVector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of packet and multiple new vector signals should cause a partial vector and that this would not really be significant beerformance hit.) Name Bits Description SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J Field ULC Field ULC URC LLC Bits [63:39] [38:26] [25:13] Field Face X Y Bits [24] [23:12] [11:0] Format Bit Unsigned Unsigned SC_SP#_valid 1 Valid Valid SC_SP#_valid 1 Valid Sc_SP#_uast_quad_data 1 Valid 2 0 -> Indicates centroids 1 -> Indicates centers	
EndOfVector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of backet and multiple new vector signals should cause a partial vector and that this would not really be significant berformance hit.) Name Bits Description SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J Field ULC URC LLC LRC Bits [63:39] [38:26] [25:13] [12:0] Format SE4M20 SE4M20 SE4M20 SE4M20 SC_SP#_valid 1 Valid Valid Sc_SP#_valid 1 SC_SP#_valid 1 Valid Unsigned Unsigned Sc_SP#_type 2 0 -> Indicates centroids 1 -> Indicates centers 1 -> Indicates centers	
Description Name Bits Description SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J Field ULC Bits 63:39] [28:26] [25:13] Field ULC URC LLC Bits 63:39] [28:26] [25:13] Format SE4M20 SE4M20 SE4M20 SC_SP#_valid 1 Valid SC_SP#_valid 1 Valid SC_SP#_last_quad_data 1 This bit will be set on the last transfer of data per quad. SC_SP#_type 2 0 -> Indicates centroids 1 -> Indicates centers 1 -> Indicates centers	
Name Bits Description SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J Field ULC URC Bits [63:39] [38:26] [25:13] [12:0] Format SE 4M20 SE 4M20 SE 4M20 SE 4M20 Type 2 Field Face X Y Bits [24] [23:12] [11:0] Format Bit Unsigned Unsigned SC_SP#_valid 1 Valid Valid SC_SP#_last_quad_data 1 This bit will be set on the last transfer of data per quad. SC_SP#_type 2 0 -> Indicates centroids 1 -> Indicates centers	
SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J Field ULC URC LLC LRC Bits [63:39] [38:26] [25:13] [12:0] Format SE4M20 SE4M20 SE4M20 Type 2 Field Face X Y Bits [23:12] [11:0] Format Bit Unsigned Unsigned Unsigned Unsigned SC_SP#_valid 1 Valid Valid 0 -> Indicates centroids 0 -> Indicates centroids SC_SP#_type 2 0 -> Indicates centroids 1 -> Indicates centroids 1 -> Indicates centroids	
SC_SP#_data 100 IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit) Type 0 or 1, First clock I, second clk J Field ULC URC LLC LRC Bits [63:39] [38:26] [25:13] [12:0] Format SE4M20 SE4M20 SE4M20 Type 2 Field Face X Y Bits [23:12] [11:0] Format Bit Unsigned Unsigned Unsigned Unsigned SC_SP#_valid 1 Valid Valid 0 -> Indicates centroids 0 -> Indicates centroids SC_SP#_type 2 0 -> Indicates centroids 1 -> Indicates centroids 1 -> Indicates centroids	
Field ULC URC LRC Bits [63:39] [38:26] [25:13] [12:0] Format SE4M20 SE4M20 SE4M20 SE4M20 Type 2 Field Face X Y Bits [24] [23:12] [11:0] Format Bit Unsigned Unsigned SC_SP#_valid 1 Valid Valid SC_SP#_last_quad_data 1 This bit will be set on the last transfer of data per quad. SC_SP#_type 2 0 -> Indicates centroids 1 -> Indicates centers	
Bits [63:39] [38:26] [25:13] [12:0] Format SE4M20 SE4M20 SE4M20 SE4M20 Type 2 Field Face X Y Bits [24] [23:12] [11:0] Format Bit Unsigned Unsigned SC_SP#_valid 1 Valid Valid SC_SP#_last_quad_data 1 This bit will be set on the last transfer of data per quad. SC_SP#_type 2 0 -> Indicates centroids 1 1 > Indicates centers 1	
Type 2 Field Face X Y Bits [24] [23:12] [11:0] Format Bit Unsigned Unsigned SC_SP#_valid 1 Valid SC_SP#_last_quad_data 1 This bit will be set on the last transfer of data per quad. SC_SP#_type 2 0 -> Indicates centroids 1 -> Indicates centers	
Field Face X Y Bits [24] [23:12] [11:0] Format Bit Unsigned Unsigned SC_SP#_valid 1 Valid SC_SP#_last_quad_data 1 This bit will be set on the last transfer of data per quad. SC_SP#_type 2 0 -> Indicates centroids 1 -> Indicates centers	
Format Bit Unsigned SC_SP#_valid 1 Valid SC_SP#_last_quad_data 1 This bit will be set on the last transfer of data per quad. SC_SP#_type 2 0 -> Indicates centroids 1 -> Indicates centers	
SC_SP#_last_quad_data 1 This bit will be set on the last transfer of data per quad. SC_SP#_type 2 0 -> Indicates centroids 1 -> Indicates centers	
SC_SP#_last_quad_data 1 This bit will be set on the last transfer of data per quad. SC_SP#_type 2 0 -> Indicates centroids 1 -> Indicates centers	
1 -> Indicates centers	
2 -> Indicates X,Y Data and faceness on data bus	
The SC shall look at state data to determine how many types to send for the	
Exhibit 2037. doc R400_Sequences.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***	

ATI Ex. 2109 IPR2023-00922 Page 140 of 326

	ORIGINATE DATE 24 September, 2001	EDIT DATE <u>4 September, 20157</u> Japung, 200316	DOCUMENT-REV. NUM. GEN-CXXXXX-REVA	PAGE 39 of 54
	i	nterpolation process.		

The # is included for clarity in the spec and will be replaced with a prefix of u#_ in the verilog module statement for the SC and the SP block will have neither because the instantiation will insert the prefix.

23.2.222.2.2 SC_SQ

This is the control information sent to the sequencer in order to synchronize and control the interpolation and/or loading data into the GPRs needed to execute a shader program on the sent pixels. This data will be sent over two clocks per transfer with 1 to 16 transfers. Therefore the bus (approx 108 bits) could be folded in half to approx 54 bits.

Name	Bits	Description
Name SC_SQ_data	Bits 46	Description Control Data sent to the SQ 1 clk transfers Event - valid data consist of event_id and state_id. Instruct SQ to post an event vector to send state id and event_id through request fifo and onto the reservation stations making sure state id and/or event_id gets back to the CP. Events only follow end of packets so no pixel vectors will be in progress.
		Empty Quad Mask – Transfer Control data consisting of pc_dealloc or new_vector. Receipt of this is to transfer pc_dealloc or new_vector without any valid quad data. New vector will always be posted to request fifo and pc_dealloc will be attached to any pixel vector outstanding or posted in request fifo if no valid quad outstanding.
		2 clk transfers Quad Data Valid – Sending quad data with or without new_vector or pc_dealloc. New vector will be posted to request fifo with or without a pixel vector and pc_dealloc will be posted with a pixel vector unless none is in progress. In this case the pc_dealloc will be posted in the request queue. Filler quads will be transferred with The Quad mask set but the pixel corresponding pixel mask set to zero.
SC_SQ_valid	1	SC sending valid data, 2 nd clk could be all zeroes

SC_SQ_data – first clock and second clock transfers are shown in the table below.

Name	BitField	Bits	Description
1 st Clock Transfer			
SC_SQ_event	0	1	This transfer is a 1 clock event vector Force quad_mask =
			new_vector=pc_dealloc=0
SC_SQ_event_id	[5:1]	4	This field identifies the event 0 => denotes an End Of State Event 1

Exhibit 2037.docR400_Sequences.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257851

Formatted: Bullets and Numbering

ATI Ex. 2109 IPR2023-00922 Page 141 of 326

Γ		ORIGINATE I	DATE	EI	DIT DATE	R400 Sequencer Specification	PAGE
		24 September	, 2001		ember, 20157		40 of 54
Η			<u>_</u>	T Tan	=> TBD	0.000 000 000 000 000 000 000 000 000 0	
	SC SQ state	id	[8:6]	3	State/constant	pointer (6*3+3)	
	SC SQ pc de		[11:9]	3		en for the Parameter Cache	
	SC SQ new		12	1		ait for Vertex shader done count > 0 a	ind after
					dispatching the	Pixel Vector the SQ will decrement th	e count.
	SC_SQ_quad	mask	[16:13]	4	Quad Write ma	sk left to right SP0 => SP3	
	SC_SQ_end_of_prim 17 1 End Of the primitive						
	SC_SQ_pix_m	nask	[33:18]	16	Valid bits for all	pixels SP0=>SP3 (UL,UR,LL,LR)	
	SC SQ provo	k_vtx	[35:34]	2	Provoking verte	x for flat shading	
	SC_SQ_lod_correct_0 [44:36] 9 LOD correction for quad 0 (SP0) (9 bits per quad)						
	SC_SQ_lod_c	orrect_1	[53:45] 9 LOD correction for guad 1 (SP1) (9 bits per guad)				
]							
	2nd Clock Tra	nsfer					
]	SC_SQ_lod_c	orrect_2	[8:0]	9	LOD correction	for quad 2 (SP2) (9 bits per quad)	
	SC_SQ_lod_c	1000	[17:9]	9		for quad 3 (SP3) (9 bits per quad)	
	SC_SQ_pc_pt	rO	[28:18]	11		ne pointer for vertex 0	
	SC_SQ_pc_pt	r1	[39:29]	11	Parameter Cacl	ne pointer for vertex 1	
	SC_SQ_pc_pt	r2	[50:40]	11	Parameter Cac	ne pointer for vertex 2	
	SC_SQ_prim_	type	[53:51]	3		d Real time command need to load te	x cords from
					alternate buffer		
					000: Sprite (poi	nt)	
					001: Line		
					010: Tri_rect		
					100: Realtime S		
					101: Realtime L		
				J	110: Realtime T	ri_rect	

Name	Bits	Description
SQ_SC_free_buff	1	Pipelined bit that instructs SC to decrement count of buffers in use.
SQ_SC_dec_cntr_cnt	1	Pipelined bit that instructs SC to decrement count of new vector and/or event
		sent to prevent SC from overflowing SQ interpolator/Reservation request fifo.

The scan converter will submit a partial vector whenever:

1.) He gets a primitive marked with an end of packet signal.

2.) A current pixel vector is being assembled with at least one or more valid quads and the vector has been marked for deallocate when a primitive marked new_vector arrives. The Scan Converter will submit a partial vector (up to 16quads with zero pixel mask to fill out the vector) prior to submitting the new_vector marker\primitive.

(This will prevent a hang which can be demonstrated when all primitives in a packet three vectors are culled except for a one quad primitive that gets marked pc_dealloc (vertices maximum size). In this case two new_vectors are submitted and processed, but then one valid quad with the pc_dealloc creates a vector and then the new would wait for another vertex vector to be processed, but the one being waited for could never export until the pc_dealloc signal made it through and thus the hang.)

Exhibit 2037. doc/R400_Sequesn.cer.doc 74373 Bytes**** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257852

ATI Ex. 2109 IPR2023-00922 Page 142 of 326

ORIGINATE	DATE	EDIT	DATE	Ξ [DOCUMENT-REV. NUM.	PAGE	
24 Septembe	24 September, 2001		4 September, 20		GEN-CXXXXX-REVA	41 of 54	
23.2.322.2.3 SQ to S.	X(SP): II					4	
Name	Direction	,	Bits	Descrip	tion		
SQ_SPx_interp_flat_vtx	SQ→SPx		2		ng vertex for flat shading		
SQ_SPx_interp_flat_gouraud	SQ→SPx		1		jouraud shading	1	
SQ_SPx_interp_cyl_wrap SQ_SPx_interp_param_gen	SQ→SPx SQ→SPx		4		annel needs to be cylindrical wrapp te Parameter	ea	
SQ_SPx_interp_prim_type	SQ→SPx		2)] of primitive type sent by SC		
SQ_SPx_interp_buff_swap	SQ→SPx		1		IJ buffers		
SQ_SPx_interp_IJ_line	SQ→SPx		2	IJ line n			
SQ_SPx_interp_mode	SQ→SPx		1		Centroid sampling		
SQ_SXx_pc_ptr0 SQ_SXx_pc_ptr1	SQ→SXx SQ→SXx		11 11		ter Cache Pointer		
SQ SXx pc ptr2	SQ→SXx		11		ter Cache Pointer		
SQ_SXx_rt_sel	SQ→SXx		1		between RT and Normal data (Bit 2 o	f prim type)	
SQ_SX0_pc_wr_en	SQ→SX0		8		able for the PC memories		
SQ_SX1_pc_wr_en SQ_SXx_pc_wr_addr	SQ→SX1 SQ→SXx		8		able for the PC memories Idress for the PCs		
SQ SXx pc channel mask	SQ→SXX		4	Channe			
SQ SXx pc ptr valid	SQ→SXx		1		pinters are valid.		
SQ_SPx_interp_valid	SQ→SPx		1		ation control valid		
22 2 422 2 4 SO to S	D. Staniu	na Donio	tor C	lata		4	(Formatted: Bullets and Numbering
23.2.422.2.4 SQ to S		• •					
This is a broadcast bus that ser	nds the VSI		tion to				
Name		Direction SQ→SPx		Bits 96	Description Pointers of indexes or HOS surface	information	
SQ_SPx_vsr_data SQ_SPx_vsr_wrt_addr		SQ→SPX SQ→SPX		3	Staging register write address	mormation	
SQ SPx vsr rd addrSQ SPx	vsr double	SQ→SPx		31	Staging register read address0: No	rmal 96 bits	
					per vert 1: double 192 bits per vert		
SQ_SP0_vsr_valid		SQ→SP0		1	Data is valid		
SQ_SP1_vsr_valid		SQ→SP1		1	Data is valid		
SQ_SP1_vsr_valid SQ_SP2_vsr_valid		SQ→SP1 SQ→SP2			Data is valid Data is valid		
SQ_SP1_vsr_valid		SQ→SP1		1	Data is valid		
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read	SO : Va	SQ→SP1 SQ→SP2 SQ→SP3 SQ→SPx	face	1 1 1	Data is valid Data is valid Data is valid		
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3		$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ tex inter	face	1 1 1	Data is valid Data is valid Data is valid		
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read		$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ tex inter	face	1 1 1	Data is valid Data is valid Data is valid		
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SP3_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1	ce Signal	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ $tex inter$ $Table$		1 1 1 1	Data is valid Data is valid Data is valid Increment the read pointers		
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interface The area difference between	<u>ce Signal</u> the two me	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> ethods is no	ot suffi	1 1 1 1	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface		Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interface The area difference between requirements of the VSISRs. T	<u>ce Signal</u> the two me <u>herefore, t</u>	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ tex inter <u>Table</u> ethods is no he POR for	ot suffi rthis	1 1 1 icient to v	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the	data to the	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. <u>I</u> VSISRs (via the Shader Sequered)	<u>ce Signal</u> the two me <u>'herefore, t</u> uencer) in	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ tex inter <u>Table</u> thods is no he POR for full, 32-bit	ot suffi <u>r this i</u> floatir	1 1 1 icient to v interface	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up	data to the to six 32-bit	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. I VSISRs (via the Shader Sequificating-point values to each VSISR)	the two me the two me <u>herefore, t</u> uencer) in SISR where	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> thods is not he POR for full, 32-bit e four or motion	ot suffi <u>r this i</u> floatir ore val	1 1 1 icient to v interface ng-point ues requi	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	data to the to six 32-bit	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. <u>I</u> VSISRs (via the Shader Sequered)	the two me the two me <u>herefore, t</u> uencer) in SISR where	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> thods is not he POR for full, 32-bit e four or motion	ot suffi <u>r this i</u> floatir ore val	1 1 1 icient to v interface ng-point ues requi	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	data to the to six 32-bit	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. I VSISRs (via the Shader Sequificating-point values to each VSISR)	the two me the two me <u>herefore, t</u> uencer) in SISR where	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> thods is not he POR for full, 32-bit e four or motion	ot suffi <u>r this i</u> floatir ore val	1 1 1 icient to v interface ng-point ues requi	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	data to the to six 32-bit	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. I VSISRs (via the Shader Sequificating-point values to each VSISR)	the two me the two me <u>herefore, t</u> uencer) in SISR where	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> thods is not he POR for full, 32-bit e four or motion	ot suffi <u>r this i</u> floatir ore val	1 1 1 icient to v interface ng-point ues requi	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	data to the to six 32-bit	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. I VSISRs (via the Shader Sequificating-point values to each VSISR)	the two me the two me <u>herefore, t</u> uencer) in SISR where	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> thods is not he POR for full, 32-bit e four or motion	ot suffi <u>r this i</u> floatir ore val	1 1 1 icient to v interface ng-point ues requi	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	data to the to six 32-bit	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. I VSISRs (via the Shader Sequificating-point values to each VSISR)	the two me the two me <u>herefore, t</u> uencer) in SISR where	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> thods is not he POR for full, 32-bit e four or motion	ot suffi <u>r this i</u> floatir ore val	1 1 1 icient to v interface ng-point ues requi	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	data to the to six 32-bit	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. I VSISRs (via the Shader Sequificating-point values to each VSISR)	the two me the two me <u>herefore, t</u> uencer) in SISR where	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> thods is not he POR for full, 32-bit e four or motion	ot suffi <u>r this i</u> floatir ore val	1 1 1 icient to v interface ng-point ues requi	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	data to the to six 32-bit	
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. I VSISRs (via the Shader Sequificating-point values to each VSISR)	the two me the two me <u>herefore, t</u> uencer) in SISR where	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> thods is not he POR for full, 32-bit e four or motion	ot suffi <u>r this i</u> floatir ore val	1 1 1 icient to v interface ng-point ues requi	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	data to the to six 32-bit	
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. I VSISRs (via the Shader Sequificating-point values to each VSISR)	the two me the two me <u>herefore, t</u> uencer) in SISR where	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> thods is not he POR for full, 32-bit e four or motion	ot suffi <u>r this i</u> floatir ore val	1 1 1 icient to v interface ng-point ues requi	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	data to the to six 32-bit	
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. I VSISRs (via the Shader Sequificating-point values to each VSISR)	the two me the two me <u>herefore, t</u> uencer) in SISR where	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> thods is not he POR for full, 32-bit e four or motion	ot suffi <u>r this i</u> floatir ore val	1 1 1 icient to v interface ng-point ues requi	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	data to the to six 32-bit	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. I VSISRs (via the Shader Sequificating-point values to each VSISR)	the two me the two me <u>herefore, t</u> uencer) in SISR where	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> thods is not he POR for full, 32-bit e four or motion	ot suffi <u>r this i</u> floatir ore val	1 1 1 icient to v interface ng-point ues requi	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	data to the to six 32-bit	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. I VSISRs (via the Shader Sequificating-point values to each VSISR)	the two me the two me <u>herefore, t</u> uencer) in SISR where	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> thods is not he POR for full, 32-bit e four or motion	ot suffi <u>r this i</u> floatir ore val	1 1 1 icient to v interface ng-point ues requi	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	data to the to six 32-bit	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. I VSISRs (via the Shader Sequificating-point values to each VSISR)	the two me the two me <u>herefore, t</u> uencer) in SISR where	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> thods is not he POR for full, 32-bit e four or motion	ot suffi <u>r this i</u> floatir ore val	1 1 1 icient to v interface ng-point ues requi	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	data to the to six 32-bit	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interfac The area difference between requirements of the VSISRs. I VSISRs (via the Shader Sequificating-point values to each VSISR)	the two me the two me <u>herefore, t</u> uencer) in SISR where	$SQ \rightarrow SP1$ $SQ \rightarrow SP2$ $SQ \rightarrow SP3$ $SQ \rightarrow SP3$ $SQ \rightarrow SPx$ <i>tex inter</i> <u>Table</u> thods is not he POR for full, 32-bit e four or motion	ot suffi <u>r this i</u> floatir ore val	1 1 1 icient to v interface ng-point ues requi	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	data to the to six 32-bit	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interface The area difference between trequirements of the VSISRs. T VSISRs (via the Shader Sequire floating-point values to each V bits wide. In the case where an	the two me therefore, t uencer) in SISR where event is se	SQ \rightarrow SP1 SQ \rightarrow SP2 SQ \rightarrow SP3 SQ \rightarrow SP3 tex inter Table ethods is no <u>he POR for</u> full, 32-bit e four or mo nt the 5 LSE	n this i floatir floatir re val 3s of V	icient to v interface ng-point ues requi (GT_SQ_)	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the <u>format.</u> The VGT can transmit up re two transmission clocks. The da vsisr_data contain the eventID.	<u>data to the</u> to six 32-bit ta bus is 96	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interface The area difference between trequirements of the VSISRs. T VSISRs (via the Shader Sequire floating-point values to each V bits wide. In the case where an	the two me therefore, t uencer) in SISR where event is se	SQ \rightarrow SP1 SQ \rightarrow SP2 SQ \rightarrow SP3 SQ \rightarrow SP3 tex inter Table ethods is no <u>he POR for</u> full, 32-bit e four or mo nt the 5 LSE	n this i floatir floatir re val 3s of V	icient to v interface ng-point ues requi (GT_SQ_)	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the format. The VGT can transmit up re two transmission clocks. The da	<u>data to the</u> to six 32-bit ta bus is 96	Formatted: Bullets and Numbering
SQ_SP1_vsr_valid SQ_SP2_vsr_valid SQ_SP3_vsr_valid SQ_SPx_vsr_read 23.2.522.2.5_VGT to 3 23.2.5.122.2.5.1 Interface The area difference between trequirements of the VSISRs. T VSISRs (via the Shader Sequire floating-point values to each V bits wide. In the case where an	the two me therefore, t uencer) in SISR where event is se	SQ \rightarrow SP1 SQ \rightarrow SP2 SQ \rightarrow SP3 SQ \rightarrow SP3 tex inter Table ethods is no <u>he POR for</u> full, 32-bit e four or mo nt the 5 LSE	n this i floatir floatir re val 3s of V	icient to v interface ng-point ues requi (GT_SQ_)	Data is valid Data is valid Data is valid Increment the read pointers warrant complicating the interface is that the VGT will transmit the <u>format.</u> The VGT can transmit up re two transmission clocks. The da vsisr_data contain the eventID.	<u>data to the</u> to six 32-bit ta bus is 96	Formatted: Bullets and Numbering

ATI Ex. 2109 IPR2023-00922 Page 143 of 326

	ORIGINATE DATE		EDIT DATE	R400 Sequencer Specification	PAGE			
CALL	24 September	, 2001	4 September, 20157		42 of 54			
Name		Bits	Description					
VGT_SQ_vsisr_data 96		96	Pointers of indexes or HOS surface information					
VGT_SQ_event 1		1	VGT is sending an event					
VGT_SQ_vsisr_continued 1		0: Normal 96 bits per vert 1: double 192 bits per vert						
VGT_SQ_end_of_vtx_vect 1			Indicates the last VSISR data set for the current process vector (for double vector data, "end of vector" is set on the first vector)					
VGT_SQ_indx_valid 1		Vsisr data is valid						
VGT_SQ_state 3		Render State (6*3+3 for constants). This signal is guaranteed to be correct when "VGT_SQ_vgt_end_of_vector" is high.						
			Data on the VGT_SQ is valid receive (see write-up for standard R400 SEND/RTR interface handshaking)					
SQ_VGT_rtr		1	Ready to receive (see handshaking)	write-up for standard R400 SEND/	RTR interface			

23.2.5.222.2.5.2 Interface Diagrams

--- Formatted: Bullets and Numbering

Exhibit 2037.doc.R400_Sequences.doc 74373 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257854

ATI Ex. 2109 IPR2023-00922 Page 144 of 326

PROTECTIVE ORDER MATERIAL

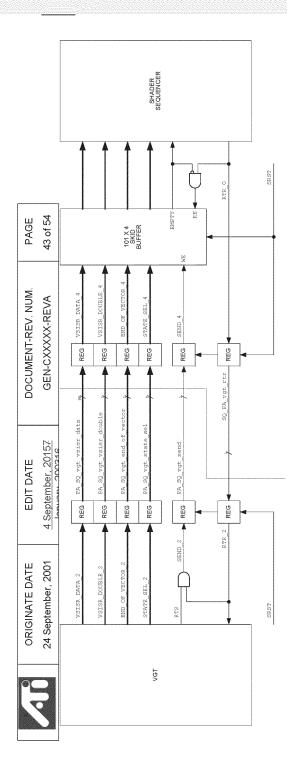
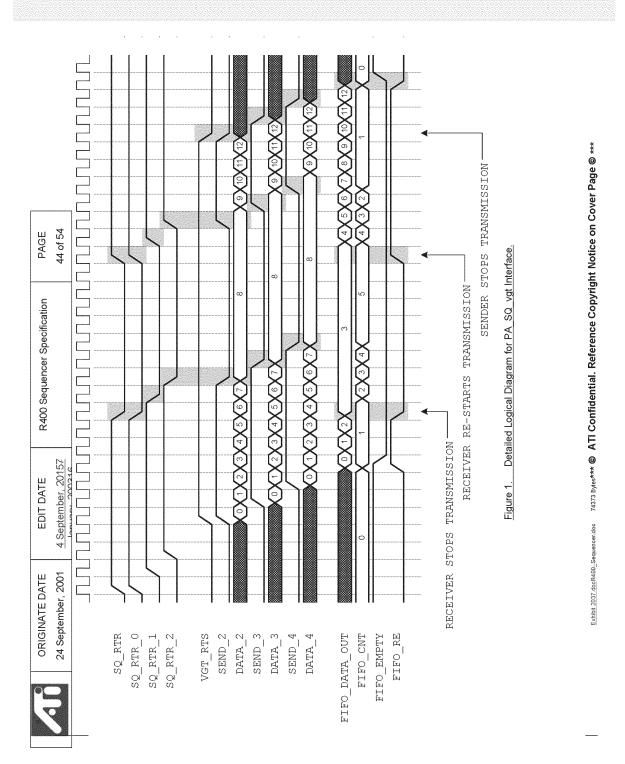


Exhibit 2027 doeR400_Sequences 74373 Bytes +++ @ ATI Confidential. Reference Copyright Notice on Cover Page @ +++

AMD1044_0257855

ATI Ex. 2109 IPR2023-00922 Page 145 of 326 PROTECTIVE ORDER MATERIAL



ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 20157	GEN-CXXXXX-REVA	45 of 54

- Formatted: Bullets and Numbering

23.2.622.2.6 SQ to SX: Control bus

Name	Direction	Bits	Description
SQ_SXx_exp_type	SQ→SXx	2	00: Pixel without z (1 to 4 buffers) 01: Pixel with z (1 to 4 buffers) 10: Position (1 or 2 results) 11: Pass thru (4,8 or 12 results aligned)
SQ_SXx_exp_number	SQ→SXx	2	Number of locations needed in the export buffer (encoding depends on the type see bellow).
SQ_SXx_exp_alu_id	SQ→SXx	1	ALU ID
SQ_SXx_exp_valid	SQ→SXx	1	Valid bit
SQ_SXx_exp_state	SQ→SXx	3	State Context
SQ_SXx_free_done	SQ→SXx	1	Pulse that indicates that the previous export is finished from the point of view of the SP. This does not necessarily mean that the data has been transferred to RB or PA, or that the space in export buffer for that particular vector thread has been freed up.
SQ_SXx_free_alu_id	SQ→SXx	1	ALU ID

Depending on the type the number of export location changes:

- Type 00 : Pixels without Z
 - 00 = 1 buffer
 - 01 = 2 buffers
 - 10 = 3 buffers
 - 11 = 4 buffer

.

.

- Type 01: Pixels with Z
 - 00 = 2 Buffers (color + Z)
 - 01 = 3 buffers (2 color + Z)
 - \circ 10 = 4 buffers (3 color + Z)
 - \circ 11 = 5 buffers (4 color + Z)
- Type 10 : Position export
 - 00 = 1 position
 - \circ 01 = 2 positions
 - 1X = Undefined
 - Type 11: Pass Thru
 - 00 = 4 buffers
 - \circ 01 = 8 buffers
 - 10 = 12 buffers
 - 11 = Undefined

Below the thick black line is the end of transfer packet that tells the SX that a given export is finished. The report packet will always arrive either before or at the same time than the next export to the same ALU id.

23.2.7 22.2.7 SX to	SQ : Output fil	e control		¢-	Formatted: Bullets and Numberin
Name	Direction	Bits	Description	٦	
SXx_SQ_exp_count_rdy	SXx→SQ	1	Raised by SX0 to indicate that the following two fields reflect the result of the most recent export		
SXx_SQ_exp_pos_avail	SXx→SQ	2	Specifies whether there is room for another position. 00 : 0 buffers ready 01 : 1 buffer ready 10 : 2 or more buffers ready		
SXx_SQ_exp_buf_avail	SXx→SQ	7	Specifies the space available in the output buffers. 0: buffers are full 1: 2K-bits available (32-bits for each of the 64		

Exhibit 2037. docR400_Sequences.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257857

ATI Ex. 2109 IPR2023-00922 Page 147 of 326

ORIGINATE D	DATE	EDIT D	ATE	R400 Sequencer Specification	PAGE	7
24 September,	2001	4 September			46 of 54	
				els in a clause)		T
			64	128K-bits available (16 128-bit entries pixels) 127: RESERVED	s for each of	

23.2.822.2.8 SQ to TP: Control bus

Once every clock, the fetch unit sends to the sequencer on which RS line it is now working and if the data in the GPRs is ready or not. This way the sequencer can update the fetch valid bits flags for the reservation station. The sequencer also provides the instruction and constants for the fetch to execute and the address in the register file where to write the fetch return data.

Name	Direction	Bits	Description
TPx_SQ_data_rdy	TPx→ SQ	1	Data ready
TPx_SQ_rs_line_num	TPx→ SQ	6	Line number in the Reservation station
TPx_SQ_type	$TPx \rightarrow SQ$	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_TPx_send	SQ→TPx	1	Sending valid data
SQ_TPx_const	SQ→TPx	48	Fetch state sent over 4 clocks (192 bits total)
SQ_TPx_instr	SQ→TPx	24	Fetch instruction sent over 4 clocks
SQ_TPx_end_of_group	SQ→TPx	1	Last instruction of the group
SQ_TPx_Type	SQ→TPx	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_TPx_gpr_phase	SQ→TPx	2	Write phase signal
SQ_TP0_lod_correct	SQ→TP0	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP0_pix_mask	SQ→TP0	4	Pixel mask 1 bit per pixel
SQ_TP1_lod_correct	SQ→TP1	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP1_pix_mask	SQ→TP1	4	Pixel mask 1 bit per pixel
SQ_TP2_lod_correct	SQ→TP2	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP2_pix_mask	SQ→TP2	4	Pixel mask 1 bit per pixel
SQ_TP3_lod_correct	SQ→TP3	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP3_pix_mask	SQ→TP3	4	Pixel mask 1 bit per pixel
SQ_TPx_rs_line_num	SQ→TPx	6	Line number in the Reservation station
SQ_TPx_write_gpr_index	SQ->TPx	7	Index into Register file for write of returned Fetch Data
SQ_TPx_ctx_id	SQ→TPx	3	The state context ID (needed for multisample resolves)

23.2.922.2.9 TP to SQ: Texture stall

The TP sends this signal to the SQ and the SPs when its input buffer is full.

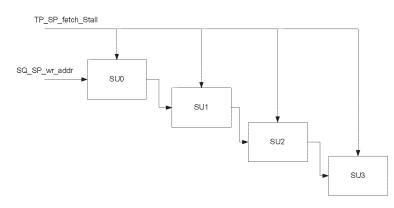


Exhibit 2037.doc.R400_Sequencer.doc 74373 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

Formatted: Bullets and Numbering

AMD1044_0257858

ATI Ex. 2109 IPR2023-00922 Page 148 of 326

A II	ORIGINATE DATE 24 September, 2001	4 Septer	DIT DATE ember, 20157		DOCUMENT-REV. NUM. GEN-CXXXXX-REVA	PAGE 47 of 54
Name TP_SQ_fet	Direc ch_stall TP→		Bits 1	Descrij Do not	ption send more texture request if asserted	

23.2.10 SQ to SP: Texture stall

23.2.1122.2.10 SQ to SP: GPR and auto counter

Name	Direction	Bits	Description
SQ SPx gpr_wr_addr	SQ→SPx	7	Write address
SQ_SPx_gpr_rd_addr	SQ→SPx	7	Read address
SQ SPx gpr rd en	SQ→SPx	1	Read Enable
SQ_SP0_gpr_pspv_wr_en	SQ→SPx	4	Write Enable for the GPRs of SP0 for PS and PV
SQ_SP1_gpr_pspv_wr_en	SQ→SPx	4	Write Enable for the GPRs of SP1 for PS and PV
SQ_SP2_gpr_pspv_wr_en	SQ→SPx	4	Write Enable for the GPRs of SP2 for PS and PV
SQ SP3 gpr pspv wr en	<u>SQ→SPx</u>	4	Write Enable for the GPRs of SP3 for PS and PV
<u>SQ SP0 gpr int wr en</u>	<u>SQ→SPx</u>	1	Write Enable for the GPRs of SP0 for Inputs (interp/vtx)
<u>SQ SP1 gpr int wr en</u>	<u>SQ→SPx</u>	1	Write Enable for the GPRs of SP1 for Inputs (interp/vtx)
<u>SQ SP2 gpr int wr en</u>	<u>SQ→SPx</u>	1	Write Enable for the GPRs of SP2 for Inputs (interp/vtx)
SQ_SP3_gpr_int_wr_enSQ_SP3_gpr_wr_en	<u>SQ→SPx</u> SQ→SPx	14	Write Enable for the GPRs of SP3 for Inputs (interp/vtx) GPRs of SP3
SQ_SPx_gpr_phase	SQ→SPx	2	The phase mux (arbitrates between inputs, ALU SRC reads and writes)
SQ_SPx_channel_mask	SQ→SPx	4	The channel mask
SQ_SPx_gpr_input_sel	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_auto_count	SQ→SPx	21	Auto count generated by the SQ, common for all shader pipes

Exhibit 2037.docR400_Sequences.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

4

AMD1044_0257859

ATI Ex. 2109 IPR2023-00922 Page 149 of 326

29	ORIGINATE D	ATE	EDIT	DATE	R400 Sequencer Specification	PAGE	
<u>4</u>	24 September,	2001	4 Septemb			48 of 54	
2 1 2 2 2	2.11 SQ to	SPv.	Instruction			*-	- Formatted: Bullets and Numbering
	<u></u>						
Name SO SPx in	str start		ection →SPx	Bits 1	Description Instruction start		
		→SPx	24	Transferred over 4 cycles 0: SRC A Negate Argument Modifier 0:0 SRC A Abs Argument Modifier 1:1 SRC A Swizzle 9:2 Vector Dst 15:10 Per channel Select 23:16 00: GPR 01: PV 10: PS 11: Constant (if 11 has to channels)	5		
					1: SRC B Negate Argument Modifier 0:0 SRC B Abs Argument Modifier 1:1 SRC B Swizzle 9:2 Scalar Dst 15:10 Per channel Select 23:16 00: GPR 01: PV 10: PS 11: Constant (if 11 has t channels)	5	
					- 2: SRC C Negate Argument Modifier 0:0 SRC C Abs Argument Modifier 1:1 SRC C Swizzle 9:2 Unused 15:10 Per channel Select 23:16 00: GPR 01: PV 10: PS 11: Constant (if 11 has to channels)	5	
					- 3: Vector Opcode Scalar Opcode Vector Clamp 11:11 Scalar Clamp 12:12 Vector Write Mask 16:13 Scalar Write Mask 20:17 Unused 23:21		
	ed_override		→SP0	4	0: Use per channel RGBA field (ena channel logic, if not set only pay attent seting). 1: Use GPR	on to the 11	
	ed_override		→SP1	4	0: Use per channel RGBA field (ena channel logic, if not set only pay attent seting). 1: Use GPR	on to the 11	
SQ_SP2_pr	ed_override		→SP2	4	0: Use per channel RGBA field (ena channel logic, if not set only pay attent seting). 1: Use GPR	on to the 11	
SQ_SP3_pr	ed_override	SQ-	→SP3	4	0: Use per channel RGBA field (ena channel logic, if not set only pay attent		

AMD1044_0257860

ATI Ex. 2109 IPR2023-00922 Page 150 of 326

	ORIGINATE DATE		ORIGINATE DATE EDIT DAT			DOCUMENT-REV. NUM.	PAGE	
	24 September, 20	001	01 <u>4 September, 20157</u>		~~~~	GEN-CXXXXX-REVA	49 of 54	
SQ SPx e	exp id	SQ→S		1	setin	GPR		
SQ_SPx_e				SPx 1		0: Not Exporting 1: Exporting		
SQ_SPx_si SQ_SPx_V		SQ→S SQ→S		2	Use for th 0 : N 1: W 2: W	signal the incoming constant instead of the i e next group of 16. ormal mode aterfall on SRCA aterfall on SRCB aterfall on SRCC	registered one	

23.2.1322.2.12_SQ to SX: write mask interface (must be aligned with the SP data) *

Name	Direction	Bits	Description
SQ_SX0_write_mask	SQ→SP0	8	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. This is for the data coming of SP0 and SP2.
SQ_SX1_ write_mask	SQ→SP1	8	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. This is for the data coming of SP1 and SP3.

23.2.1422.2.13 SP to SQ: Constant address load/ Predicate Set/Kill set

Name	Direction	Bits	Description
SP0_SQ_const_addr	SP0→SQ	36	Constant address load / predicate vector load (4 bits only)/
			Kill vector load (4 bits only) to the sequencer
SP0_SQ_valid	SP0→SQ	1	Data valid
SP1_SQ_const_addr	SP1→SQ	36	Constant address load / predicate vector load (4 bits only)/
			Kill vector load (4 bits only) to the sequencer
SP1_SQ_valid	SP1→SQ	1	Data valid
SP2_SQ_const_addr	SP2→SQ	36	Constant address load / predicate vector load (4 bits only)/
			Kill vector load (4 bits only) to the sequencer
SP2_SQ_valid	SP2→SQ	1	Data valid
SP3_SQ_const_addr	SP3→SQ	36	Constant address load / predicate vector load (4 bits only)/
			Kill vector load (4 bits only) to the sequencer
SP3_SQ_valid	SP3→SQ	1	Data valid
SP0_SQ_data_type	SP→SQ	2	Data Type
			0: Constant Load
			1: Predicate Set
			2: Kill vector load

Because of the sharing of the bus none of the MOVA, PREDSET or KILL instructions may be coissued.

23.2.1522.2.14 SQ to SPx: constant broadcast

	Name	Direction	Bits	Description
	SQ_SPx_const	SQ→SPx	128	Constant broadcast
2	<u>3.2.1622.2.15</u> SQ to			

Name	Direction	Bits	Description	1000
SQ_RBB_rs	SQ→CP	1	Read Strobe	1000
SQ_RBB_rd	SQ→CP	32	Read Data	
SQ_RBBM_nrtrtr	SQ→CP	1	Optional	1000
SQ_RBBM_rtr	SQ→CP	1	Real-Time (Optional)	100

Exhibit 2037.docR400_Sequences.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

AMD1044_0257861

ATI Ex. 2109 IPR2023-00922 Page 151 of 326

1	ORIGINATE D 24 September,		EDIT D <u>4 Septembe</u>	er, 201		PAGE 50 of 54	
2 1 7 2 2	2.16 CP to	SO' F	PRRM bus			4-	- Formatted: Bullets and Numbering
. 1 / <u></u>	. <u></u> 0, 10	Directio		Bits	Description		
bbm we		CP→S(1	Write Enable		
rbbm_we		CP→S(-	15	Address Upper Extent is TBD (16:2)		
rbbm wd		CP→S(-	32	Data		
bbm be	*****	CP→SC		4	Byte Enables		
bbm re		CP→SC	2	1	Read Enable		
bb_rs0		CP→SC	2	1	Read Return Strobe 0		
bb_rs1		CP→S0	2	1	Read Return Strobe 1		
bb_rd0		CP→S(כ	32	Read Data 0		
bb_rd1		CP→SC	2	32	Read Data 0		
RBBM_SC	_soft_reset	CP→SC	2	1	Soft Reset		
<u>2.1822</u>	. <u>2.17_</u> SQ to	CP: S	tate repoi	t		*-	Formatted: Bullets and Numbering
Name	~~~~~	Directio	on	Bits	Description	18	
SQ_CP_vs	_event	SQ→C	Р	1	Vertex Shader Event		
SQ_CP_vs	_eventid	SQ→C	Р	5	Vertex Shader Event ID		
SQ_CP_ps		SQ→C		1	Pixel Shader Event		
00 00	and a second field	00 0	0	~		25	

Pixel Shader Event ID

23.322.3 Example of control flow program execution

5

We now provide some examples of execution to better illustrate the new design.

SQ→CP

Given the program:

SQ_CP_ps_eventid

Alu 0 Alu 1 Tex 0 Tex 1 Alu 3 Serial Alu 4 Tex 2 Alu 5 Alu 6 Serial Tex 3 Alu 7 Alloc Position 1 buffer Alu 8 Export Tex 4 Alloc Parameter 3 buffers Alu 9 Export 0 Tex 5 Alu 10 Serial Export 2 Alu 11 Export 1 End

Would be converted into the following CF instructions:

Execute 0 Alu 0 Alu 0 Tex 0 Tex 1 Alu 0 Alu 0 Tex 0 Alu 1 Alu 0 Tex Execute 0 Alu Alloc Position 1 Execute 0 Alu 0 Tex Alloc Param 3 Execute_end 0 Alu 0 Tex 1 Alu 0 Alu

And the execution of this program would look like this:

Exhibit 2037.docR400_Sequencer.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

Formatted: Bullets and Numbering

AMD1044_0257862

ATI Ex. 2109 IPR2023-00922 Page 152 of 326

	ORIGIN	ATE DAT	E	EDIT	DATE		DOCUN	ENT-RE	V. NUM.	1	PAGE
<u>~~~</u>	24 Septe	mber, 20	01		per, 20157		GEN-C	XXXXX	-REVA	5'	1 of 54
Put thread in V	Vertex RS:			lanuani	200216	1					
Execution Loop Itera Call returr Predicate Export ID GPR Base Export Ba Context P	ow Instructio Count Marka ators (4x9 bits pointers (4x Bits(4x64 bit (1 bit), (EXIE e Ptr (8 bits), use Ptr (7 bits thr (3 bits).(CF ection bits (16	er (3 or 4 s), (Ll) :12 bits), s), (PB))) (GPR)), (EB) PTR)	bits), (ÉC (CRP)								
State Bits											
	CM LI		CRP	PB	EXID	G	SPR	EB	CPTR	LC	OD
0 0			0	0	0	0		0	0	0	
00 — N 01 — F	Wait (2 bits) No allocation Position expo	needed rt allocati	íon needeo								
Allocation 00 – N 01 – F 10 – F 11 – p Allocation Position A	No allocation Position expo Parameter or pass thru (out Size (4 bits) Nlocated (PO ad of a new ca	needed rt allocati pixel exp t of order (SIZE) S_ALLO	ion needed oort neede export) C)								
Allocation 00 – N 01 – F 10 – F 11 – p Allocation Position A First threa	No allocation Position expo Parameter or pass thru (out Size (4 bits) Nlocated (PO ad of a new ca	needed rt allocati pixel exp t of order (SIZE) S_ALLO	ion needed oort neede export) C)								
Allocation 00 - N 01 - F 10 - F 11 - p Allocation Position A First threa Last (1 bit Status Bits VALID	No allocation Position expo Parameter or pass thru (out) Size (4 bits) Ullocated (PO dd of a new cr y), (LAST) TYPE	needed rt allocati pixel exp t of order (SIZE) S_ALLOC ontext (Fi	ion needed ort needed export) C) IRST) NG SE		d export)	SIZE	POS_A	LLOC	FIRST	LAST	Г
Allocation 00 – N 01 – F 10 – F 11 – p Allocation Position A First threa Last (1 bit Status Bits VALID 1 Then the f	No allocation Position expo Parameter or poass thru (out Size (4 bits) Vilocated (PO ad of a new co ad of a new co (LAST) TYPE ALU thread is pick	needed rt allocati pixel exp t of order (SIZE) S_ALLOC ontext (FI 0 PENDIN 0	ion needed oort needed export) C) IRST) NG SE 0 r the execu	RIAL	ALLOC 0	0 I flow ir	0		FIRST	LAST 0	T
Allocation 00 – N 01 – F 10 – F 11 – p Allocation Position A First threa Last (1 bit Status Bits VALID 1 Then the f Execute 0 J It execute state retur	No allocation Position expo Parameter or bass thru (out Size (4 bits) Nilocated (PO ad of a new ca bit of a new ca character (PO ad	reeded rt allocati pixel exp t of order (SIZE) S_ALLOC ontext (FI 0 reed up for 0 Tex (ro ALU ir	ion needed oort needed export) C) IRST) NG SE 0 r the execu 0 Tex 1	RIAL Alu 0 A	ALLOC 0 e first contro	0 I flow ir 0 Alu e RS f	0 nstruction 1 Alu or a resc	: O Tex	1	0 e. Here	e is the
Allocation 00 – N 01 – F 10 – F 11 – F Allocation Position A First threa Last (1 bit Status Bits VALID 1 Then the f Execute 0 It execute state retur State Bits CFP E	No allocation Position expo Parameter or bass thru (out Size (4 bits) Nilocated (PO ad of a new ca bit of a new ca character (PO ad	reeded rt allocati pixel exp t of order (SIZE) S_ALLOC ontext (FI 0 reed up for 0 Tex (ro ALU ir	ion needed oort needed export) C) IRST) NG SE 0 r the exect 0 Tex 1 nstructions	RIAL tition of the Alu 0 A and goes	ALLOC 0 e first contro .lu 0 Tex s back to th	0 I flow ir 0 Alu e RS f	0 nstruction 1 Alu or a resc	: O Tex urce rec	1 uest change	e. Here	e is the
Allocation 00 - N 01 - F 10 - F 11 - p Allocation Position A First threa Last (1 bit Status Bits VALID 1 Then the f Execute 0 2 It execute state Bits CFP EF 0 2 Status Bits	No allocation Position expo Parameter or bass thru (out Size (4 bits) Ullocated (PO ad of a new cr ch (LAST) TYPE ALU thread is pick Alu 0 Alu es the first tw rned to the R CM LI 0 TYPE	reeded rt allocati pixel exp t of order (SIZE) S_ALLOC ontext (FI 0 reed up for 0 Tex (ro ALU ir	ion needed oort needed export) C) IRST) NG SE 0 r the exect 0 Tex 1 nstructions CRP 0	RIAL tition of the Alu 0 A and goes	ALLOC 0 e first contro lu 0 Tex s back to th EXID 0	0 I flow ir 0 Alu e RS f	0 nstruction 1 Alu or a resc	: 0 Tex urce rec EB 0	1 uest change	e. Here	e is the
Allocation 00 – N 01 – F 10 – F 11 – p Allocation Position A First threa Last (1 bit Status Bits VALID 1 Then the f Execute 0 2 It execute state Bits CFP E 0 2 Status Bits VALID 1 1	No allocation Position expo Parameter or bass thru (out Size (4 bits) Nilocated (PO dd of a new co dd of a new	PENDIN 0 PENDIN 0 PENDIN 0 PENDIN 0 PENDIN 0 PENDIN 0	ion needed oort needed export) C) IRST) NG SE 0 r the execu 0 Tex 1 nstructions CRP 0 NG SE 0	RIAL and goes RIAL RIAL RIAL	ALLOC 0 e first contro lu 0 Tex s back to th EXID 0 ALLOC 0	0 I flow ir 0 Alu e RS f G 0 SIZE 0	0 struction 1 Alu or a resc PR POS_A 0	: 0 Tex urce rec EB 0	1 uest chang CPTR 0 FIRST 1	0 e. Here 0 LAST 0	e is the OD
Allocation 00 - N 01 - F 10 - F 11 - p Allocation Position A First threa Last (1 bit Status Bits VALID 1 Then the f Execute 0 It execute state retur State Bits VALID 1 Then whe back in this st State Bits	No allocation Position expo Parameter or bass thru (out Size (4 bits) Nilocated (PO dd of a new co dd of a new	PENDIN 0 PENDIN 0 PENDIN 0 PENDIN 0 PENDIN 0 PENDIN 0	ion needed oort needed export) C) IRST) NG SE 0 r the execu 0 Tex 1 nstructions CRP 0 NG SE 0	RIAL and goes RIAL RIAL RIAL	ALLOC 0 e first contro lu 0 Tex s back to th EXID 0 ALLOC 0	0 I flow ir 0 Alu e RS f 0 SIZE 0 read to	0 struction 1 Alu or a resc PR POS_A 0	: 0 Tex urce rec EB 0	1 uest chang CPTR 0 FIRST 1	0 e. Here LC 0 thread	e is the OD

AMD1044_0257863

ATI Ex. 2109 IPR2023-00922 Page 153 of 326

A î		ATE DATE mber, 2001	EDIT E	er, 20157	R400) Sequencer Spec	ification	PAGE 52 of 54	
Status Bits									
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST	
1	ALU	1	1	0	0	0	1	0	

Because of the serial bit the arbiter must wait for the texture to return and clear the PENDING bit before it can pick the thread up. Lets say that the texture reads are complete, then the arbiter picks up the thread and returns it in this state:

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
0	6	0	0	0	0	0	0	0	0

Status Bits	S							
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	0	0	0	0	0	1	0

Again the TP frees up, the arbiter picks up the thread and executes. It returns in this state:

State Bi	ts								
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
0	7	0	0	0	0	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	0	0	0	1	0

Now, even if the texture has not returned we can still pick up the thread for ALU execution because the serial bit is not set. The thread will however come back to the RS for the second ALU instruction because it has the serial bit set.

NAMES AND POST OFFICE ADDRESS OF TAXABLE POST OF		IN CONTRACTOR OF	Company April Company Anno Anno Anno Anno Anno Anno Anno An	CONVERSION AND DESCRIPTION OF THE OWNER			AND DESCRIPTION OF THE OWNER OF T		
CFP	ECM	LI	CRP	PB		GPR	EB	CPTR	LOD
0	8	0	0	0	0	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	1	0	0	0	1	0

As soon as the TP clears the pending bit the thread is picked up and returns:

State Bit	s									
		LI	CRP	PB	EXID	GPR	EB	CPTR	LOD	HOMOSON.
0	9	0	0	0	0	0	0	0	0	

Status Bit	s							
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	0	0	0	0	0	1	0

Exhibit 2037.doc R400_Sequencer.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257864

ATI Ex. 2109 IPR2023-00922 Page 154 of 326

		RIGINATE		EDIT DATE DOCUMENT-REV. NUM. 4 September, 20157 GEN-CXXXXX-REVA Japunar, 200316 Instrument of the sector of					PAGE 53 of 54
	up by the te O Alu	TP and re	turns:						
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
	0	0	0	0	0	0	0	0	0

VALID TYPE PEN	IDING SERIAL A		S_ALLOC FIRST	LAST
1 ALU 1	0 0	0 0	1	0

Picked up by the ALU and returns (lets say the TP has not returned yet): Alloc Position 1 $\,$

r-

State B	its	***	Manual Calculation Manual Science 20							
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD	
2	0	0	0	0	0	0	0	0	0	

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	01	1	0	1	0

If the SX has the place for the export, the SQ is going to allocate and pick up the thread for execution. It returns to the RS in this state:

Execute 0 Alu 0 Tex

State Bit	S									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD	
3	1	0	0	0	0	0	0	0	0	

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	1	0	0	0	1	1	0

Now, since the TP has not returned yet, we must wait for it to return because we cannot issue multiple texture requests. The TP returns, clears the PENDING bit and we proceed:

Alloc Param 3

State Bits											
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD		
4	0	0	0	0	1	0	0	0	0		

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	10	3	1	1	0

Once again the SQ makes sure the SX has enough room in the Parameter cache before it can pick up this thread.

Exhibit 2037.docR400_Sequencer.doc 74373 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257865

ATI Ex. 2109 IPR2023-00922 Page 155 of 326

		IGINATE I September		EDIT DATE R400 Sequencer Specification						PAGE 4 of 54
Exe State Bi	cute_end (ts) Alu O	Tex 1 A	lu O Alu						
CFP	ECM	LI	CRF	PB	EXI) GPF	EB	CPTR	L	LOD
5	1	0	0	0	1	0	100	0	0	5
Status E	Bits									

NNM STORE S		A CONTRACTOR OF	NOT THE REAL PROPERTY OF THE PARTY OF THE PA	AND DESCRIPTION OF THE OWNER OF T	CONTRACTOR OF STREET, STRE	And the second	CONTRACTOR OF CONT	
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	1	0	0	0	1	1	0

This executes on the TP and then returns:

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
5	2	0	0	0	1	0	100	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	1	0	0	1	1	1

Waits for the TP to return because of the textures reads are pending (and SERIAL in this case). Then executes and does not return to the RS because the LAST bit is set. This is the end of this thread and before dropping it on the floor, the SQ notifies the SX of export completion.

24.23. Open issues

Need to do some testing on the size of the register file as well as on the register file allocation method (dynamic VS static).

Saving power?

Exhibit 2037.docR400_Sequences.doc 74373 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

- Formatted: Bullets and Numbering

AMD1044_0257866

ATI Ex. 2109 IPR2023-00922 Page 156 of 326

	ORIGINATE DATE	EDIT DATE	-	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 2	2015	GEN-CXXXXX-REVA	1 of 56
Author:	Laurent Lefebvre				•
ssue To:			opy No:		
33ue 10.			юру но.		
	R400 S	equencer	Spec	cification	
		SQ)		
		Version	2 40		
		version	2.10		
				r block (SEQ). It provides an ov	
	uired capabilities and expe cks, and provides internal sta		ock. It also	describes the block interfaces,	internal sub-
	ALLY UPDATED FIELDS:				
Document L	ocation: C:\pe			cks\sq\R400_Sequencer.doc	
Document L	ocation: C:\pe	Sequencer Specifi	cation	cks\sq\R400_Sequencer.doc	
Document L	ocation: C:\pe		cation	cks\sq\R400_Sequencer.doc Signature/Date	
Document L	ocation: C:\pe net Search Title: R400	Sequencer Specifi	cation		
Document L	ocation: C:\pe net Search Title: R400	Sequencer Specifi	cation		
Document L	ocation: C:\pe net Search Title: R400	Sequencer Specifi	cation		
Document L	ocation: C:\pe net Search Title: R400	Sequencer Specifi	cation		
Document Lo Current Intra	ocation: C:\pe net Search Title: R400	Sequencer Specifi	cation		
Document Lo Current Intra	ocation: C:\pe net Search Title: R400	Sequencer Specifi	cation		
Document Lo Current Intra	ocation: C:\pe inet Search Title: R400 Name/Dept	Sequencer Specifi	ALS	Signature/Date	
Document Lo Current Intra	CUMENT CONTAIN	Sequencer Specifi APPROV	TIAL IN	Signature/Date	
Document Lo Current Intra	CUMENT CONTAIN	S CONFIDEN	TIAL IN	Signature/Date	
Document Lo Current Intra	CUMENT CONTAIN	S CONFIDEN	TIAL IN	Signature/Date Signature/Date FORMATION THAT CO EST OF ATI TECHNOL	
Document Lo Current Intra	CUMENT CONTAIN	S CONFIDEN	TIAL IN	Signature/Date Signature/Date FORMATION THAT CO EST OF ATI TECHNOL	
Document Lo Current Intra	CUMENT CONTAIN NTIALLY DETRIME INC. THROUGH L	S CONFIDEN NTAL TO THE	TIAL IN E INTER ED USE	Signature/Date FORMATION THAT CO EST OF ATI TECHNOL OR DISCLOSURE.	OGIES
Document Lo Current Intra Remarks: THIS DOO SUBSTA	CUMENT CONTAIN Name/Dept	S CONFIDEN NTAL TO THE INAUTHORIZ	TIAL IN E INTER ED USE	Signature/Date FORMATION THAT CO EST OF ATI TECHNOL OR DISCLOSURE.	OGIES
Document Lo Current Intra Remarks: THIS DOO SUBSTA	CUMENT CONTAIN Name/Dept CUMENT CONTAIN NTIALLY DETRIME INC. THROUGH L	Sequencer Specifi APPROV S CONFIDEN NTAL TO THE INAUTHORIZ	TIAL IN E INTER ED USE	Signature/Date FORMATION THAT CO EST OF ATI TECHNOL OR DISCLOSURE.	OGIES
Document Lo Current Intra Remarks: THIS DOO SUBSTA	CUMENT CONTAIN Name/Dept CUMENT CONTAIN NTIALLY DETRIME INC. THROUGH L 1, ATI Technologies Inc. All 2001. The use of this cop ork. The copyright notice oprietary information and tra	Sequencer Specifi APPROV SCONFIDEN SCONFIDEN NTAL TO THE INAUTHORIZ rights reserved. T yright notice is inte is not an admissi ade secrets of ATI.	TIAL IN TIAL IN E INTER ED USE the materia anded to pr ion that pu No part o	Signature/Date Signature/Date FORMATION THAT CO EST OF ATI TECHNOL OR DISCLOSURE. I in this document constitutes an ovide notice that ATI owns a co ublication has occurred. This w f this document may be used, re	OGIES
Document Lo Current Intra Remarks: THIS DOC SUBSTA	CUMENT CONTAIN Name/Dept CUMENT CONTAIN NTIALLY DETRIME INC. THROUGH L 1, ATI Technologies Inc. All 2001. The use of this cop ork. The copyright notice	Sequencer Specifi APPROV SCONFIDEN SCONFIDEN NTAL TO THE INAUTHORIZ rights reserved. T yright notice is inte is not an admissi ade secrets of ATI.	TIAL IN TIAL IN E INTER ED USE the materia anded to pr ion that pu No part o	Signature/Date Signature/Date FORMATION THAT CO EST OF ATI TECHNOL OR DISCLOSURE. I in this document constitutes an ovide notice that ATI owns a co ublication has occurred. This w f this document may be used, re	OGIES
Document Lo Current Intra Remarks: THIS DOO SUBSTA	CUMENT CONTAIN Name/Dept CUMENT CONTAIN NTIALLY DETRIME INC. THROUGH L 1, ATI Technologies Inc. All 2001. The use of this cop ork. The copyright notice oprietary information and tra	Sequencer Specifi APPROV SCONFIDEN SCONFIDEN NTAL TO THE INAUTHORIZ rights reserved. T yright notice is inte is not an admissi ade secrets of ATI.	TIAL IN TIAL IN E INTER ED USE the materia anded to pr ion that pu No part o	Signature/Date Signature/Date FORMATION THAT CO EST OF ATI TECHNOL OR DISCLOSURE. I in this document constitutes an ovide notice that ATI owns a co ublication has occurred. This w f this document may be used, re	OGIES
Document Lo Current Intra Remarks: THIS DOC SUBSTA	CUMENT CONTAIN Name/Dept	S CONFIDEN S CONFIDEN NTAL TO THE JNAUTHORIZ rights reserved. T yright notice is inte is not an admissi ade secrets of ATI. hout the prior writte	TIAL IN TIAL IN TIAL IN TIAL IN E INTER ED USE the materia anded to pr ion that pu No part o on permissio	Signature/Date Signature/Date FORMATION THAT CO EST OF ATI TECHNOL OR DISCLOSURE.	OGIES unpublished oyright in this vork contains eproduced, or
Document Lo Current Intra Remarks: THIS DOC SUBSTA	CUMENT CONTAIN Name/Dept	S CONFIDEN S CONFIDEN NTAL TO THE JNAUTHORIZ rights reserved. T yright notice is inte is not an admissi ade secrets of ATI. hout the prior writte	TIAL IN TIAL IN TIAL IN TIAL IN E INTER ED USE the materia anded to pr ion that pu No part o on permissio	Signature/Date Signature/Date FORMATION THAT CO EST OF ATI TECHNOL OR DISCLOSURE. I in this document constitutes an ovide notice that ATI owns a co ublication has occurred. This w f this document may be used, re	OGIES unpublished oyright in this vork contains eproduced, or

AMD1044_0257867

ATI Ex. 2109 IPR2023-00922 Page 157 of 326



Table Of Contents

1000	OVERVIEW	
1.1	Top Level Block Diagram	
1.2	Data Flow graph (SP)	
1.3	Control Graph	
2.	INTERPOLATED DATA BUS	
3.		
4. 5.	SEQUENCER INSTRUCTIONS CONSTANT STORES	
э . 5.1	Memory organizations	
5.2	Management of the Control Flow Constants	
5.3	Management of the re-mapping tables	
	5.3.1 R400 Constant management	
	5.3.2 Proposal for R400LE constant management Error! Bookmark not	
	5.3.3 Dirty bits	
	5.3.4 Free List Block	
	5.3.5 De-allocate Block	
	5.3.6 Operation of Incremental model	
5.4		
5.5	Real Time Commands	
5.6	Constant Waterfalling	
6.	LOOPING AND BRANCHES	
6.1	The controlling state	
6.2	The Control Flow Program	18
6.2	0.2.1 Control flow instructions table	19
6.3	Implementation	
6.4	Data dependant predicate instructions	
6.5	HW Detection of PV,PS	
6.6	Register file indexing	
6.7	Debugging the Shaders	
	0.7.1 Method 1: Debugging registers	
	5.7.2 Method 2: Exporting the values in the GPRs	
7.	PIXEL KILL MASK	
8.	MULTIPASS VERTEX SHADERS (HOS) ERROR! BOOKMARK NOT D	
9. 10.	REGISTER FILE ALLOCATION FETCH ARBITRATION	
12.		
13.		
14.		
15.	IJ FORMAT	
15.1		
16.	STAGING REGISTERS	29

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

		ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
6	UU	24 September, 2001	4 September, 2015	GEN-CXXXXX-REVA	3 of 56
17.					
17.1	8				
	.1.1	i.			
	.1.2	Vertex exports:			
	.1.3				
17.2					
18. 18.1					
18.2		•			
19.	SPEC	IAL INTERPOLATION	MODES		33
19.1					
19.2 19.3					
	.3.1				
19 20.					
20.1				***************************************	
21.	XY AC	DRESS IMPORTS			35
21.1					
22.					
23.					
23.1					
23.2					
	.2.1				
	.2.2	seasoe			
	.2.3				
	.2.4		-		
_	.2.5	VGT to SQ : Vertex in	iterface		39
23	.2.6	SQ to SX: Control bus	3		
23	.2.7	SX to SQ : Output file	control		
23	.2.8	SQ to TP: Control bus	3		
23	.2.9	TP to SQ: Texture sta			
23	.2.10	SQ to SP: Texture sta		Error! Bookmark not	defined.
23	.2.11	SQ to SP: GPR and a	uto counter		
23	.2.12				
_	.2.13			ned with the SP data)	
	.2.13			Set/Kill set	
	.2.14				
	.2.15				
-					
	.2.17				
23	.2.18	SQ to CP: State repor	n		

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE				
24 September, 2001		4 September, 2015		4 of 56				
23.3 Exar	23.3 Example of control flow program execution							
24. OPEN	24. OPEN ISSUES							

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257870

ATI Ex. 2109 IPR2023-00922 Page 160 of 326



ORIGINATE DATE

24 September, 2001

Revision Changes:

Rev 0.1 (Laurent Lefebvre) Date: May 7, 2001

Rev 0.2 (Laurent Lefebvre) Date : July 9, 2001 Rev 0.3 (Laurent Lefebvre) Date : August 6, 2001 Rev 0.4 (Laurent Lefebvre) Date : August 24, 2001

Rev 0.5 (Laurent Lefebvre) Date : September 7, 2001 Rev 0.6 (Laurent Lefebvre) Date : September 24, 2001 Rev 0.7 (Laurent Lefebvre) Date : October 5, 2001

Rev 0.8 (Laurent Lefebvre) Date : October 8, 2001 Rev 0.9 (Laurent Lefebvre) Date : October 17, 2001

Rev 1.0 (Laurent Lefebvre) Date : October 19, 2001 Rev 1.1 (Laurent Lefebvre) Date : October 26, 2001

Rev 1.2 (Laurent Lefebvre) Date : November 16, 2001 Rev 1.3 (Laurent Lefebvre) Date : November 26, 2001 Rev 1.4 (Laurent Lefebvre) Date : December 6, 2001

Rev 1.5 (Laurent Lefebvre) Date : December 11, 2001

Rev 1.6 (Laurent Lefebvre) Date : January 7, 2002

Rev 1.7 (Laurent Lefebvre) Date : February 4, 2002 Rev 1.8 (Laurent Lefebvre) Date : March 4, 2002

Rev 1.9 (Laurent Lefebvre) Date : March 18, 2002 Rev 1.10 (Laurent Lefebvre) Date : March 25, 2002 Rev 1.11 (Laurent Lefebvre) Date : April 19, 2002 Rev 2.0 (Laurent Lefebvre) Date : April 19, 2002 EDIT DATE

4 September, 2015

First draft.

Changed the interfaces to reflect the changes in the SP. Added some details in the arbitration section. Reviewed the Sequencer spec after the meeting on August 3, 2001.

Added the dynamic allocation method for register file and an example (written in part by Vic) of the flow of pixels/vertices in the sequencer. Added timing diagrams (Vic)

Changed the spec to reflect the new R400 architecture. Added interfaces.

Added constant store management, instruction store management, control flow management and data dependant predication.

Changed the control flow method to be more flexible. Also updated the external interfaces.

Incorporated changes made in the 10/18/01 control flow meeting. Added a NOP instruction, removed the conditional_execute_or_jump. Added debug registers.

Refined interfaces to RB. Added state registers.

Added SEQ \rightarrow SP0 interfaces. Changed delta precision. Changed VGT \rightarrow SP0 interface. Debug Methods added.

Interfaces greatly refined. Cleaned up the spec.

Added the different interpolation modes.

Added the auto incrementing counters. Changed the VGT \rightarrow SQ interface. Added content on constant management. Updated GPRs.

Removed from the spec all interfaces that weren't directly tied to the SQ. Added explanations on constant management. Added $PA \rightarrow SQ$ synchronization fields and explanation.

Added more details on the staging register. Added detail about the parameter caches. Changed the call instruction to a Conditionnal_call instruction. Added details on constant management and updated the diagram.

Added Real Time parameter control in the SX interface. Updated the control flow section.

New interfaces to the SX block. Added the end of clause modifier, removed the end of clause instructions.

Rearangement of the CF instruction bits in order to ensure byte alignement.

Updated the interfaces and added a section on exporting rules.

Added CP state report interface. Last version of the spec with the old control flow scheme New control flow scheme

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE		
	24 September, 2001	4 September, 2015		6 of 56		
Rev 2.01 (Lau	rent Lefebvre)	Cha	nged slightly the control flow instru	uctions to		
Date : May 2,			v force jumps and calls.			
	rent Lefebvre)		ated the Opcodes. Added type fie	ld to the		
Date : May 13			stant/pred interface. Added Last fie			
,	,		→SP instruction load interface.			
Rev 2.03 (Lau	rent Lefebyre)	SP	interface updated to include p	redication		
Date : July 15,			mizations. Added the predicate			
Date Foury For		•	uctions.	ino otan		
Rev 2 04 (Lau	rent Lefebvre)		umented the new parameter generatio	n scheme		
Date :August 2	,	for XY coordinates points and lines STs.				
	rent Lefebvre)	Some interface changes and an architectural				
Date : Septem	2	change to the auto-counter scheme.				
	rent Lefebvre)	Widened the event interface to 5 bits. Some other				
Date : October		little typos corrected.				
	rent Lefebvre)	Loops, jumps and calls are now using a 13 b				
Date : October	,		ress which allows to jump and call			
	14, 2002		ind any control flow addresses (
			ires to be even anymore).	0003 1101		
Rev 2.08 (Lau	rent Lefebure)		ification updates after discussion with	Clav		
Date : October	,	Oldi	incution updates after discussion with	olay.		
Rev 2.09 (Lau		Corr	ected the SQ \rightarrow SP staging register into	orfaco		
Date : January		0011		snace.		
Rev 2.10 (Lau		۵ddi	ing R500 modifications			
Date : April 8,		Aud				
Date . April 0,	2000					

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257872

ATI Ex. 2109 IPR2023-00922 Page 162 of 326



1. Overview

The sequencer chooses four ALU threads (two from each bank), a vertex cache and a fetch thread to execute, and executes all of the instructions in a block before looking for a new clause of the same type. Two ALU threads are executed interleaved to hide the ALU latency. The arbitrator will give priority to older threads. There are two separate reservation stations, one for pixel vectors and one for vertices vectors. This way a pixel can pass a vertex and a vertex can pass a pixel.

There are also 2 separate ALU banks from which the SQ picks the ALU threads to be executed in parallel.

To support the shader pipe the sequencer also contains the shader instruction store, constant store, control flow constants and texture state. The height shader pipes also execute the same two instructions thus there is only one sequencer for the whole chip but it issues 2 instructions every four clocks.

The sequencer first arbitrates between vectors of 64 vertices that arrive directly from primitive assembly and vectors of 16 quads (64 pixels) that are generated in the scan converter.

The vertex or pixel program specifies how many GPRs it needs to execute. The sequencer will not start the next vector until the needed space is available in the GPRs.

Exhibit 2033.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257873

ATI Ex. 2109 IPR2023-00922 Page 163 of 326

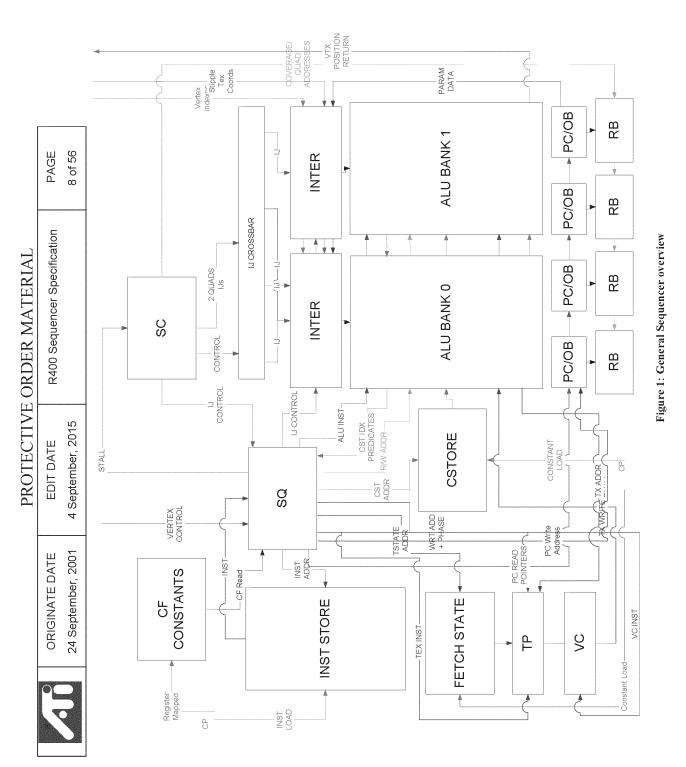


Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257874

ATI Ex. 2109 IPR2023-00922 Page 164 of 326

100	
	n

1.1 Top Level Block Diagram

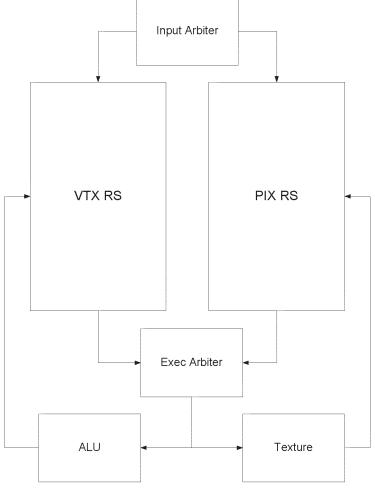


Figure 2: Reservation stations and arbiters

Under this new scheme, the sequencer (SQ) will only use one global state management machine per vector type (pixel, vertex) that we call the reservation station (RS).

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257875

ATI Ex. 2109 IPR2023-00922 Page 165 of 326 PROTECTIVE ORDER MATERIAL



4 September, 2015

R400 Sequencer Specification EDIT DATE

10 of 56 PAGE

1.2 Data Flow graph (SP)

R500 CONFIGURATION

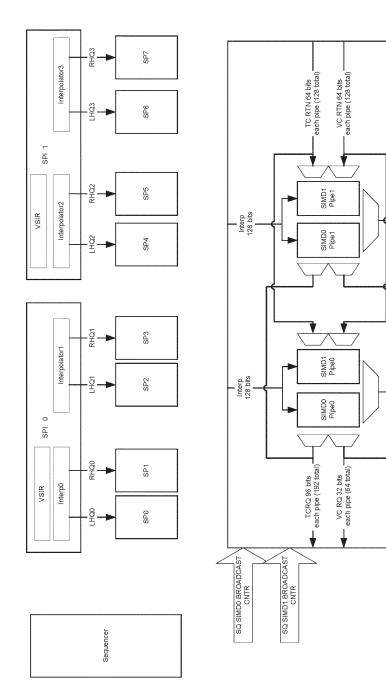


Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

Shared Export 32 bits

Shared Export 32 bits

Figure 3: The shader Pipe

AMD1044_0257876

ATI Ex. 2109 IPR2023-00922 Page 166 of 326

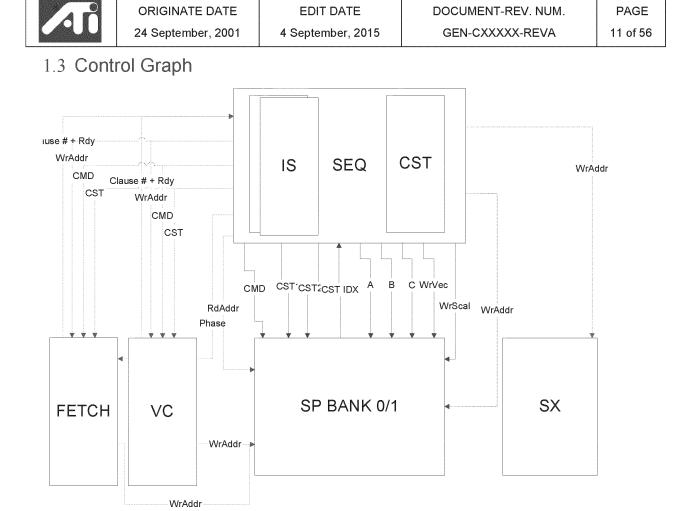


Figure 4: Sequencer Control interfaces

In green is represented the Fetch control interface, in red the ALU control interface, in blue the Interpolated/Vector control interface and in purple is the output file control interface.

2. Interpolated data bus

The interpolators contain an IJ buffer to pack the information as much as possible before writing it to the register file.

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257877

ATI Ex. 2109 IPR2023-00922 Page 167 of 326

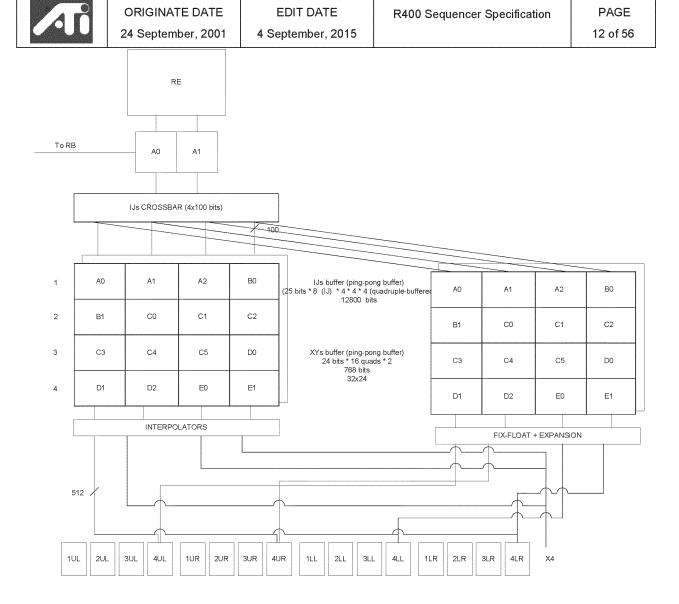


Figure 5: Interpolation buffers

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257878

ATI Ex. 2109 IPR2023-00922 Page 168 of 326

			2 T23					V V 32- 48- 35 51	V V 36- 52- 39 55	<pre>< < </pre> < < <pre>< </pre>	V V 44-60- 47 63	_ ×
			1 T2						> % %			$-\hat{F}$
			0 T2					~ 1 6 19	20- <	24- 27	31 ² 8 <	\rightarrow
			9 72					> °-0	> 7-4	<u></u> <u></u>	2°, 7°, <	
			11							Ш	ш	
			<u><u> </u></u>		****	-					â	
	PAGE	f 56	T17			Хü	×п		8	5	ß	
	PA	13 of 56	T16			ы	ш				BO	P2
T			T15			ÊO	Ш	5	D2			Ω
PROTECTIVE ORDER MATERIAL	NUM.	AV	T10 T11 T12 T13 T14 T15 T16 T17 T18 T19 T20 T21 T22 T23	ХĘ	∑ B2		Xg	C	C4	C5		
MAT	REV.	X-R	T13	5	D2		8	<u>8</u>				
ERN	ENT-	×	T12	Б	D2		8	AO	A1	A2		
)RD	DOCUMENT-REV. NUM.	GEN-CXXXXX-REVA	10 T11	D U E	×2	C5 ≺		f f		ОШ	μ	
VE (ă		T10		Q 4	C5	L C				8	
CTI			T9		2	C5			S	5	C2	
OTE	μ	, 2015	Т8	×ິ	28	≿ ວ	×Ω				BO	
PR	EDIT DATE	4 September, 2015	T7	IJ	00	5	C2	5	D2			Ò
		4 Sep	TG	ទ	CO	5	C3	ü	C4	C5		
			T5	ХE			₩ X	B1				
	ATE	2001	Τ4	ы			BG	AO	A1	A2		
	TED	nber,	T3	Б Ш			BO	51 8 ×∕	X 52- 55	X 56- 59	63 - X	
	ORIGINATE DATE	24 September, 2001	Т2		¥ ₹	₹X		X 32- 2 35	3 % X 39 % X	×48	× × × × 44 - 6	>
	0 R	24 8	F	AO	A1	A2		16- 19	23 ² 0 X	ХҮ 24- 27	31 - X 31 - 28	\rightarrow
	PC		TO	AO	A1	A2		-3 × 0-3	¥. 7-4	, ⇒ %	15 - 7	
			*************	o Sp	- SP	S P	ი ი	O S	<u>ч</u> С	N SP	ი ი	

Exhibit 2038 doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

Figure 6: Interpolation timing diagram

AMD1044_0257879

ATI Ex. 2109 IPR2023-00922 Page 169 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 2015		14 of 56

Above is an example of a tile the sequencer might receive from the SC. The write side is how the data get stacked into the XY and IJ buffers, the read side is how the data is passed to the GPRs. The IJ information is packed in the IJ buffer 4 quads at a time or two clocks. The sequencer allows at any given time as many as four quads to interpolate a parameter. They all have to come from the same primitive. Then the sequencer controls the write mask to the GPRs to write the valid data in.

3. Instruction Store

There is going to be two instruction stores for the whole chip. They will each contain 4096 instructions of 96 bits each.

They will be 1 port memories; Ports are allocated in this fashion (but not necessarily in this order):

ALU 0 SIMD0 CF	ALU 0 SIMD1 CF
ALU 0 SIMD0	ALU 0 SIMD1
ALU 1 SIMD0 CF	ALU 1 SIMD1 CF
ALU 1 SIMD0	ALU 1 SIMD1
Fetch CF	Fetch CF
Fetch	Fetch
VC CF	VC CF
VC	VC

Fetch and VC can steal one another's ports with stated resource having priority over its port (this is not really necessary for the R500 but will be for any derivative part because there will only be one instruction store).

Writes are opportunistic.

The instruction store is loaded by the CP thru the register mapped registers.

The VS_BASE and PS_BASE context registers are used to specify for each context where its shader is in the instruction memory.

For the Real time commands the story is quite the same but for some small differences. There are no wrap-around points for real time so the driver must be careful not to overwrite regular shader data. The shared code (shared subroutines) uses the same path as real time.

4. Sequencer Instructions

All control flow instructions instructions are handled by the sequencer only. The ALUs will perform NOPs during this time (MOV PV,PV, PS,PS) if they have nothing else to do.

5. Constant Stores

5.1 Memory organizations

A likely size for the ALU constant store is 1024x128 bits. The read BW from the ALU constant store is 128 bits/clock and the write bandwidth is 32 bits/clock (directed by the CP bus size not by memory ports).

The maximum logical size of the constant store for a given shader is 256 constants. Or 512 for the pixel/vertex shader pair. The size of the re-mapping table is 128 lines (each line addresses 4 constants). The write granularity is 4 constants or 512 bits. It takes 16 clocks to write the four constants. Real time requires 256 lines in the physical memory (this is physically register mapped).

There will be two of those memories and two of each remapping read memories.

The texture state is also kept in a similar memory. The size of this memory is 320x96 bits (128 texture states for regular mode, 32 states for RT). The memory thus holds 128 texture states (192 bits per state). The logical size exposes 32 different states total, which are going to be shared between the pixel and the vertex shader. The size of the re-mapping table to for the texture state memory is 32 lines (each line addresses 1 texture state lines in the real

Exhibit 2038.doc 81670 Bytes*** [©] ATI Confidential. Reference Copyright Notice on Cover Page [©] ***

AMD1044_0257880

ATI Ex. 2109 IPR2023-00922 Page 170 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 2015	GEN-CXXXXX-REVA	15 of 56

memory). The CP write granularity is 1 texture state lines (or 192 bits). The driver sends 512 bits but the CP ignores the top 320 bits. It thus takes 6 clocks to write the texture state. Real time requires 32 lines in the physical memory (this is physically register mapped).

The control flow constant memory doesn't sit behind a renaming table. It is register mapped and thus the driver must reload its content each time there is a change in the control flow constants. Its size is 320*32 because it must hold 8 copies of the 32 dwords of control flow constants and the loop construct constants must be aligned.

The constant re-mapping tables for texture state and ALU constants are logically register mapped for regular mode and physically register mapped for RT operation.

5.2 Management of the Control Flow Constants

The control flow constants are register mapped, thus the CP writes to the according register to set the constant, the SQ decodes the address and writes to the block pointed by its current base pointer (CF_WR_BASE). On the read side, one level of indirection is used. A register (SQ_CONTEXT_MISC.CF_RD_BASE) keeps the current base pointer to the control flow block. This register is copied whenever there is a state change. Should the CP write to CF after the state change, the base register is updated with the (current pointer number +1)% number of states. This way, if the CP doesn't write to CF the state is going to use the previous CF constants.

5.3 Management of the re-mapping tables

5.3.1 R400 Constant management

The sequencer is responsible to manage two re-mapping tables (one for the constant store and one for the texture state). On a state change (by the driver), the sequencer will broadside copy the contents of its re-mapping tables to a new one. We have 8 different re-mapping tables we can use concurrently.

The constant memory update will be incremental, the driver only need to update the constants that actually changed between the two state changes.

For this model to work in its simplest form, the requirement is that the physical memory MUST be at least twice as large as the logical address space + the space allocated for Real Time. In our case, since the logical address space is 512 and the reserved RT space can be up to 256 entries, the memory must be of sizes 1280 and above. Similarly the size of the texture store must be of 32*2+32 = 96 entries and above.

5.3.2 Dirty bits

Two sets of dirty bits will be maintained per logical address. The first one will be set to zero on reset and set when the logical address is addressed. The second one will be set to zero whenever a new context is written and set for each address written while in this context. The reset dirty is not set, then writing to that logical address will not require de-allocation of whatever address stored in the renaming table. If it is set and the context dirty is not set, then the physical address store needs to be de-allocated and a new physical address is necessary to store the incoming data. If they are both set, then the data will be written into the physical address held in the renaming for the current logical address. No de-allocation or allocation takes place. This will happen when the driver does a set constant twice to the same logical address between context changes. NOTE: It is important to detect and prevent this, failure to do it will allow multiple writes to allocate all physical memory and thus hang because a context will not fit for rendering to start and thus free up space.

5.3.3 Free List Block

A free list block that would consist of a counter (called the IFC or Initial Free Counter) that would reset to zero and incremented every time a chunk of physical memory is used until they have all been used once. This counter would be checked each time a physical block is needed, and if the original ones have not been used up, us a new one, else check the free list for an available physical block address. The count is the physical address for when getting a chunk from the counter.

Storage of a free list big enough to store all physical block addresses.

Maintain three pointers for the free list that are reset to zero. The first one we will call write_ptr. This pointer will identify the next location to write the physical address of a block to be de-allocated. Note: we can never free more

Exhibit 2038.doc 81670 Bytes*** [©] ATI Confidential. Reference Copyright Notice on Cover Page [©] ***

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 2015		16 of 56

physical memory locations than we have. Once recording address the pointer will be incremented to walk the free list like a ring.

The second pointer will be called stop_ptr. The stop_ptr pointer will be advanced by the number of address chunks de-allocates when a context finishes. The address between the stop_ptr and write_ptr cannot be reused because they are still in use. But as soon as the context using then is dismissed the stop ptr will be advanced.

The third pointer will be called read_ptr. This pointer will point will point to the next address that can be used for allocation as long as the read_ptr does not equal the stop_ptr and the IFC is at its maximum count.

5.3.4 De-allocate Block

This block will maintain a free physical address block count for each context. While in current context, a count shall be maintained specifying how many blocks were written into the free list at the write_ptr pointer. This count will be reset upon reset or when this context is active on the back and different than the previous context. It is actually a count of blocks in the previous context that will no longer be used. This count will be used to advance the write_ptr pointer to make available the set of physical blocks freed when the previous context was done. This allows the discard or de-allocation of any number of blocks in one clock.

5.3.5 Operation of Incremental model

The basic operation of the model would start with the write_ptr, stop_ptr, read_ptr pointers in the free list set to zero and the free list counter is set to zero. Also all the dirty bits and the previous context will be initialized to zero. When the first set constants happen, the reset dirty bit will not be set, so we will allocate a physical location from the free list counter because its not at the max value. The data will be written into physical address zero. Both the additional copy of the renaming table and the context zeros of the big renaming table will be updated for the logical address that was written by set start with physical address of 0. This process will be repeated for any logical address that are not dirty until the context changes. If a logical address is hit that has its dirty bits set while in the same context, both dirty bits would be set, so the new data will be over-written to the last physical address assigned for this logical address. When the first draw command of the context is detected, the previous context stored in the additional renaming table will be copied to the larger renaming table in the current (new) context location. Then the set constant logical address it replaced in the renaming table would be entered at the write_ptr pointer location on the free list and the write_ptr will be incremented. The de-allocation counter for the previous context (eight) will be incremented. This as set states come in for this context one of the following will happen:

- 1.) No dirty bits are set for the logical address being updated. A line will be allocated of the free-list counter or the free list at read_ptr pointer if read_ptr != to stop_ptr .
- Reset dirty set and Context dirty not set. A new physical address is allocated, the physical address in the renaming table is put on the free list at write_ptr and it is incremented along with the de-allocate counter for the last context.
- 3.) Context dirty is set then the data will be written into the physical address specified by the logical address.

This process will continue as long as set states arrive. This block will provide backpressure to the CP whenever he has not free list entries available (counter at max and stop_ptr == read_ptr). The command stream will keep a count of contexts of constants in use and prevent more than max constants contexts from being sent.

Whenever a draw packet arrives, the content of the re-mapping table is written to the correct re-mapping table for the context number. Also if the next context uses less constants than the current one all exceeding lines are moved to the free list to be de-allocated later. This happens in parallel with the writing of the re-mapping table to the correct memory.

Now preferable when the constant context leaves the last ALU clause it will be sent to this block and compared with the previous context that left. (Init to zero) If they differ than the older context will no longer be referenced and thus can be de-allocated in the physical memory. This is accomplished by adding the number of blocks freed this context to the stop_ptr pointer. This will make all the physical addresses used by this context available to the read_ptr allocate pointer for future allocation.

This device allows representation of multiple contexts of constants data with N copies of the logical address space. It also allows the second context to be represented as the first set plus some new additional data by just storing the delta's. It allows memory to be efficiently used and when the constants updates are small it can store multiple

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257882

ATI Ex. 2109 IPR2023-00922 Page 172 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 2015	GEN-CXXXXX-REVA	17 of 56

context. However, if the updates are large, less contexts will be stored and potentially performance will be degraded. Although it will still perform as well as a ring could in this case.

5.4 Constant Store Indexing

In order to do constant store indexing, the sequencer must be loaded first with the indexes (that come from the GPRs). There are 144 wires from the exit of the SP to the sequencer (9 bits pointers x 16 vertexes/clock).

MOVA R1.X,R2.X // Loads the sequencer with the content of R2.X, also copies the content of R2.X into R1.X ADD R3,R4,C0[R2.X]// Uses the state from the sequencer to add R4 to C0[R2.X] into R3

Note that we don't really care about what is in the brackets because we use the state from the MOVA instruction. R2.X is just written again for the sake of simplicity and coherency.

The storage needed in the sequencer in order to support this feature is 2*64*9 bits = 1152 bits.

The address register is a signed integer, which ranges from -256 to 255.

5.5 Real Time Commands

The real time commands constants are written by the CP using the register mapped registers allocated for RT. It works is the same way than when dealing with regular constant loads BUT in this case the CP is not sending a logical address but rather a physical address and the reads are not passing thru the re-mapping table but are directly read from the memory. The boundary between the two zones is defined by the CONST_EO_RT control register. Similarly, for the fetch state, the boundary between the two zones is defined by the TSTATE_EO_RT control register.

5.6 Constant Waterfalling

In order to have a reasonable performance in the case of constant store indexing using the address register, we are going to have the possibility of using the physical memory port for read only. This way we can read 1 constant per clock and thus have a worst-case waterfall mode of 1 vertex per clock. There is a small synchronization issue related with this as we need for the SQ to make sure that the constants where actually written to memory (not only sent to the sequencer) before it can allow the first vector of pixels or vertices of the state to go thru the ALUs. To do so, the sequencer keeps 8 bits (one per render state) and sets the bits whenever the last render state is written to memory and clears the bit whenever a state is freed.

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257883

ATI Ex. 2109 IPR2023-00922 Page 173 of 326

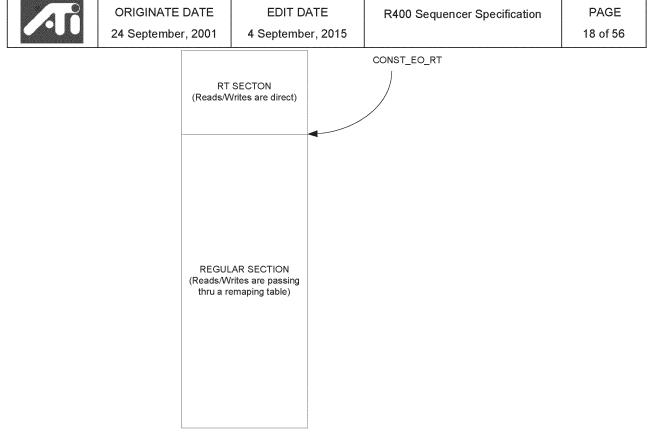


Figure 7: The Constant store

6. Looping and Branches

Loops and branches are planned to be supported and will have to be dealt with at the sequencer level. We plan on supporting constant loops and branches using a control program.

6.1 The controlling state.

The R400 controling state consists of:

Boolean[255:0] Loop_count[7:0][31:0] Loop_Start[7:0][31:0] Loop_Step[7:0][31:0]

That is 256 Booleans and 32 loops.

We have a stack of 4 elements for nested calls of subroutines and 4 loop counters to allow for nested loops.

This state is available on a per shader program basis.

6.2 The Control Flow Program

We'd like to be able to code up a program of the form:

1: Loop 2: Exec TexFetch

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 2015	GEN-CXXXXX-REVA	19 of 56
3:	TexFetch	•		
4:	ALU			
5:	ALU			
6:	TexFetch			
7:	End Loop			
8:	ALU Export			

But realize that 3: may be dependent on 2: and 4: is almost certainly dependent on 2: and 3:. Without clausing, these dependencies need to be expressed in the Control Flow instructions. Additionally, without separate 'texture clauses' and 'ALU clauses' we need to know which instructions to dispatch to the Texture Unit and which to the ALU unit. This information will be encapsulated in the flow control instructions.

Each control flow instruction will contain 2 bits of information for each (non-control flow) instruction:

a) ALU or Texture

b) Serialize Execution

(b) would force the thread to stop execution at this point (before the instruction is executed) and wait until all textures have been fetched. Given the allocation of reserved bits, this would mean that the count of an 'Exec' instruction would be limited to about 8 (non-control-flow) instructions. If more than this were needed, a second Exec (with the same conditions) would be issued.

Another function that relies upon 'clauses' is allocation and order of execution. We need to assure that pixels and vertices are exported in the correct order (even if not all execution is ordered) and that space in the output buffers are allocated in order. Additionally data can't be exported until space is allocated. A new control flow instruction:

Alloc <buffer select -- position, parameter, pixel or vertex memory. And the size required>.

would be created to mark where such allocation needs to be done. To assure allocation is done in order, the actual allocation for a given thread can not be performed unless the equivalent allocation for all previous threads is already completed. The implementation would also assure that execution of instruction(s) following the serialization due to the Alloc will occur in order -- at least until the next serialization or change from ALU to Texture. In most cases this will allow the exports to occur without any further synchronization. Only 'final' allocations or position allocations are guaranteed to be ordered. Because strict ordering is required for pixels, parameters and positions, this implies only a single alloc for these structures. Vertex exports to memory do not require ordering during allocation and so multiple 'allocs' may be done.

6.2.1 Control flow instructions table

Here is the revised control flow instruction set.

Note that whenever a field is marked as RESERVED, it is assumed that all the bits of the field are cleared (0).

	NOP						
47 44	4/44 43 42 0						
0000	Addressing	RESERVED					

This is a regular NOP.

	Execute								
47 44 43 40 34 33 28 27 16 15 12 11 0									
0001	Addressing	RESERV	Vertex	Instructions type + serialize (6	Count	Exec Address			
	_	ED	Cache	instructions)					

	Execute_End								
47 44	43	40 34	33 28	2716	1512	11 0			
0010	0010 Addressing RESERV Vertex		Instructions type + serialize (6	Count	Exec Address				
	ED Cache		instructions)						

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257885

ATI Ex. 2109 IPR2023-00922 Page 175 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 2015		20 of 56

Execute up to 6 instructions at the specified address in the instruction memory. The Instruction type field tells the sequencer the type of the instruction (LSB) (1 = Texture, 0 = ALU and whether to serialize or not the execution (MSB) (1 = Serialize, 0 = Non-Serialized). If the corresponding VC bit is set then VC is used instead of TP/ALU. If Execute_End this is the last execution block of the shader program.

Vertex Cache	Serialize	Instruction Type (Resource)	
0	0	0	: ALU instruction, not yielding
0	0	1	: ALU instruction, yielding
0	1	0	: Texture instruction, not yielding
0	1	1	: Texture instruction, yielding
1	0	0	: Vertex cache instruction, not yielding
1	0	1	: Vertex cache instruction, yielding
1	1	0	: Vertex cache instruction, not yielding
1	1	1	: Vertex cache instruction, yielding

	Conditional_Execute								
47 44	4744 43 42 4134 3328 2716 1512 110								
0011	Addressing	Condition	Boolean address	Vertex Cache	Instructions type + serialize (6 instructions)	Count	Exec Address		

	Conditional_Execute_End								
47 44	43	42	41 34	3328	2716	1512	11 0		
0100	Addressing	Condition	Boolean address	Vertex Cache	Instructions type + serialize (6 instructions)	Count	Exec Address		

If the specified Boolean (8 bits can address 256 Booleans) meets the specified condition then execute the specified instructions (up to 9 instructions). If the condition is not met, we go on to the next control flow instruction. If Conditional_Execute_End and the condition is met, this is the last execution block of the shader program.

	Conditional_Execute_Predicates									
47 44	43	42	41 36	35 34	3328	2716	1512	11 0		
0101	Addressing	Condition	RESERVED	Predicate vector	Vertex Cache	Instructions type + serialize (6 instructions)	Count	Exec Address		

	Conditional_Execute_Predicates_End									
47 44	43	42	41 36	35 34	3328	2716	1512	11 0		
0110	Addressing	Condition	RESERVED	Predicate vector	Vertex Cache	Instructions type + serialize (6 instructions)	Count	Exec Address		

Check the AND/OR of all current predicate bits. If AND/OR matches the condition execute the specified number of instructions. We need to AND/OR this with the kill mask in order not to consider the pixels that aren't valid. If the condition is not met, we go on to the next control flow instruction. If Conditional_Execute_Predicates_End and the condition is met, this is the last execution block of the shader program.

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE DATE		ORIGINATE DATE EDIT DATE			DOCUMENT-REV. NUM.			
24 September, 2		otember, 200	1 4 Sept	ember, 2015		GEN-CXXXXX-	21 of 56		
		*****	Conditional_Ex	xecute_Predic	cates_No_	Stall	******		
47 44	43	42	41 36	35 34	3328	2716	1512	11 0	
1101	Addressing	Condition	RESERVED	Predicate vector	Vertex Cache	Instructions type + serialize (6 instructions)	Count	Exec Address	

	Conditional_Execute_Predicates_No_Stall_End							
47 44	43	42	41 36	35 34	3328	2716	1512	11 0
1110	Addressing	Condition	RESERVED	Predicate vector	Vertex Cache	Instructions type + serialize (6	Count	Exec Address
						instructions)		

Same as Conditionnal_Execute_Predicates but the SQ is not going to wait for the predicate vector to be updated. You can only set this in the compiler if you know that the predicate set is only a refinement of the current one (like a nested if) because the optimization would still work.

Loop_Start					
47 44	43	42 21	20 16	1513	12 0
0111	Addressing	RESERVED	loop ID	RESERVED	Jump address

Loop Start. Compares the loop iterator with the end value. If loop condition not met jump to the address. Forward jump only. Also computes the index value. The loop id must match between the start to end, and also indicates which control flow constants should be used with the loop.

	Loop_End								
4744	43	42	41 36	3534	33 22	21	20 16	1513	12 0
1000	Addressing	Cond	RESERVED	Predicate	RESERVED	Pred	loop ID	RESERVED	start
				Vector		break			address

Loop end. Increments the counter by one, compares the loop count with the end value. If loop condition met, continue, else, jump BACK to the start of the loop. If predicate break != 0, then compares predicate vector n (specified by predicate Vector) to condition. If all bits meet condition then break the loop.

The way this is described does not prevent nested loops, and the inclusion of the loop id make this easy to do.

Conditionnal_Call						
47 44	43 42 41 34 33 14 13 12 0					12 0
1001	Addressing	Condition	Boolean address	RESERVED	Force Call	Jump address

If the condition is met, jumps to the specified address and pushes the control flow program counter on the stack. If force call is set the condition is ignored and the call is made always.

	Return				
47 44	43 42 0				
1010	Addressing	RESERVED			

Pops the topmost address from the stack and jumps to that address. If nothing is on the stack, the program will just continue to the next instruction.

	Conditionnal_Jump						
47 44	43	42	41 34	33	32 14	13	12 0
1011	Addressing	Condition	Boolean	FW only	RESERVED	Force Jump	Jump address
			address				

If force jump is set the condition is ignored and the jump is made always. If FW only is set then only forward jumps are allowed.

Exhibit 2038.doc 81670 Bytes*** [©] ATI Confidential. Reference Copyright Notice on Cover Page [©] ***



PAGE 22 of 56

Allocate					
47 44	43	4241	40	39 3	20
1100	Debug	Buffer Select	No Serial	RESERVED	Size

Buffer Select takes a value of the following:

01 – position export (ordered export)

10 – parameter cache or pixel export (ordered export)

11 – pass thru (out of order exports).

Size field is only used to reserve space in the export buffer for pass thru exports. Valid values are 1 (1 line) thru 9 (9 lines). It should be determined by the compiler/assembler by taking max index used +1.

If debug is set this is a debug alloc (ignore if debug DB_ON register is set to off).

By default the serial bit is set on an alloc. If the No Serial bit is asserted then the serial bit won't be set in the SQ.

6.3 Implementation

The envisioned implementation has a buffer that maintains the state of each thread. A thread lives in a given location in the buffer during its entire life, but the buffer has FIFO qualities in that threads leave in the order that they enter. Actually two buffers are maintained -- one for Vertices and one for Pixels. The intended implementation would allow for:

16 entries for vertices 48 entries for pixels.

From each buffer, arbitration logic attempts to select 1 thread for the texture unit and 2 (interleaved) thread for the ALU unit. Once a thread is selected it is read out of the buffer, marked as invalid, and submitted to appropriate execution unit. It is returned to the buffer (at the same place) with its status updated once all possible sequential instructions have been executed. A switch from ALU to TEX or visa-versa or a Serialize_Execution modifier forces the thread to be returned to the buffer.

Each entry in the buffer will be stored across two physical pieces of memory - most bits will be stored in a 1 read port device. Only bits needed for thread arbitration will be stored in a highly multi-ported structure. The bits kept in the 1 read port device will be termed 'state'. The bits kept in the multi-read ported device will be termed 'status'.

'State Bits' needed include:

- 1. Control Flow Instruction Pointer (13 bits),
- 2. Execution Count Marker 4 bits),
- 3. Loop Iterators (4x9 bits),
- 4. Loop Counters (4x9 bits),
- 5. Call return pointers (4x13 bits),
- 6. Predicate Bits (64 bits),
- 7. Export ID (4 bits),
- 8. Parameter Cache base Ptr (7 bits),
- 9. GPR Base Ptr (8 bits),
- 10. Context Ptr (3 bits).
- 11. LOD corrections (6x16 bits)
- 12. Valid bits (64 bits)
- 13. RT (1 bit) Signifies that this thread is a Real Time thread. This bit must be sent to the Constant store state machine when reading it.

Absent from this list are 'Index' pointers. These are costly enough that I'm presuming that they are instead stored in the GPRs. The first seven fields above (Control Flow Ptr, Execution Count, Loop Counts, call return ptrs, Predicate bits, PC base ptr and export ID) are updated every time the thread is returned to the buffer based on how much

Exhibit 2038.doc 81670 Bytes*** [©] ATI Confidential. Reference Copyright Notice on Cover Page [©] ***

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 2015	GEN-CXXXXX-REVA	23 of 56

progress has been mode on thread execution. GPR Base Ptr, Context Ptr and LOD corrections are unchanged throughout execution of the thread.

'Status Bits' needed include:

- Valid Thread
- ALU engine needed
- Texture engine needed
- VC engine needed
- Texture Reads are outstanding
- VC Reads are outstanding
- Alu bank (0/1)
- Waiting on Texture Read to Complete
- Allocation Wait (2 bits)
- 00 No allocation needed
- 01 Position export allocation needed (ordered export)
- 10 Parameter or pixel export needed (ordered export)
- 11 pass thru (out of order export)
- Allocation Size (4 bits)
- Position Allocated
- Mem/Color Allocated
- First thread of a new context
- Event thread (NULL thread that needs to trickle down the pipe)
- Last (1 bit)
- Pulse SX (1 bit)

All of the above fields from all of the entries go into the arbitration circuitry. The arbitration circuitry will select a winner for both the Texture Engine and for the ALU engine. There are actually two sets of arbitration -- one for pixels and one for vertices. A final selection is then done between the two. But the rest of this implementation summary only considers the 'first' level selection which is similar for both pixels and vertices.

Texture arbitration requires no allocation or ordering so it is purely based on selecting the 'oldest' thread that requires the Texture Engine.

ALU arbitration is a little more complicated. First, only threads where either of Texture_Reads_outstanding or Waiting_on_Texture_Read_to_Complete are '0' are considered. Then if Allocation_Wait is active, these threads are further filtered based on whether space is available. If the allocation is position allocation, then the thread is only considered if all 'older' threads have already done their position allocation (position allocated bits set). If the allocation is parameter or pixel allocation, then the thread is only considered if it is a parameter or pixel allocation, then the thread is only considered if it is a parameter or pixel or position allocation, has its First_thread_of_a_new_context bit set and would cause ALU interleaving with another thread performing the same parameter or pixel or position allocation. Finally the 'oldest' of the threads that pass through the above filters is selected. If the thread needed to allocate, then at this time the allocation is done, based on Allocation_Size. If a thread has its "last" bit set, then it is also removed from the buffer, never to return.

If I now redefine 'clauses' to mean 'how many times the thread is removed from the thread buffer for the purpose of exection by either the ALU or Texture engine', then the minimum number of clauses needed is 2 -- one to perform the allocation for exports (execution automatically halts after an 'Alloc' instruction) (but doesn't performs the actual allocation) and one for the actual ALU/export instructions. As the 'Alloc' instruction could be part of a texture clause (presumably the final instruction in such a clause), a thread could still execute in this minimal number of 2 clauses, even if it involved texture fetching.

The Texture_Reads_Outstanding and VC_reads_Outstanding bits tell the SQ that a texture or VC read is outstanding. In this case, if we encounter a serial bit we need to wait until both resources are free (pending = 0) in order to proceed.

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044 0257889

ATI Ex. 2109 IPR2023-00922 Page 179 of 326



PAGE 24 of 56

6.4 Data dependant predicate instructions

Data dependant conditionals will be supported in the R400. The only way we plan to support those is by supporting three vector/scalar predicate operations of the form:

PRED_SETE_PUSH - similar to SETE except that the result is 'exported' to the sequencer. PRED_SETNE_PUSH - similar to SETNE except that the result is 'exported' to the sequencer. PRED_SETGT_PUSH - similar to SETGT except that the result is 'exported' to the sequencer PRED_SETGTE_PUSH - similar to SETGTE except that the result is 'exported' to the sequencer

For the scalar operations only we will also support the two following instructions:

PRED_SETÉ PRED_SETNE PRED_SETGT PRED_SET_INV PRED_SET_POP PRED_SET_CLR PRED_SET_RESTORE

Details about actual implementation of these opcodes are in the shader pipe architectural spec.

The export is a single bit - 1 or 0 that is sent using the same data path as the MOVA instruction. The sequencer will maintain 1 set of 64 bits predicate vectors (in fact 2 sets because we interleave two programs but only 1 will be exposed) and use it to control the write masking. This predicate is maintained across clause boundaries.

Then we have two conditional execute bits. The first bit is a conditional execute "on" bit and the second bit tells us if we execute on 1 or 0. For example, the instruction:

P0_ADD_# R0,R1,R2

Is only going to write the result of the ADD into those GPRs whose predicate bit is 0. Alternatively, P1_ADD_# would only write the results to the GPRs whose predicate bit is set. The use of the P0 or P1 without precharging the sequencer with a PRED instruction is undefined.

6.5 HW Detection of PV,PS

Because of the control program, the compiler cannot detect statically dependant instructions. In the case of nonmasked writes and subsequent reads the sequencer will insert uses of PV,PS as needed. This will be done by comparing the read address and the write address of consecutive instructions. For masked writes, the sequencer will insert detect wich channels to read from the GPRs and which ones to read from the PV/PS.

6.6 Register file indexing

Because we can have loops in fetch clause, we need to be able to index into the register file in order to retrieve the data created in a fetch clause loop and use it into an ALU clause. The instruction will include the base address for register indexing and the instruction will contain these controls:

Bit 6	
0	'absolute register'
1	'relative register'
0	'previous vector'
1	previous scalar
	Bit 6 0 1 0 1

In the case of an absolute register we just take the address as is. In the case of a relative register read we take the base address and we add to it the loop_index and this becomes our new address that we give to the shader pipe.

The sequencer is going to keep a loop index computed as such:

Index = Loop_iterator*Loop_step + Loop_start.

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 2015	GEN-CXXXXX-REVA	25 of 56

We loop until loop_iterator = loop_count. Loop_step is a signed value [-128...127]. The computed index value is a 10 bit counter that is also signed. Its real range is [-256,256]. The tenth bit is only there so that we can provide an out of range value to the "indexing logic" so that it knows when the provided index is out of range and thus can make the necessary arrangements.

6.7 Debugging the Shaders

In order to be able to debug the pixel/vertex shaders efficiently, we provide 2 methods.

6.7.1 *Method 1: Debugging registers*

Current plans are to expose 2 debugging, or error notification, registers:

1. address register where the first error occurred

2. count of the number of errors

The sequencer will detect the following groups of errors:

- count overflow

- constant indexing overflow

- register indexing overflow

Compiler recognizable errors:

- jump errors

relative jump address > size of the control flow program

call stack
 call with stack full
 return with stack empty

With all the other errors, program can continue to run, potentially to worst-case limits.

If indexing outside of the constant or the register range, causing an overflow error, the hardware is specified to return the value with an index of 0. This could be exploited to generate error tokens, by reserving and initializing the 0th register (or constant) for errors.

{ISSUE : Interrupt to the driver or not?}

6.7.2 Method 2: Exporting the values in the GPRs

1) The sequencer will have a debug active, count register and an address register for this mode.

Under the normal mode execution follows the normal course.

Under the debug mode it is assumed that the program is always exporting n debug vectors and that all other exports to the SX block (but for position) will be turned off (changed into NOPs) by the sequencer (even if they occur before the address stated by the ADDR debug register).

7. Pixel Kill Mask

A vector of 64 bits is kept by the sequencer per group of pixels/vertices. Its purpose is to optimize the texture fetch requests and allow the shader pipe to kill pixels using the following instructions:

MASK_SETE MASK_SETNE MASK_SETGT MASK_SETGTE

8. Register file allocation

The register file allocation for vertices and pixels can either be static or dynamic. In both cases, the register file in managed using two round robins (one for pixels and one for vertices). In the dynamic case the boundary between

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257891

ATI Ex. 2109 IPR2023-00922 Page 181 of 326

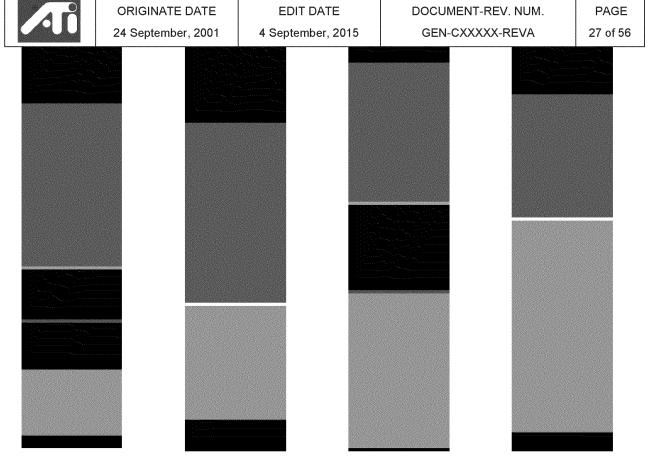
ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 2015		26 of 56

pixels and vertices is allowed to move, in the static case it is fixed to 128-VERTEX_REG_SIZE for vertices and PIXEL_REG_SIZE for pixels.

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257892

ATI Ex. 2109 IPR2023-00922 Page 182 of 326



Above is an example of how the algorithm works. Vertices come in from top to bottom; pixels come in from bottom to top. Vertices are in orange and pixels in green. The blue line is the tail of the vertices and the green line is the tail of the pixels. Thus anything between the two lines is shared. When pixels meets vertices the line turns white and the boundary is static until both vertices and pixels share the same "unallocated bubble". Then the boundary is allowed to move again. The numbering of the GPRs starts from the bottom of the picture at index 0 and goes up to the top at index 127.

9. Fetch Arbitration

The fetch arbitration logic chooses one of the n potentially pending fetch clauses to be executed. The choice is made by looking at the Vs and Ps reservation stations and picking the first one ready to execute. Once chosen, the clause state machine will send one 2x2 fetch per clock (or 4 fetches in one clock every 4 clocks) until all the fetch instructions of the clause are sent. This means that there cannot be any dependencies between two fetches of the same clause.

The arbitrator will not wait for the fetches to return prior to selecting another clause for execution. The fetch pipe will be able to handle up to X(?) in flight fetches and thus there can be a fair number of active clauses waiting for their fetch return data.

10. VC Arbitration

The VC arbitration logic chooses one of the n potentially pending VC clauses to be executed. The choice is made by looking at the Vs and Ps reservation stations and picking the first one ready to execute. Once chosen, the clause state machine will send one 2x2 fetch per clock (or 4 fetches in one clock every 4 clocks) until all the fetch instructions of the clause are sent. This means that there cannot be any dependencies between two fetches of the same clause.

Exhibit 2033.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257893

ATI Ex. 2109 IPR2023-00922 Page 183 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
24 September, 2001	4 September, 2015		28 of 56

The arbitrator will not wait for the fetches to return prior to selecting another clause for execution. The VC pipe will be able to handle up to X(?) in flight VC fetches and thus there can be a fair number of active clauses waiting for their fetch return data.

11. ALU Arbitration

ALU arbitration proceeds in almost the same way than fetch arbitration. The ALU arbitration logic chooses one of the n potentially pending ALU clauses to be executed. The choice is made by looking at the Vs and Ps reservation stations and picking the first one ready to execute. There are two ALU arbitres, one for the even clocks and one for the odd clocks. For example, here is the sequencing of two interleaved ALU clauses (E and O stands for Even and Odd sets of 4 clocks):

Einst0 Oinst0 Einst1 Oinst1 Einst2 Oinst2 Einst0 Oinst3 Einst1 Oinst4 Einst2 Oinst0... Proceeding this way hides the latency of 8 clocks of the ALUs. Also note that the interleaving also occurs across clause boundaries.

12. Handling Stalls

When the output file is full, the sequencer prevents the ALU arbitration logic from selecting the last clause (this way nothing can exit the shader pipe until there is place in the output file. If the packet is a vertex packet and the position buffer is full (POS_FULL) then the sequencer also prevents a thread from entering an exporting clause. The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU arbiter will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

12.1 SP stall conditions

12.1.1 PS Stalls

None.

12.1.2 PV Stalls

None.

13. Content of the reservation station FIFOs

The reservation FIFOs contain the state of the vector of pixels and vertices. We have two sets of those: one for pixels, and one for vertices. They contain 3 bits of Render State 7 bits for the base address of the GPRs, some bits for LOD correction and coverage mask information in order to fetch fetch for only valid pixels, the quad address.

14. The Output File

The output file is where pixels are put before they go to the RBs. The write BW to this store is 256 bits/clock. Just before this output file are staging registers with write BW 512 bits/clock and read BW 256 bits/clock. The staging registers are 4x128 (and there are 16 of those on the whole chip).

15. IJ Format

The IJ information sent by the PA is of this format on a per quad basis:

We have a vector of IJ's (one IJ per pixel at the centroid of the fragment or at the center of the pixel depending on the mode bit). All pixel's parameters are always interpolated at full 20x24 mantissa precision.

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257894

ATI Ex. 2109 IPR2023-00922 Page 184 of 326

	ORIGINATE DATE	EDIT DAT	E	DOCUM	ENT-REV. NUM.	PAGE	
	24 September, 2001 4 September, 2		2015	GEN-C	XXXXX-REVA	29 of 56	
P 0 = A + I	J(0) * (B - A) + J(0)	*(C - A)					
P1 = A + I	(1) * (B - A) + J(1) *	(C - A)	P0	P1			
P2 = A + I(2) * (B - A) + J(2) * (C - A)							
P3 = A + I	J(3) * (B - A) + J(3) =	(C - A)					
			P2	P3			

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

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

Total number of bits : 20*8 + 4*8 + 4*2 = 200.

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 +/- 1024.

15.1 Interpolation of constant attributes

Because of the floating point imprecision, we need to take special provisions if all the interpolated terms are the same or if two of the terms are the same.

16. Staging Registers

In order for the reuse of the vertices to be 14, the sequencer will have to re-order the data sent IN ORDER by the VGT for it to be aligned with the parameter cache memory arrangement. Given the following group of vertices sent by the VGT:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 || 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 || 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 || 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

The sequencer will re-arrange them in this fashion:

0 1 2 3 16 17 18 19 32 33 34 35 48 49 50 51 || 4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 || 8 9 10 11 24 25 26 27 40 41 42 43 56 57 58 59 || 12 13 14 15 28 29 30 31 44 45 46 47 60 61 62 63

The || markers show the SP divisions. In the event a shader pipe is broken, the SQ is responsible to insert padding to account for the missing pipe. For example, if SP1 is broken, vertices 4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 will not be sent by the VGT to the SQ **AND** the SQ is responsible to "jump" over these vertices in order for no valid vertices to be sent to an invalid SP.

The most straightforward, *non-compressed* interface method would be to convert, in the VGT, the data to 32-bit floating point prior to transmission to the VSISRs. In this scenario, the data would be transmitted to (and stored in) the VSISRs in full 32-bit floating point. This method requires three 24-bit fixed-to-float converters in the VGT. Unfortunately, it also requires and additional 3,072 bits of storage across the VSISRs. This interface is illustrated in Figure 9. The area of the fixed-to-float converters and the VSISRs for this method is roughly estimated as 0.759sqmm using the R300 process. The gate count estimate is shown in Figure 8.

Exhibit 2033.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257895

ATI Ex. 2109 IPR2023-00922 Page 185 of 326



Basis for 8-deep Latch Memory (fron	n R300)		
8x24-bit	11631	μ^2	60.57813 μ^2 per bit
Area of 96x8-deep Latch Memory	46524	μ^2	
Area of 24-bit Fix-to-float Converter $4712 \mu^2 per converter$			
	Block	Quantity	Area
Method 1	DIOON		
Method 1	F2F	3	14136
Method 1		3 16	14136 744384

Figure 8: Area Estimate for VGT to Shader Interface

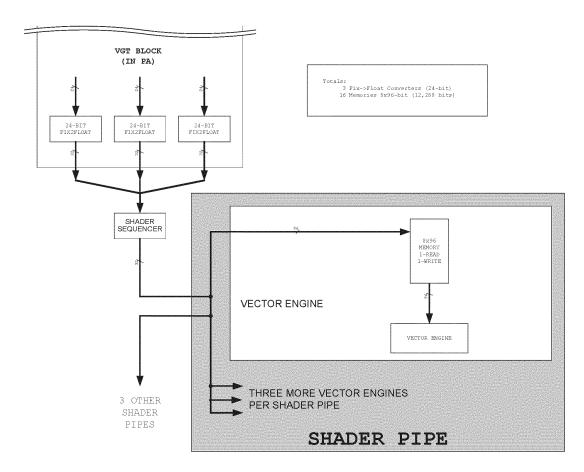


Figure 9:VGT to Shader Interface

17. The parameter cache

The parameter cache is where the vertex shaders export their data. It consists of 16 128x128 memories (1R/1W). The reuse engine will make it so that all vertexes of a given primitive will hit different memories. The allocation

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257896

ATI Ex. 2109 IPR2023-00922 Page 186 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 2015	GEN-CXXXXX-REVA	31 of 56

method for these memories is a simple round robin. The parameter cache pointers are mapped in the following way: 4MSBs are the memory number and the 7 LSBs are the address within this memory.

MEMORY NUMBER	ADDRESS
4 bits	7 bits

The PA generates the parameter cache addresses as the positions come from the SQ. All it needs to do is keep a Current_Location pointer (7 bits only) and as the positions comes increment the memory number. When the memory number field wraps around, the PA increments the Current_Location by VS_EXPORT_COUNT (a snooped register from the SQ). As an example, say the memories are all empty to begin with and the vertex shader is exporting 8 parameters per vertex (VS_EXPORT_COUNT = 8). The first position received is going to have the PC address 00000000000 the second one 00010000000, third one 0010000000 and so on up to 11110000000. Then the next position received (the 17th) is going to have the address 00000001000, the 18th 00010001000, the 19th 00100001000 and so on. The Current_location is NEVER reset BUT on chip resets. The only thing to be careful about is that if the SX doesn't send you a full group of positions (<64) then you need to fill the address space so that the next group starts correctly aligned (for example if you receive only 33 positions then you need to add 2*VS_EXPORT_COUNT to Current_Location and reset the memory count to 0 before the next vector begins).

17.1 Export restrictions

17.1.1 Pixel exports:

Pixels can export 1,2,3 or 4 color buffers to the SX(+z). The exports will be done in order. The exports will always be ordered to the SX.

17.1.2 Vertex exports:

Position or parameter caches can be exported in any order in the shader program. It is always better to export posistion as soon as possible. Position has to be exported in a single export block (no texture instructions can be placed between the exports). Parameter cache exports can be done in any order with texture instructions interleaved. The exports will always be allocated in order to the SX.

17.1.3 Pass thru exports:

Pass thru exports have to be done in groups of the form:

```
Alloc 1 thru 5 (max export offset + 1, for example if using EM4 alloc size 5) Execute ALU(ADDR) ALU(DATA) ALU(DATA) ALU(DATA)...
```

When exporting to more than EM0, one MUST write to EM4 also (the write may be predicated if you don't need the export). This is used to initialize the buffers in the SX.

There cannot be any serialize bits set OR texture Reads between the EA and the last EM.

Memory exports will be surfaced using a macro extension; here is what needs to happen inside the macro:

The macro needs to create a special constant of the form:

Stream ID constant:

- = Integer that holds BaseAddressInBytes/4 in bits (29:0). Bits 31:30 should be 0b01.
- .y = 2**23

.X

.z = Integer that holds register field data. Note that this data must be organized so that it

always represents a 'valid' floating point number, with the relevant bits in (23 - 0); One way of doing this would be to take the 23 bits and add 2**23.

.w = max index value + 2**23

Output to EXaddress:

.x = Base of array (in low 30 bits)/4

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***



- .y = Index value (in low 23 bits)
- .z = Register Field data (in low 23 bits)
- .w = Max Index value (in low 23 bits)

Also Assume that C0:

.x = 0.0 .y = 1.0

The Macro expansion would be as follows:

MULADD EA = Rindex.xxxx,C0.xyxx,CstreamID; MOV EMx (x = 0 thru 4) = Rdata;

The SX will check for invalid writes and mask out the data so it won't be written to memory. Invalid writes are:

- 1) Index value >= Max Index value
- 2) bit 31 != 0 (negative index)
- 3) bits [30:23] != 23 + IEEE_EXP_BIAS (127) (meaning the index was too big to be represented using 23 bits)

They cannot have texture instructions interleaved in the export block. These exports are not guaranteed to be ordered.

Also, when doing a pass thru export, the shader must still do either a position and PC export (if Vertex) or a color export (if Pixel). The pass thru export can occur anywhere in any shader program and thus can be used to debug. There can be any number of pass thru export blocks throughout the pixel or vertex shader or both.

17.2 Arbitration restrictions

Here are the Sequencer arbitration restrictions:

- 1) Cannot execute a serialized thread if the corresponding texture pending bit and VC pending is set
- 2) Cannot allocate position if any older thread has not allocated position
- 3) Cannot execute a texture clause if texture reads are pending
- 4) Cannot execute a VC clause if VC reads are pending
- 5) Cannot execute last if texture pending (even if not serial)
- Cannot allocate if not last for color exports.
- 7) Cannot allocate if not last for PC exports.

18. Export Types

The export type (or the location where the data should be put) is specified using the destination address field in the ALU instruction. Here is a list of all possible export modes:

18.1 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

Exhibit 2038.doc 81670 Bytes*** [©] ATI Confidential. Reference Copyright Notice on Cover Page [©] ***



(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.)

18.2 Pixel Shading

1

2

- Color for buffer 0 (primary)
- Color for buffer 1
- Color for buffer 2 - Color for buffer 3
- 3 Color for buff 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 'alpha' component)
- 62:63 Empty

19. Special Interpolation modes

19.1 Real time commands

We are unable to use the parameter memory since there is no way for a command stream to write into it. Instead we need to add three 4x128 memories (one for each of three vertices x 4 interpolants). These will be mapped onto the register bus and written by type 0 packets, and output to the the parameter busses (the sequencer and/or PA need to be able to address the reatime parameter memory as well as the regular parameter store. This mode is triggered by the primitive type: REAL TIME. The actual memories are in the in the SX blocks. The parameter data memories are hooked on the RBBM bus and are loaded by the CP using register mapped memory.

19.2 Sprites/ XY screen coordinates/ FB information

XY screen coordinates may be needed in the shader program. This functionality is controlled by the param_gen register (in SQ) in conjunction with the SND_XY register (in SC) and the param_gen_pos. Also it is possible to send the faceness information (for OGL front/back special operations) to the shader using the same control register. Here is a list of all the modes and how they interact together:

The Data is going to be written in the register specified by the param_gen_pos register.

Param_Gen disable, snd_xy disable = No modification Param_Gen disable, snd_xy enable = No modification Param_Gen enable, snd_xy disable = Sign(faceness)garbage,(Sign Point)garbage,Sign(Line)s, t Param_Gen enable, snd_xy enable = Sign(faceness)screenX,(Sign Point)screenY,Sign(Line)s, t

In other words,

The generated vector is (X in RED, Y in GREEN, S in BLUE and T in ALPHA): X,Y,S,T These values are always supposed to be positive and any shader use of them should use the ABS function (as their sign bits will now be used for flags). SignX = BackFacing SignY = Point Primitive SignS = Line Primitive SignT = currently unused as a flag.

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***



If !Point & !Line, then it is a Poly.

I would assume that one implementation which allows for generic texture lookup (using 3D maps) for poly stipple and AA for the driver would be

}

19.3 Auto generated counters

In the cases we are dealing with multipass shaders, the sequencer is going to generate a vector count to be able to both use this count to write the 1st pass data to memory and then use the count to retrieve the data on the 2nd pass. The count is always generated in the same way but it is passed to the shader in a slightly different way depending on the shader type (pixel or vertex). This is toggled on and off using the GEN_INDEX_PIX/VTX register. The sequencer is going to keep two counters, one for pixels and one for vertices. Every time a full vector of vertices or pixels is written to the GPRs the counter is incremented. Every time a RST_PIX_COUNT or RST_VTX_COUNT events are received, the corresponding counter is reset. While there is only one count broadcast to the GPRs, the LSB are hardwired to specific values making the index different for all elements in the vector. Since the count must be different for all pixels/vertices and the 4 LSBs (16 positions) are hardwired to the corresponding shader unit the SQ has two choices:

1) Maintain a 19 bit counter that counts the vectors of 64. In this case the phase must be appended to the count before the count is broadcast to the SPs:

2) Maintain a 21 bits counter that counts sub-vectors of 16. In this case only the counter is sent to the Sps:

Countor (Of hite)	Llowely size of (A late)
Counter (21 bits)	margwireg (4 bits)

19.3.1 Vertex shaders

In the case of vertex shaders, if GEN_INDEX_VTX is set, the data will be put into the x field of the third register (it means that the compiler must allocate 3 GPRs in all multipass vertex shader modes).

19.3.2 Pixel shaders

In the case of pixel shaders, if GEN_INDEX_PIX is set, the data will be put in the x field of the param_gen_pos+1 register.

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

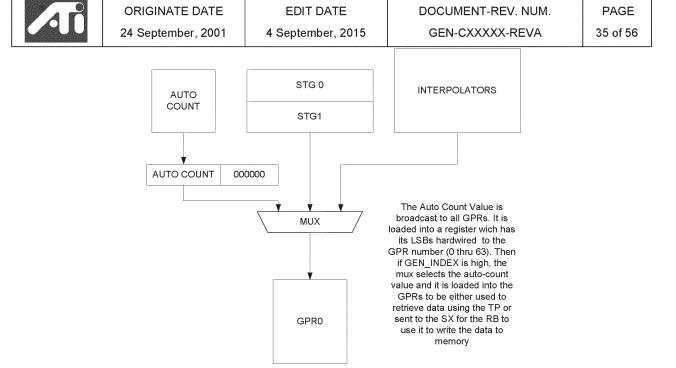


Figure 10: GPR input mux Control

20. State management

Every clock, the sequencer will report to the CP the oldest states still in the pipe. These are the states of the programs as they enter the last ALU clause.

20.1 Parameter cache synchronization

In order for the sequencer not to begin a group of pixels before the associated group of vertices has finished, the sequencer will keep a 6 bit count per state (for a total of 8 counters). These counters are initialized to 0 and every time a vertex shader exports its data TO THE PARAMETER CACHE, the corresponding pointer is incremented. When the SC sends a new vector of pixels with the SC_SQ_new_vector bit asserted, the sequencer will first check if the count is greater than 0 before accepting the transmission (it will in fact accept the transmission but then lower its ready to receive). Then the sequencer waits for the count to go to one and decrements it. The sequencer can then issue the group of pixels to the interpolators. Every time the state changes, the new state counter is initialized to 0.

21. XY Address imports

The SC will be able to send the XY addresses to the GPRs. It does so by interleaving the writes of the IJs (to the IJ buffer) with XY writes (to the XY buffer). Then when writing the data to the GPRs, the sequencer is going to interpolate the IJ data or pass the XY data thru a Fix \rightarrow float converter and expander and write the converted values to the GPRs. The Xys are currently SCREEN SPACE COORDINATES. The values in the XY buffers will wrap. See section 19.2 for details on how to control the interpolation in this mode.

21.1 Vertex indexes imports

In order to import vertex indexes, we have 16 8x96 staging registers. These are loaded one line at a time by the VGT block (96 bits). They are loaded in floating point format and can be transferred in 4 or 8 clocks to the GPRs.

22. Registers

Please see the auto-generated web pages for register definitions.

Exhibit 2038.doc 81670 Bytes*** [©] ATI Confidential. Reference Copyright Notice on Cover Page [©] ***

AMD1044_0257901

ATI Ex. 2109 IPR2023-00922 Page 191 of 326



23. Interfaces

23.1 External Interfaces

Whenever an x is used, it means that the bus is broadcast to all units of the same name. For example, if a bus is named SQ \rightarrow SPx it means that SQ is going to broadcast the same information to all SP instances.

23.2 SC to SP Interfaces

23.2.1 SC SP#

There is one of these interfaces at front of each of the SP (buffer to stage pixel interpolators). This interface transmits the I,J data for pixel interpolation. For the entire system, two quads per clock are transferred to the 4 SPs, so each of these 4 interfaces transmits one half of a quad per clock. The interface below describes a half of a quad worth of data

The actual data which is transferred per quad is

Ref Pix I => S4.20 Floating Point I value *4 Ref Pix J => S4.20 Floating Point J value *4

This equates to a total of 200 bits which transferred over 2 clocks and therefor needs an interface 100 bits wide

Additionally, X,Y data (12-bit unsigned fixed) is conditionally sent across this data bus over the same wires in an additional clock. The X,Y data is sent on the lower 24 bits of the data bus with faceness in the msb. Transfers across these interfaces are synchronized with the SC SQ IJ Control Bus transfers.

The data transfer across each of these busses is controlled by a IJ BUF INUSE COUNT in the SC. Each time the SC has sent a pixel vector's worth of data to the SPs, he will increment the IJ BUF INUSE COUNT count. Prior to sending the next pixel vectors data, he will check to make sure the count is less than MAX BUFER MINUS 2, if not the SC will stall until the SQ returns a pipelined pulse to decrement the count when he has scheduled a buffer free. Note: We could/may optimize for the case of only sending only IJ to use all the buffers to pre-load more. Currently it is planned for the SP to hold 2 double buffers of I,J data and two buffers of X,Y data, so if either X,Y or Centers and Centroids are on, then the SC can send two Buffers.

In at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will increment buffer write address pointers correctly all the time. (We may revisit this for both the SX,SP,SQ and add a EndOfVector signal on all interfaces to guit early. We opted for the simple mode first with a belief that only the end of packet and multiple new vector signals should cause a partial vector and that this would not really be significant performance hit.)

Name	Bits	Description
SC_SP#_data	100	IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit)
Type 0 or 1, First clock I, second clk J		Type 0 or 1, First clock I, second clk J
		Field ULC URC LLC LRC
		Bits [63:39] [38:26] [25:13] [12:0]
		Format SE4M20 SE4M20 SE4M20 SE4M20
		Type 2
		Field Face X Y
		Bits [24] [23:12] [11:0]
		Format Bit Unsigned Unsigned
SC_SP#_valid	1	Valid
SC_SP#_last_quad_data	1	This bit will be set on the last transfer of data per quad.
SC SP# type	2	0 -> Indicates centroids
		1 -> Indicates centers
		2 -> Indicates X,Y Data and faceness on data bus
		The SC shall look at state data to determine how many types to send for the

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

24 September, 2001	4 September, 2015	GEN-CXXXXX-REVA	37 of 56
ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE

The **#** is included for clarity in the spec and will be replaced with a prefix of u**#**_ in the verilog module statement for the SC and the SP block will have neither because the instantiation will insert the prefix.

23.2.2 SC_SQ

This is the control information sent to the sequencer in order to synchronize and control the interpolation and/or loading data into the GPRs needed to execute a shader program on the sent pixels. This data will be sent over two clocks per transfer with 1 to 16 transfers. Therefore the bus (approx 108 bits) could be folded in half to approx 54 bits.

Name	Bits	Description
SC_SQ_data	46	Control Data sent to the SQ
		1 clk transfers
		Event – valid data consist of event_id and
		state_id. Instruct SQ to post an
		event vector to send state id and
		event_id through request fifo
		and onto the reservation stations
		making sure state id and/or event_id
		gets back to the CP. Events only
		follow end of packets so no pixel
		vectors will be in progress.
		Empty Quad Mask – Transfer Control data
		consisting of pc_dealloc
		or new_vector. Receipt of this is to
		transfer pc dealloc or new vector
		without any valid quad data. New
		vector will always be posted to
		request fifo and pc_dealloc will be
		attached to any pixel vector
		outstanding or posted in request fifo
		if no valid quad outstanding.
		2 clk transfers
		Quad Data Valid – Sending quad data with or
		without new_vector or pc_dealloc.
		New vector will be posted to request
		fifo with or without a pixel vector and
		pc_dealloc will be posted with a pixel
		vector unless none is in progress. In
		this case the pc_dealloc will be
		posted in the request queue.
		Filler quads will be transferred with
		The Quad mask set but the pixel
		corresponding pixel mask set to
		zero.
SC SQ valid	1	SC sending valid data, 2 nd clk could be all zeroes

SC_SQ_data – first clock and second clock transfers are shown in the table below.

Name	BitField	Bits	Description
1 st Clock Transfer			
SC_SQ_event	0	1	This transfer is a 1 clock event vector Force quad_mask =
			new_vector=pc_dealloc=0
SC_SQ_event_id	[5:1]	4	This field identifies the event 0 => denotes an End Of State Event 1

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE [DATE	EI	DIT DATE	R400 Sequencer Specification	PAGE			
	24 September	2001	4 Sep	tember, 2015		38 of 56			
	*******			=> TBD	=> TBD				
SC_SQ_state	_id	[8:6]	3	State/constant	oointer (6*3+3)				
SC_SQ_pc_de	ealloc	[11:9]	3	Deallocation to	ken for the Parameter Cache				
SC_SQ_new_	vector	12	1		ait for Vertex shader done count > 0 a Pixel Vector the SQ will decrement th				
SC_SQ_quad	mask	[16:13]	4		sk left to right SP0 => SP3				
SC_SQ_end_		17	1	End Of the prim	*				
SC_SQ_pix_n		[33:18]	16		pixels SP0=>SP3 (UL,UR,LL,LR)				
SC SQ provo	ok vtx	[35:34]	2	Provoking verte	ex for flat shading				
SC_SQ_lod_c	orrect_0	[44:36]	9	LOD correction for quad 0 (SP0) (9 bits per quad)					
SC_SQ_lod_c	orrect_1	[53:45]	9	LOD correction	LOD correction for quad 1 (SP1) (9 bits per quad)				
2nd Clock Tra	nsfer								
SC_SQ_lod_c		[8:0]	9		for quad 2 (SP2) (9 bits per quad)				
SC_SQ_lod_c		[17:9]	9	LOD correction for quad 3 (SP3) (9 bits per quad)					
SC_SQ_pc_p1		[28:18]	11	Parameter Cache pointer for vertex 0					
SC_SQ_pc_pt	r1	[39:29]	11		he pointer for vertex 1				
SC_SQ_pc_pt	r2	[50:40]	11		he pointer for vertex 2				
SC_SQ_prim_	type	[53:51]	3		d Real time command need to load te	x cords from			
				alternate buffer					
				000: Sprite (poi	nt)				
				001: Line					
				010: Tri_rect					
				100: Realtime S					
				101: Realtime L					
				110: Realtime T	ri_rect				

Name	Bits	Description
SQ_SC_free_buff	1	Pipelined bit that instructs SC to decrement count of buffers in use.
SQ_SC_dec_cntr_cnt	1	Pipelined bit that instructs SC to decrement count of new vector and/or event sent to prevent SC from overflowing SQ interpolator/Reservation request fifo.

The scan converter will submit a partial vector whenever:

1.) He gets a primitive marked with an end of packet signal.

2.) A current pixel vector is being assembled with at least one or more valid quads and the vector has been marked for deallocate when a primitive marked new_vector arrives. The Scan Converter will submit a partial vector (up to 16quads with zero pixel mask to fill out the vector) prior to submitting the new_vector marker\primitive.

(This will prevent a hang which can be demonstrated when all primitives in a packet three vectors are culled except for a one quad primitive that gets marked pc_dealloc (vertices maximum size). In this case two new_vectors are submitted and processed, but then one valid quad with the pc_dealloc creates a vector and then the new would wait for another vertex vector to be processed, but the one being waited for could never export until the pc_dealloc signal made it through and thus the hang.)

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***



23.2.3 SQ to SX(SP): Interpolator bus

Name	Direction	Bits	Description
SQ_SPx_interp_flat_vtx	SQ→SPx	2	Provoking vertex for flat shading
SQ_SPx_interp_flat_gouraud	SQ→SPx	1	Flat or gouraud shading
SQ_SPx_interp_cyl_wrap	SQ→SPx	4	Wich channel needs to be cylindrical wrapped
SQ_SPx_interp_param_gen	SQ→SPx	1	Generate Parameter
SQ_SPx_interp_prim_type	SQ→SPx	2	Bits [1:0] of primitive type sent by SC
SQ_SPx_interp_buff_swap	SQ→SPx	1	Swapp IJ buffers
SQ_SPx_interp_IJ_line	SQ→SPx	2	IJ line number
SQ_SPx_interp_mode	SQ→SPx	1	Center/Centroid sampling
SQ_SXx_pc_ptr0	SQ→SXx	11	Parameter Cache Pointer
SQ_SXx_pc_ptr1	SQ→SXx	11	Parameter Cache Pointer
SQ_SXx_pc_ptr2	SQ→SXx	11	Parameter Cache Pointer
SQ_SXx_rt_sel	SQ→SXx	1	Selects between RT and Normal data (Bit 2 of prim type)
SQ_SX0_pc_wr_en	SQ→SX0	8	Write enable for the PC memories
SQ_SX1_pc_wr_en	SQ→SX1	8	Write enable for the PC memories
SQ_SXx_pc_wr_addr	SQ→SXx	7	Write address for the PCs
SQ_SXx_pc_channel_mask	SQ→SXx	4	Channel mask
SQ_SXx_pc_ptr_valid	SQ→SXx	1	Read pointers are valid.
SQ_SPx_interp_valid	SQ→SPx	1	Interpolation control valid
SQ_SPx_SIMD_engine	SQ→SPx	1	Tells which SIMD engine this data belongs to

23.2.4 SQ to SP: Staging Register Data

This is a broadcast bus that sends the VSISR information to the staging registers of the shader pipes.

Name	Direction	Bits	Description
SQ_SPx_vsr_data	SQ→SPx	96	Pointers of indexes or HOS surface information
SQ_SPx_vsr_wrt_addr	SQ→SPx	3	Staging register write address
SQ_SPx_vsr_rd_addr	SQ→SPx	3	Staging register read address
SQ_SP0_vsr_valid	SQ→SP0	1	Data is valid
SQ_SP1_vsr_valid	SQ→SP1	1	Data is valid
SQ_SP2_vsr_valid	SQ→SP2	1	Data is valid
SQ_SP3_vsr_valid	SQ→SP3	1	Data is valid
SQ_SPx_vsr_read	SQ→SPx	1	Increment the read pointers

23.2.5 VGT to SQ : Vertex interface

23.2.5.1 Interface Signal Table

The area difference between the two methods is not sufficient to warrant complicating the interface or the state requirements of the VSISRs. <u>Therefore, the POR for this interface is that the VGT will transmit the data to the</u> <u>VSISRs (via the Shader Sequencer) in full, 32-bit floating-point format.</u> The VGT can transmit up to six 32-bit floating-point values to each VSISR where four or more values require two transmission clocks. The data bus is 96 bits wide. In the case where an event is sent the 5 LSBs of VGT_SQ_vsisr_data contain the eventID.

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE DATE 24 September, 2001		EDIT DATE	R400 Sequencer Specification	PAGE			
			4 September, 2015		40 of 56			
Name		Bits	Description					
VGT_SQ_vsisi	_data	96	Pointers of indexes or HOS	surface information				
VGT_SQ_ever	nt	1	VGT is sending an event					
VGT_SQ_vsisi	_continued	1	0: Normal 96 bits per vert 1: double 192 bits per vert					
VGT_SQ_end	_of_vtx_vect	1	Indicates the last VSISR data set for the current process vector (for double vector data, "end_of_vector" is set on the first vector)					
VGT_SQ_indx	valid	1	Vsisr data is valid					
VGT_SQ_state	•	3	Render State (6*3+3 for constants). This signal is guaranteed to be correct when "VGT_SQ_vgt_end_of_vector" is high.					
VGT_SQ_send 1			Data on the VGT_SQ is valid receive (see write-up for standard R400 SEND/RTR interface handshaking)					
SQ_VGT_rtr		1	Ready to receive (see write-up for standard R400 SEND/RTR interface handshaking)					

23.2.5.2 Interface Diagrams

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257906

ATI Ex. 2109 IPR2023-00922 Page 196 of 326

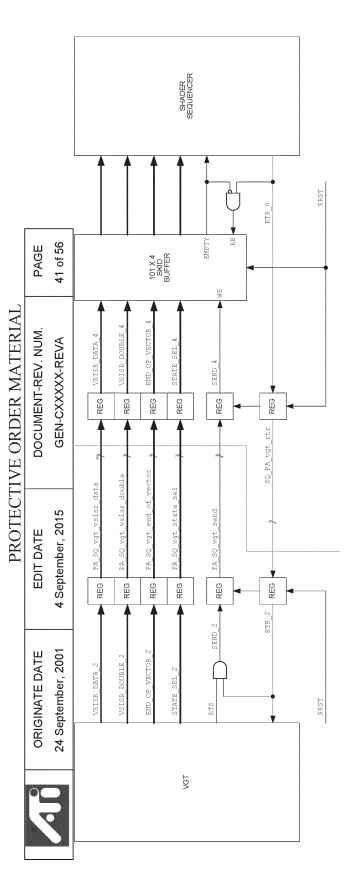


Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257907

ATI Ex. 2109 IPR2023-00922 Page 197 of 326

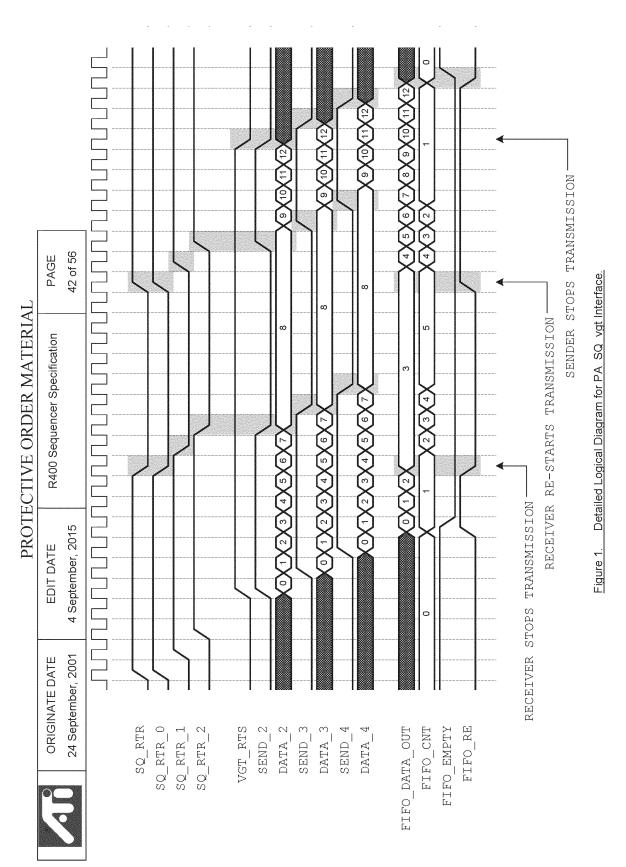


Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***



23.2.6~ SQ to SX: Control bus

Name	Direction	Bits	Description
SQ_SXx_exp_type	SQ→SXx	2	00: Pixel without z (1 to 4 buffers) 01: Pixel with z (1 to 4 buffers) 10: Position (1 or 2 results) 11: Pass thru (1 to 5 results aligned)
SQ_SXx_exp_number	SQ→SXx	2	Number of locations needed in the export buffer (encoding depends on the type see bellow).
SQ_SXx_exp_alu_id	SQ→SXx	4	ALU ID. Revolving ID 0 thru 15. Memory exports have to increment this count by 4 or 8 depending on the size requested. Other type of exports increment the ID by 1.
SQ SXx exp valid	SQ→SXx	1	Valid bit
SQ_SXx_exp_state	SQ→SXx	3	State Context
SQ_SXx_free_done	SQ→SXx	1	Pulse that indicates that the previous export is finished from the point of view of the SP. This does not necessarily mean that the data has been transferred to RB or PA, or that the space in export buffer for that particular vector thread has been freed up.
SQ_SXx_free_alu_id	SQ→SXx	4	ALU ID that was used at allocate time.

Depending on the type the number of export location changes:

- Type 00 : Pixels without Z
 - o 00 = 1 buffer
 - 01 = 2 buffers
 - \circ 10 = 3 buffers
 - 11 = 4 buffer
 - Type 01: Pixels with Z
 - \circ 00 = 2 Buffers (color + Z)
 - \circ 01 = 3 buffers (2 color + Z)
 - \circ 10 = 4 buffers (3 color + Z)
 - \circ 11 = 5 buffers (4 color + Z)
- Type 10 : Position export
 - \circ 00 = 1 position
 - \circ 01 = 2 positions
 - \circ 1X = Undefined
- Type 11: Pass Thru
 - \circ 00 = 4 buffers
 - \circ 01 = 8 buffers
 - \circ 10 = Undefined
 - 11 = Undefined

Below the thick black line is the end of transfer packet that tells the SX that a given export is finished. The report packet will always arrive either before or at the same time than the next export to the same ALU id.

Name	Direction	Bits	Description
SXx_SQ_pix_free_count0	SXx→SQ	6	How many slots where just freed in the SX for bank0
SXx_SQ_pix_count0_valid	SXx→SQ	1	Free_count0 is valid
SXx_SQ_pix_free_count1	SXx→SQ	6	How many slots where just freed in the SX for bank1
SXx_SQ_pix_count1_valid	SXx→SQ	1	Free_count1 is valid
SXx_SQ_pos_free_count0	SXx→SQ	4	How many slots where just freed in the SX for bank0
SXx_SQ_pos_count0_valid	SXx→SQ	1	Free_count0 is valid
SXx_SQ_pos_free_count1	SXx→SQ	4	How many slots where just freed in the SX for bank1

23.2.7 SX to SQ : Output file control

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

ORIGINATE DATE		DATE	EDIT DATE			R400 Sequencer Specification	PAGE
	24 September,	2001	001 4 Septemb				44 of 56
SXx_SQ_pos_	_count1_valid	SXx→S	SQ	1 F	Fre	e_count1 is valid	
SXx_SQ_mem	1_export_free	SXx_→S	SQ	1 F	Fre	ed a memory export slot	

23.2.8 SQ to TP: Control bus

Once every clock, the fetch unit sends to the sequencer on which RS line it is now working and if the data in the GPRs is ready or not. This way the sequencer can update the fetch valid bits flags for the reservation station. The sequencer also provides the instruction and constants for the fetch to execute and the address in the register file where to write the fetch return data.

Name	Direction	Bits	Description
TPx_SQ_data_rdy	$TPx \rightarrow SQ$	1	Data ready
TPx_SQ_rs_line_num	$TPx \rightarrow SQ$	6	Line number in the Reservation station
TPx_SQ_type	TPx→ SQ	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_TPx_send	SQ→TPx	1	Sending valid data
SQ_TPx_const	SQ→TPx	48	Fetch state sent over 4 clocks (192 bits total)
SQ_TPx_instr	SQ→TPx	24	Fetch instruction sent over 4 clocks
SQ_TPx_end_of_group	SQ→TPx	1	Last instruction of the group
SQ_TPx_Type	SQ→TPx	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_TPx_gpr_phase	SQ→TPx	2	Write phase signal
SQ_TP0_lod_correct	SQ→TP0	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP0_pix_mask	SQ→TP0	4	Pixel mask 1 bit per pixel
SQ_TP1_lod_correct	SQ→TP1	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP1_pix_mask	SQ→TP1	4	Pixel mask 1 bit per pixel
SQ TP2 lod correct	SQ→TP2	6	LOD correct 3 bits per comp 2 components per quad
SQ TP2 pix mask	SQ→TP2	4	Pixel mask 1 bit per pixel
SQ TP3 lod correct	SQ→TP3	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP3_pix_mask	SQ→TP3	4	Pixel mask 1 bit per pixel
SQ_TPx_rs_line_num	SQ→TPx	6	Line number in the Reservation station
SQ_TPx_write_gpr_index	SQ->TPx	7	Index into Register file for write of returned Fetch Data
SQ_TPx_ctx_id	SQ→TPx	3	The state context ID (needed for multisample resolves)
SQ_TPx_SIMD	SQ->TPx	1	Tells the TP from which SIMD the data is coming from.

23.2.9 SQ to VC: Control bus

Once every clock, the VC unit sends to the sequencer on which RS line it is now working and if the data in the GPRs is ready or not. This way the sequencer can update the fetch valid bits flags for the reservation station. The sequencer also provides the instruction and constants for the fetch to execute and the address in the register file where to write the fetch return data.

Name	Direction	Bits	Description
VCx_SQ_data_rdy	$VCx \rightarrow SQ$	1	Data ready
VCx_SQ_rs_line_num	$VCx \rightarrow SQ$	6	Line number in the Reservation station
VCx_SQ_type	VCx→ SQ	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_VCx_send	SQ→VCx	1	Sending valid data
SQ_VCx_const	SQ→VCx	48	Fetch state sent over 4 clocks (192 bits total)
SQ_VCx_instr	SQ→VCx	24	Fetch instruction sent over 4 clocks
SQ_VCx_end_of_group	SQ→VCx	1	Last instruction of the group
SQ_VCx_Type	SQ→VCx	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_VCx_gpr_phase	SQ→VCx	2	Write phase signal
SQ_VC0_pix_mask	SQ→VC0	4	Pixel mask 1 bit per pixel
SQ_VC1_pix_mask	SQ→VC1	4	Pixel mask 1 bit per pixel
SQ_VC2_pix_mask	SQ→VC2	4	Pixel mask 1 bit per pixel
SQ_VC3_pix_mask	SQ→VC3	4	Pixel mask 1 bit per pixel
SQ_VCx_rs_line_num	SQ→VCx	6	Line number in the Reservation station

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE DATE		EDI	Г DATE		DOCUMENT-REV. NUM.	PAGE
24 September, 2001			4 Septer	mber, 2	2015	GEN-CXXXXX-REVA	45 of 56
SQ_VCx_write_gpr_index SQ->VCx		x	7 Index into Register file for write of returned Feto			etch Data	
SQ_VCx_SIMD SQ->VC		x			Tells the VC from which SIMD the data is coming from.		

23.2.10 TP to SQ: Texture stall

The TP sends this signal to the SQ and the SPs when its input buffer is full. Stall needs to be aligned with the Instruction start.

Name	Direction	Bits	Description
TP_SQ_fetch_stall	TP→ SQ	1	Do not send more texture request if asserted

23.2.11 VC to SQ: Vertex Cache stall

The VCsends this signal to the SQ and the SPs when its input buffer is full. Stall needs to be aligned with the Instruction start.

Name	Direction	Bits	Description
VC_SQ_fetch_stall	$VC \rightarrow SQ$	1	Do not send more vertex cache request if asserted

23.2.12 SQ to SP: GPR and auto counter

Name	Direction	Bits	Description
SQ_SPx_simd0_gpr_wr_addr	SQ→SPx	7	Write address
SQ_SPx_simd0_gpr_rd_addr	SQ→SPx	7	Read address
SQ_SPx_simd0_gpr_rd_en	SQ→SPx	1	Read Enable
SQ_SP0_simd0_gpr_pspv_wr_en	SQ→SP0 (SP4)	4	Write Enable for the GPRs of SP0 for PS and PV
SQ_SP1_simd0_gpr_pspv_wr_en	SQ→SP1 (SP5)	4	Write Enable for the GPRs of SP1 for PS and PV
SQ_SP2_simd0_gpr_pspv_wr_en	SQ→SP2 (SP6)	4	Write Enable for the GPRs of SP2 for PS and PV
SQ_SP3_simd0_gpr_pspv_wr_en	SQ→SP3 (SP7)	4	Write Enable for the GPRs of SP3 for PS and PV
SQ_SP0_simd0_gpr_int_wr_en	SQ→SP0	1	Write Enable for the GPRs of SP0 for Inputs (interp/vtx)
SQ_SP1_simd0_gpr_int_wr_en	SQ→SP1	1	Write Enable for the GPRs of SP1 for Inputs (interp/vtx)
SQ_SP2_simd0_gpr_int_wr_en	SQ→SP2	1	Write Enable for the GPRs of SP2 for Inputs (interp/vtx)
SQ_SP3_simd0_gpr_int_wr_en	SQ→SP3	1	Write Enable for the GPRs of SP3 for Inputs (interp/vtx)
SQ_SPx_gpr_phase	SQ→SPx	2	The phase mux (arbitrates between inputs, ALU SRC reads and writes)
SQ SPx simd0 channel mask	SQ→SPx	4	The channel mask for SIMD0
SQ_SPx_gpr_input_sel	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_auto_count	SQ→SPx	21	Auto count generated by the SQ, common for all shader pipes
SQ_SPx_simd0_fetch_swizzle	SQ <u>→</u> SPx	6	Swizzle code for the TP request (2 bits per channel ignore W as it is not used). Bits [10] X mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W Bits [32] Y mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W Bits [54] Z mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W
SQ_SPx_simd0_fetch_resource	SQ <u>→</u> SPx	1	Resource in use currently 0: TP 1: VC
SQ_SPx_simd1_gpr_wr_addr	SQ→SPx	7	Write address
SQ_SPx_simd1_gpr_rd_addr	SQ→SPx	7	Read address
SQ_SPx_simd1_gpr_rd_en	SQ→SPx	1	Read Enable
SQ_SP0_simd1_gpr_pspv_wr_en	SQ→SP0 (SP4)	4	Write Enable for the GPRs of SP0 for PS and PV

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE DATE EDIT DATE			R400 Sequencer Specification	PAGE			
\mathbb{M}		24 September, 2001	4 September, 2	015		46 of 56		
	SQ_SP1_s	simd1_gpr_pspv_wr_en	SQ→SP1 (SP5)	4	Write Enable for the GPRs of SP1 for PS and PV			
[simd1_gpr_pspv_wr_en	SQ→SP2 (SP6)	4	Write Enable for the GPRs of SP2 for P	S and PV		
	SQ_SP3_s	simd1_gpr_pspv_wr_en	SQ→SP3 (SP7)	4	Write Enable for the GPRs of SP3 for P	S and PV		
	SQ_SPx_	simd1_channel_mask	SQ→SPx	4	The channel mask for SIMD1			
	SQ_SPx_s	simd1_fetch_resource	SQ→SPx	1	Resource in use currently 0: TP			
					1: VC			
	SQ_SPx_simd1_fetch_swizzle		simd1_fetch_swizzle SQ→SPx		Swizzle code for the TP request (2 bits ignore W as it is not used). Bits [10] X mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z Bits [32] Y mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z Bits [54] Z mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z	3=GPR_W 3=GPR_W		
	SQ_SP0_9	simd1_gpr_int_wr_en	SQ→SP0	1	Write Enable for the GPRs of SP (interp/vtx)	P0 for Inputs		
	SQ_SP1_simd1_gpr_int_wr_en SQ→SP1		SQ→SP1	1	Write Enable for the GPRs of SP (interp/vtx)	P1 for Inputs		
			SQ→SP2	1	Write Enable for the GPRs of SP (interp/vtx)	P2 for Inputs		
	SQ_SP3_s	simd1_gpr_int_wr_en	SQ→SP3	1	Write Enable for the GPRs of SP (interp/vtx)	23 for Inputs		

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257912

ATI Ex. 2109 IPR2023-00922 Page 202 of 326



23.2.13 SQ to SPx:

Name	Direction	Bits	Description
SQ_SPx_instr_start	SQ→SPx	1	Instruction start
SQ SPx simd0 instruct	SQ→SPx	24	Transferred over 4 cycles
			0: SRC A Negate Argument Modifier 0:0
			SRC A Abs Argument Modifier 1:1
			SRC A Swizzle 9:2
			Vector Dst 15:10
			Per channel Select 23:16
			00: GPR
			01: PV
			10: PS
			11: Constant (if 11 has to be 11 for al
			channels)
			-
			1: SRC B Negate Argument Modifier 0:0
			SRC B Abs Argument Modifier 1:1
			SRC B Swizzle 9:2
			Scalar Dst 15:10
			Per channel Select 23:16
			00: GPR
			01: PV
			10: PS
			11: Constant (if 11 has to be 11 for al
			channels)
			- 2: SRC C Negate Argument Modifier 0:0
			SRC C Abs Argument Modifier 1:1
			SRC C Swizzle 9:2
			Unused 15:10
			Per channel Select 23:16
			00: GPR
			01: PV
			10: PS
			11: Constant (if 11 has to be 11 for al
			channels)
			-
			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
			Unused 23:21
SQ_SP0_simd0_pred_override	SQ→SP0 (SP4)	4	0: Use per channel RGBA field (enables the per channe
			logic).
			1: Use GPR for PV or PS settings. LET the 11
			(constant) go thru unchanged
SQ_SP1_simd0_pred_override	SQ→SP1 (SP5)	4	0: Use per channel RGBA field (enables the per channe
			logic).
			1: Use GPR for PV or PS settings. LET the 11
		-	(constant) go thru unchanged
SQ_SP2_simd0_pred_override	SQ→SP2 (SP6)	4	0: Use per channel RGBA field (enables the per channel
			logic).
			1: Use GPR for PV or PS settings. LET the 11
			(constant) go thru unchanged
SQ_SP3_simd0_pred_override	SQ→SP3 (SP7)	4	0: Use per channel RGBA field (enables the per channel
og_or o_annuo_preu_overnue		17	
			logic).

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

11 1
8 I
TARK W

SQ

ñ	ORIGINATE DATE 24 September, 200′		EDIT DATE 4 September, 201			R400 Sequencer Specification	PAGE 48 of 56		
						Use GPR for PV or PS settings. onstant) go thru unchanged	LET	the	11
_SPx_s	simd0_stall	SC	⊋→SPx	1	St	tall signal			

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257914

ATI Ex. 2109 IPR2023-00922 Page 204 of 326

	ORIGINATE DATI		EDIT I	DATE		DOCUMENT-REV. NUM.	PAGE
	24 September, 200)1	4 Septem	ber, 201	5	GEN-CXXXXX-REVA	49 of 56
	24 September, 200 simd0_Waterfall simd1_instruct	SQ-	4 Septemi →SPx	2 2 24	Use t for th 0 : No 1: Wa 2: Wa 3: Wa 3: Wa Trans 0: SR SR SR Ve Pe	the incoming constant instead of the reg e next group of 16. formal mode aterfall on SRCA aterfall on SRCB aterfall on SRCC Sferred over 4 cycles CC A Negate Argument Modifier 0:0 CC A Abs Argument Modifier 1:1 CC A Swizzle 9:2 ctor Dst 15:10 er channel Select 23:16 00: GPR 01: PV 10: PS 11: Constant (if 11 has to b	istered one
					SR Sc Pe chanı 	C B Negate Argument Modifier 0:0 C B Abs Argument Modifier 1:1 C B Swizzle 9:2 alar Dst 15:10 er channel Select 23:16 00: GPR 01: PV 10: PS 11: Constant (if 11 has to b	
					SR SR Un Pe chanı 3: Ve Sc Ve	C C Abs Argument Modifier 1:1 C C Swizzle 9:2 used 15:10 er channel Select 23:16 00: GPR 01: PV 10: PS 11: Constant (if 11 has to be nels) ctor Opcode 4:0 10:5 alar Opcode 10:5 ctor Clamp 11:11	be 11 for all
<u>SO SP0 (</u>	simd1_pred_override	50	→SP0 (SP4)	4	Ve Sc Un	alar Clamp12:12ctor Write Mask16:13alar Write Mask20:17used23:21e per channel RGBA field (enables the	ner channel
					logic) 1: U (cons	se GPR for PV or PS settings. Li stant) go thru unchanged	ET the 11
	simd1_pred_override	SQ-	→SP1 (SP5)	4	logic) 1: U	e per channel RGBA field (enables the se GPR for PV or PS settings. Li stant) go thru unchanged	
SQ_SP2_s	simd1_pred_override	SQ-	→SP2 (SP6)	4	0: Us logic) 1: U	e per channel RGBA field (enables the	
SQ_SP3_s	simd1_pred_override	SQ-	→SP3 (SP7)	4	0: Us	e per channel RGBA field (enables the	per channel

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***



			logic). 1: Use GPR for PV or PS settings. LET the 11
			(constant) go thru unchanged
SQ_SPx_simd1_stall	SQ→SPx	1	Stall signal
SQ_SPx_simd1_Waterfall	SQ→SPx	2	Use the incoming constant instead of the registered one for the next group of 16. 0 : Normal mode 1: Waterfall on SRCA 2: Waterfall on SRCB 3: Waterfall on SRCC
SQ SPx export simd sel	SQ->SPx	1	Which SIMD engine is exporting.

23.2.14 SQ to SX: write mask interface (must be aligned with the SP data)

Name	Direction	Bits	Description
SQ_SX0_write_mask	SQ→SP0	8	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. This is for the data coming of SP0 and SP2.
SQ_SX1_ write_mask	SQ→SP1	8	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. This is for the data coming of SP1 and SP3.

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257916

ATI Ex. 2109 IPR2023-00922 Page 206 of 326



23.2.15 SP to SQ: Constant address load/ Predicate Set/Kill set

Name	Direction	Bits	Description
SP0_SQ_simd0_const_addr	(SP4) SP0→SQ	36	Constant address load 18 bits from SP0 and 18 from SP4.
SP0 SQ simd0 valid	SP0→SQ	1	Data valid
SP1_SQ_simd0_const_addr	(SP5) SP1→SQ	36	Constant address load
SP1 SQ simd0 valid	SP1→SQ	1	Data valid
SP2 SQ simd0 const addr	(SP6) SP2→SQ	36	Constant address load
SP2_SQ_simd0_valid	SP2→SQ	1	Data valid
SP3 SQ simd0 const addr	(SP7) SP3→SQ	36	Constant address load
SP3 SQ simd0 valid	SP3→SQ	1	Data valid
SP0_SQ_simd0_pred_kill_vector	(SP4) SP0→SQ	4	Data (predicates or kill/mask) 2 bits from SP0 and 2 bits from SP4
SP0_SQ_simd0_pred_kill_valid	SP0->SQ	1	Data valid
SP0_SQ_simd0_pred_kill_type	SP0->SQ	1	0: predicate vector 1: kill/mask vector
SP1_SQ_simd0_pred_kill_vector	(SP5) SP1→SQ	4	Data (predicates or kill/mask)
SP1_SQ_simd0_pred_kill_valid	SP1->SQ	1	Data valid
SP1_SQ_simd0_pred_kill_type	SP1->SQ	1	0: predicate vector 1: kill/mask vector
SP2 SQ simd0 pred kill vector	(SP6) SP2→SQ	4	Data (predicates or kill/mask)
SP2_SQ_simd0_pred_kill_valid	SP2->SQ	1	Data valid
SP2 SQ simd0 pred kill type	SP2->SQ	1	0: predicate vector
			1: kill/mask vector
SP3_SQ_simd0_pred_kill_vector	(SP7) SP3→SQ	4	Data (predicates or kill/mask)
SP3_SQ_simd0_pred_kill_valid	SP3->SQ	1	Data valid
SP3_SQ_simd0_pred_kill_type	SP3->SQ	1	0: predicate vector
			1: kill/mask vector
SP0_SQ_simd1_const_addr	(SP4) SP0→SQ	36	Constant address load 18 bits from SP0 and 18 from SP4.
SP0_SQ_simd1_valid	SP0→SQ	1	Data valid
SP1 SQ simd1 const addr	(SP5) SP1→SQ	36	Constant address load
SP1_SQ_simd1_valid	SP1→SQ	1	Data valid
SP2_SQ_simd1_const_addr	(SP6) SP2→SQ	36	Constant address load
SP2_SQ_simd1_valid	SP2→SQ	1	Data valid
SP3_SQ_simd1_const_addr	(SP7) SP3→SQ	36	Constant address load
SP3_SQ_simd1_valid	SP3→SQ	1	Data valid
SP0_SQ_simd1_pred_kill_vector	(SP4) SP0→SQ	4	Data (predicates or kill/mask) 2 bits from SP0 and 2
			bits from SP4
SP0_SQ_simd1_pred_kill_valid	SP0->SQ	1	Data valid
SP0_SQ_simd1_pred_kill_type	SP0->SQ	1	0: predicate vector
			1: kill/mask vector
SP1_SQ_simd1_pred_kill_vector	(SP5) SP1→SQ	4	Data (predicates or kill/mask)
SP1_SQ_simd1_pred_kill_valid	SP1->SQ	1	Data valid
SP1_SQ_simd1_pred_kill_type	SP1->SQ	1	0: predicate vector
			1: kill/mask vector
SP2_SQ_simd1_pred_kill_vector	(SP6) SP2→SQ	4	Data (predicates or kill/mask)
SP2_SQ_simd1_pred_kill_valid	SP2->SQ	1	Data valid
SP2_SQ_simd1_pred_kill_type	SP2->SQ	1	0: predicate vector 1: kill/mask vector
SP3_SQ_simd1_pred_kill_vector	(SP7) SP3→SQ	4	Data (predicates or kill/mask)
SP3_SQ_simd1_pred_kill_valid	SP3->SQ	1	Data valid
SP3_SQ_simd1_pred_kill_type	SP3->SQ	1	0: predicate vector
			1: kill/mask vector

Because of the sharing of the bus none of the MOVA, PREDSET or KILL instructions may be coissued.

23.2.16 SQ to SPx: constant broadcast

Name	Direction	Bits	Description

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE D	ATE	EDIT D	ATE		R400 Sequencer Specification	PAGE
AUU	24 September,	2001	1 4 Septembe		5		52 of 56
 SQ_SPx_s	imd0_const	SQ→S	Px 128 Co		Cor	nstant broadcast	
SQ_SPx_s	SQ_SPx_simd1_const SQ→SPx 128 Constant broadcast						

23.2.17 SQ to CP: RBBM bus

Name	Direction	Bits	Description
SQ_RBB_rs	SQ→CP	1	Read Strobe
SQ_RBB_rd	SQ→CP	32	Read Data
SQ_RBBM_nrtrtr	SQ→CP	1	Optional
SQ_RBBM_rtr	SQ→CP	1	Real-Time (Optional)

23.2.18 CP to SQ: RBBM bus

Name	Direction	Bits	Description
rbbm_we	CP→SQ	1	Write Enable
rbbm_a	CP→SQ	15	Address Upper Extent is TBD (16:2)
rbbm_wd	CP→SQ	32	Data
rbbm_be	CP→SQ	4	Byte Enables
rbbm_re	CP→SQ	1	Read Enable
rbb_rs0	CP→SQ	1	Read Return Strobe 0
rbb_rs1	CP→SQ	1	Read Return Strobe 1
rbb_rd0	CP→SQ	32	Read Data 0
rbb_rd1	CP→SQ	32	Read Data 0
RBBM_SQ_soft_reset	CP→SQ	1	Soft Reset

23.2.19 SQ to CP: State report

Name	Direction	Bits	Description
SQ_CP_vs_event	SQ→CP	1	Vertex Shader Event
SQ_CP_vs_eventid	SQ→CP	5	Vertex Shader Event ID
SQ_CP_ps_event	SQ→CP	1	Pixel Shader Event
SQ_CP_ps_eventid	SQ→CP	5	Pixel Shader Event ID

23.3 Example of control flow program execution

We now provide some examples of execution to better illustrate the new design.

Given the program:

Alu 0 Alu 1 Tex 0 Tex 1 Alu 3 Serial Alu 4 Tex 2 Alu 5 Alu 6 Serial Tex 3 Alu 7 Alloc Position 1 buffer Alu 8 Export Tex 4 Alloc Parameter 3 buffers Alu 9 Export 0 Tex 5 Alu 10 Serial Export 2 Alu 11 Export 1 End

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***



Would be converted into the following CF instructions:

Execute 0 Alu 0 Alu 0 Tex 0 Tex 1 Alu 0 Alu 0 Tex 0 Alu 1 Alu 0 Tex Execute 0 Alu Alloc Position 1 Execute 0 Alu 0 Tex Alloc Param 3 Execute_end 0 Alu 0 Tex 1 Alu 0 Alu

And the execution of this program would look like this:

Put thread in Vertex RS:

Control Flow Instruction Pointer (12 bits), (CFP) Execution Count Marker (3 or 4 bits), (ECM) Loop Iterators (4x9 bits), (LI) Call return pointers (4x12 bits), (CRP) Predicate Bits(4x64 bits), (PB) Export ID (1 bit), (EXID) GPR Base Ptr (8 bits), (GPR) Export Base Ptr (7 bits), (EB) Context Ptr (3 bits).(CPTR) LOD correction bits (16x6 bits) (LOD)

State Bits

otate bito									
New York Constraints of the Cons							an a		
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
0	0	0	0	0	0	0	0	0	0

Valid Thread (VALID) Texture/ALU engine needed (TYPE) Texture Reads are outstanding (PENDING) Waiting on Texture Read to Complete (SERIAL) Allocation Wait (2 bits) (ALLOC) 00 – No allocation needed 01 – Position export allocation needed (ordered export) 10 – Parameter or pixel export needed (ordered export) 11 – pass thru (out of order export) Allocation Size (4 bits) (SIZE) Position Allocated (POS_ALLOC) First thread of a new context (FIRST) Last (1 bit), (LAST)

Status Bits

		and the second	A CONTRACTOR OF					
VALID T	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
	ALU	0	0	0	0	0	1	0

Then the thread is picked up for the execution of the first control flow instruction: Execute 0 Alu 0 Alu 0 Tex 0 Tex 1 Alu 0 Alu 0 Tex 0 Alu 1 Alu 0 Tex

It executes the first two ALU instructions and goes back to the RS for a resource request change. Here is the state returned to the RS:

CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
0	2	0	0	0	0	0	0	0	0

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257919

PAGE

53 of 56

	ORIGINATE DATE 24 September, 2001		EDIT D 4 Septemb		R400) Sequencer Spec	cification	PAGE 54 of 56
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	0	0	0	0	0		0

Then when the texture pipe frees up, the arbiter picks up the thread to issue the texture reads. The thread comes back in this state:

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
0	4	0	0	0	0	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	1	0	0	0	1	0

Because of the serial bit the arbiter must wait for the texture to return and clear the PENDING bit before it can pick the thread up. Lets say that the texture reads are complete, then the arbiter picks up the thread and returns it in this state:

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
0	6	0	0	0	0	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	0	0	0	0	0	1	0

Again the TP frees up, the arbiter picks up the thread and executes. It returns in this state:

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
0	7	0	0	0	0	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	0	0	0	1	0

Now, even if the texture has not returned we can still pick up the thread for ALU execution because the serial bit is not set. The thread will however come back to the RS for the second ALU instruction because it has the serial bit set.

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
0	8	0	0	0	0	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	1	0	0	0	1	0

As soon as the TP clears the pending bit the thread is picked up and returns:

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

100	0

PAGE 55 of 56

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
0	9	0	0	0	0	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	0	0	0	0	0	1	0

Picked up by the TP and returns:

Execute 0 Alu

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
1	0	0	0	0	0	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	0	0	0	1	0

Picked up by the ALU and returns (lets say the TP has not returned yet): Alloc Position 1

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
2	0	0	0	0	0	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	01	1	0	1	0

If the SX has the place for the export, the SQ is going to allocate and pick up the thread for execution. It returns to the RS in this state:

Execute 0 Alu 0 Tex

State Bi	ts								
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
3	1	0	0	0	0	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	1	0	0	0	1	1	0

Now, since the TP has not returned yet, we must wait for it to return because we cannot issue multiple texture requests. The TP returns, clears the PENDING bit and we proceed:

Alloc Param 3

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE DATE 24 September, 2001			EDIT DATE 4 September, 2015		R400 Sequencer Specification			ation	-	PAGE 6 of 56	
Stat	te Bits									2750302537577575555555555555555		
CFF	> E	ECM	LI	CRP	PB	EXID		GPR	EB	CPTR	L	.OD
4	0		0	0	0	1		0	0	0	0)

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	10	3	1	1	0

Once again the SQ makes sure the SX has enough room in the Parameter cache before it can pick up this thread.

Execute end 0 Alu 0 Tex 1 Alu 0 Alu

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
5	1	0	0	0	1	0	100	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	1	0	0	0	1	1	0

This executes on the TP and then returns:

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
5	2	0	0	0	1	0	100	0	0

Status Bit	ts							
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	1	0	0	1	1	1

Waits for the TP to return because of the textures reads are pending (and SERIAL in this case). Then executes and does not return to the RS because the LAST bit is set. This is the end of this thread and before dropping it on the floor, the SQ notifies the SX of export completion.

24. Open issues

Need to do some testing on the size of the register file as well as on the register file allocation method (dynamic VS static).

Saving power?

Exhibit 2038.doc 81670 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257922

ATI Ex. 2109 IPR2023-00922 Page 212 of 326

Saue To:		ORIGINATE DATE	EDIT DA	TE	DOCUMENT-REV. NUM.	PAGE	
Saue To:		9 July, 2003	4 September	er, 2015 GEN-CXXXXX-REVA		1 of 54	
ACCOMPANIENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATT TECHNOLOGIES INC. HINDUGH UNAUTHORIZED USE OR DISCLOSURE. Copyright 2001. ATT Technologies Inc. All rights reserved. The material in this document constitutes an unpublished work. The copyright notice is intended to provide notice that ATT owns or information and tade secrets of ATT. No part of this document may be used, reproduced, or an information of ATT Constituted in any form or by any means without the prior written permission of ATT No part of this document may be used, reproduced or an information of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in the prior written permission of ATT	Author:	Laurent Lefebvre	A	I		-	
ACCOMPANIENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATT TECHNOLOGIES INC. HINDUGH UNAUTHORIZED USE OR DISCLOSURE. Copyright 2001. ATT Technologies Inc. All rights reserved. The material in this document constitutes an unpublished work. The copyright notice is intended to provide notice that ATT owns or information and tade secrets of ATT. No part of this document may be used, reproduced, or an information of ATT Constituted in any form or by any means without the prior written permission of ATT No part of this document may be used, reproduced or an information of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in any form or by any means without the prior written permission of ATT Constituted in the prior written permission of ATT	ssue To:			Copy No:			
SQ Periods and expected uses of the R400 Sequencer block (SEQ). It provides an overview of the block, it also describes the block interfaces, internal state diagrams. AUTOMATICALLY UPDATED FIELDS: Mame/Dept Current Intranet Search Titte: R00 Sequencer Specification Name/Dept Signature/Date Name/Dept Signature/Date Chrometers: <							
SQ Periods and expected uses of the R400 Sequencer block (SEQ). It provides an overview of the block, it also describes the block interfaces, internal state diagrams. AUTOMATICALLY UPDATED FIELDS: Mame/Dept Current Intranet Search Titte: R00 Sequencer Specification Name/Dept Signature/Date Name/Dept Signature/Date Chrometers: <							
AUTOMATICALLY UPDATED FIELDS: Downerstown: Cherofocely4000400_libidesignblocksisqh2400_Sequencer.doc Current Incardio: Cherofocely4000400_libidesignblocksisqh2400_Sequencer.doc Current Intranet Search Title Signature/Date Mame/Dept Signature/Date Mame/Dept Signature/Date Remarks: This DOCUMENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATT ECHNOLOGOISDS INC. THROUGH UNAUTHORIZED USE OR DISCLOSUREL. Copyright 2001, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished work for eco of this copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document constitutes an unpublished work for eco of this copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or antidetial, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or antidetial, proprietary information and trade secrets of ATI. No part of this docure that ATI ownes a copyright notice is not an admission that pu		R400 S	equence	er Spe	ecification		
AUTOMATICALLY UPDATED FIELDS: Downerstown: Cherofocely4000400_libidesignblocksisqh2400_Sequencer.doc Current Incardio: Cherofocely4000400_libidesignblocksisqh2400_Sequencer.doc Current Intranet Search Title Signature/Date Mame/Dept Signature/Date Mame/Dept Signature/Date Remarks: This DOCUMENT CONTAINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF ATT ECHNOLOGOISDS INC. THROUGH UNAUTHORIZED USE OR DISCLOSUREL. Copyright 2001, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished work for eco of this copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document constitutes an unpublished work for eco of this copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or antidetial, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or antidetial, proprietary information and trade secrets of ATI. No part of this docure that ATI ownes a copyright notice is not an admission that pu			0	~			
Derview: This is an architectural specification for the R400 Sequencer block (SEQ). It provides an overview of the required 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: C:perforcelv400/doc_lib/design/blocks/sqlR400_Sequencer.doc Current Intranet Search Title: R400 Sequences specification Mame/Dept Signature/Date Remarks: Signature/Date 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 internded to provide notice that ATI owns a copyright in this multished work. The copyright notice is internded to provide notice that ATI owns a copyright in this multished work. The copyright notice is internded to provide notice that ATI owns a copyright in this multished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and radie secrets of ATI. No part of this document may be excl. peroduced, or ansmitted in any form or by any means without the prior written permission of ATI Technologies Inc."			3	Q			
Derview: This is an architectural specification for the R400 Sequencer block (SEQ). It provides an overview of the required 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: C:perforcelv400/doc_lib/design/blocks/sqlR400_Sequencer.doc Current Intranet Search Title: R400 Sequences specification Mame/Dept Signature/Date Remarks: Signature/Date 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 internded to provide notice that ATI owns a copyright in this multished work. The copyright notice is internded to provide notice that ATI owns a copyright in this multished work. The copyright notice is internded to provide notice that ATI owns a copyright in this multished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and radie secrets of ATI. No part of this document may be excl. peroduced, or ansmitted in any form or by any means without the prior written permission of ATI Technologies Inc."			Versio	on 2.11			
required 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: C/perforce/v400/doc_lib/design/blocks/sq/R400_Sequencer.doc Current Intranet Search Title: R400 Sequencer Specification APPROVALS Name/Dept Signature/Date Remarks: Signature/Date Copyright 2001, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished work. The copyright notice is intended to provide notice that ATI owns a copyright in this inpublished work. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this inpublished work. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this inpublished work. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this inpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or rammitted in any form or by any means without the prior written permission of ATI Technologies Inc.*							
AUTOMATICALLY UPDATED FIELDS: Document Location: C:Qerforce\r400\doc_lib\design\blocks\sq\R400_Sequencer.doc Current Intranet Search Title: R400 Sequencer Specification APPROVALS Name/Dept Signature/Date Name/Dept Signature/Date Name/Dept Signature/Date Name/Section Signature/Date Name/Dept Signature/Date Copyright 2001, ATI 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 mpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ramsmitted in any form or by any means without the prior written permission of ATI Technologies Inc.*							
Document Location: Current Intranet Search Title: C:\perforce\r400\doc_lib\design\blocks\sq\R400_Sequencer.doc R400 Sequencer Specification APPROVALS Name/Dept Signature/Date Name/Dept Signature/Date Remarks: Signature/Date 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 inpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc."					,		
Document Location: Current Intranet Search Title: C:\perforce\r400\doc_lib\design\blocks\sq\R400_Sequencer.doc R400 Sequencer Specification APPROVALS Name/Dept Signature/Date Name/Dept Signature/Date Remarks: Signature/Date 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 inpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc."							
Document Location: Current Intranet Search Title: C:\perforce\r400\doc_lib\design\blocks\sq\R400_Sequencer.doc R400 Sequencer Specification APPROVALS Name/Dept Signature/Date Name/Dept Signature/Date Remarks: Signature/Date 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 inpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc."							
Document Location: Current Intranet Search Title: C:\perforce\r400\doc_lib\design\blocks\sq\R400_Sequencer.doc R400 Sequencer Specification APPROVALS Name/Dept Signature/Date Name/Dept Signature/Date Remarks: Signature/Date 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 inpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc."							
Current Intranet Search Title: R400 Sequencer Specification APPROVALS Name/Dept Signature/Date Name/Dept Signature/Date Remarks: 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 vork created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc."							
APPROVALS Name/Dept Signature/Date Remarks: Signature/Date 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 vork created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this nonfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc."		8			locks\sq\R400_Sequencer.doc		
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 vork created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this inpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc." EXTENDE 2004CO 84020 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © *** ATI 20 LG v. A							
Copyright 2001, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished vork created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this inpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc."		Name/Dept			Signature/Date		
Copyright 2001, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished vork created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this inpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc."							
Copyright 2001, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished vork created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this inpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc."							
Copyright 2001, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished vork created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this inpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc."							
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 inpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc." EXHIBIT 2039.doc 2012 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © *** ATI 20 LG v. A	Remarks:						
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 inpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc." EXHIBIT 2039.doc 2012 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © *** ATI 20 LG v. A							
Copyright 2001, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished vork created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this inpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc." ATI 20 LG v. A	THIS DO	CUMENT CONTAIN	S CONFIDE	INTIAL I	NFORMATION THAT CO	JLD BE	
Copyright 2001, ATI Technologies Inc. All rights reserved. The material in this document constitutes an unpublished vork created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this inpublished work. The copyright notice is not an admission that publication has occurred. This work contains onfidential, proprietary information and trade secrets of ATI. No part of this document may be used, reproduced, or ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc." Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © *** ATI 20 LG v. A						1	
work created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this impublished 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 ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc." Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © *** ATI 20. LG v. A		INC. THROUGH L	JNAUTHOR	IZED US	SE OR DISCLOSURE.		
work created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this impublished 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 ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc." Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © *** ATI 20. LG v. A							
work created in 2001. The use of this copyright notice is intended to provide notice that ATI owns a copyright in this impublished 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 ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc." Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © *** ATI 20. LG v. A							
Inpublished 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 ransmitted in any form or by any means without the prior written permission of ATI Technologies Inc." Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © *** ATI 20. LG v. A							
Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © *** ATI 20. LG v. A							
Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © *** ATI 20 LG v. A	confidential, pr	roprietary information and tra	ade secrets of A	TI. No part	of this document may be used, re		
LG v. A	transmitted in a	any form or by any means wit	nout the prior wr	iπen permis	SION OF ATT TECHNOLOGIES INC."		
LG v. A							
LG v. A	Exhi	bit 2039.doc 84302 Bytes*** ⓒ ATI (Confidential. Re	ference Co	pyright Notice on Cover Page © **	•• ATI 203	
						LG v. A IPR201:	

AMD1044_0257923

ATI Ex. 2109 IPR2023-00922 Page 213 of 326



Table Of Contents

1.	OVERVIEW	6
1.1	Top Level Block Diagram	8
1.2	Data Flow graph (SP)	9
1.3	Control Graph	
2.	INTERPOLATED DATA BUS	10
3.	INSTRUCTION STORE	
4.	SEQUENCER INSTRUCTIONS	13
5.	CONSTANT STORES	13
5.1	Memory organizations	13
5.2	Management of the Control Flow Constants	14
5.3	Management of the re-mapping tables	14
5.3.	1 R400 Constant management	14
5.3.	2 Dirty bits	14
5.3.	3 Free List Block	14
5.3.	4 De-allocate Block	15
5.3.	5 Operation of Incremental model	15
5.4	Constant Store Indexing	16
5.5	Real Time Commands.	16
5.6	Constant Waterfalling	
6.	LOOPING AND BRANCHES	17
6.1	The controlling state	
6.2	The Control Flow Program	17
6.2.	1 Control flow instructions table	18
6.3	Implementation	21
6.4	Data dependant predicate instructions	23
6.5	HW Detection of PV,PS	24
6.6	Register file indexing	
6.7	Debugging the Shaders	24
6.7.		24
6.7.	2 Method 2: Exporting the values in the GPRs	25
7.	PIXEL KILL MASK	
8.	REGISTER FILE ALLOCATION	
9.	FETCH ARBITRATION	
10.	VC ARBITRATION	
11.	ALU ARBITRATION	
12.	HANDLING STALLS	
12.1	SP stall conditions	
12.		
	1.2 PV Stalls	
13.	CONTENT OF THE RESERVATION STATION FIFOS	
14.	THE OUTPUT FILE	
15.	IJ FORMAT	
15.1	Interpolation of constant attributes	
16.	STAGING REGISTERS	
17.	THE PARAMETER CACHE	29

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	2	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
2	U	9 July, 2003	4 September, 2015	GEN-CXXXXX-REVA	3 of 54
17.1					
17.1.1					
17.1.2					
17.1.3					
17.2					

18.1		-			
18.2					
19.1					
19.2					
19.3 19.3.1		0			
19.3.1					
20. 3				•••••••••••••••••••••••••••••••••••••••	
		2			
21.1				***************************************	
23.1					
23.2					
23.2.1					
23.2.2		TANK TO A TANK TANK TANK TANK TANK TANK TANK TA			
23.2.3					
23.2.4		. , .			
23.2.5					
23.2.6	6 3	SQ to SX: Control bus			
23.2.7	7 5	SX to SQ : Output file c	ontrol		
23.2.8	8 3	SQ to TP: Control bus			
23.2.9	9 3	SQ to VC: Control bus			
23.2.1					
23.2.1	11 \	VC to SQ: Vertex Cache	e stall		
23.2.1					
23.2.1					
23.2.1				d with the SP data)	
23.2.1				et/Kill set	
23.2.1					
23.2.1					
23.2.1					
23.2.1		SQ to CP: State report .			
23.3					
24. C	JPEN	ISSUES		****	

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***



Revision Changes:

Rev 0.1 (Laurent Lefebvre) Date: May 7, 2001

Rev 0.2 (Laurent Lefebvre) Date : July 9, 2001 Rev 0.3 (Laurent Lefebvre) Date : August 6, 2001 Rev 0.4 (Laurent Lefebvre) Date : August 24, 2001

Rev 0.5 (Laurent Lefebvre) Date : September 7, 2001 Rev 0.6 (Laurent Lefebvre) Date : September 24, 2001 Rev 0.7 (Laurent Lefebvre) Date : October 5, 2001

Rev 0.8 (Laurent Lefebvre) Date : October 8, 2001 Rev 0.9 (Laurent Lefebvre) Date : October 17, 2001

Rev 1.0 (Laurent Lefebvre) Date : October 19, 2001 Rev 1.1 (Laurent Lefebvre) Date : October 26, 2001

Rev 1.2 (Laurent Lefebvre) Date : November 16, 2001 Rev 1.3 (Laurent Lefebvre) Date : November 26, 2001 Rev 1.4 (Laurent Lefebvre) Date : December 6, 2001

Rev 1.5 (Laurent Lefebvre) Date : December 11, 2001

Rev 1.6 (Laurent Lefebvre) Date : January 7, 2002

Rev 1.7 (Laurent Lefebvre) Date : February 4, 2002 Rev 1.8 (Laurent Lefebvre) Date : March 4, 2002

Rev 1.9 (Laurent Lefebvre) Date : March 18, 2002 Rev 1.10 (Laurent Lefebvre) Date : March 25, 2002 Rev 1.11 (Laurent Lefebvre) Date : April 19, 2002 Rev 2.0 (Laurent Lefebvre) Date : April 19, 2002 First draft.

Changed the interfaces to reflect the changes in the SP. Added some details in the arbitration section. Reviewed the Sequencer spec after the meeting on August 3, 2001.

Added the dynamic allocation method for register file and an example (written in part by Vic) of the flow of pixels/vertices in the sequencer. Added timing diagrams (Vic)

Changed the spec to reflect the new R400 architecture. Added interfaces.

Added constant store management, instruction store management, control flow management and data dependant predication.

Changed the control flow method to be more flexible. Also updated the external interfaces.

Incorporated changes made in the 10/18/01 control flow meeting. Added a NOP instruction, removed the conditional_execute_or_jump. Added debug registers.

Refined interfaces to RB. Added state registers.

Added SEQ \rightarrow SP0 interfaces. Changed delta precision. Changed VGT \rightarrow SP0 interface. Debug Methods added.

Interfaces greatly refined. Cleaned up the spec.

Added the different interpolation modes.

Added the auto incrementing counters. Changed the VGT \rightarrow SQ interface. Added content on constant management. Updated GPRs.

Removed from the spec all interfaces that weren't directly tied to the SQ. Added explanations on constant management. Added $PA \rightarrow SQ$ synchronization fields and explanation.

Added more details on the staging register. Added detail about the parameter caches. Changed the call instruction to a Conditionnal_call instruction. Added details on constant management and updated the diagram.

Added Real Time parameter control in the SX interface. Updated the control flow section.

New interfaces to the SX block. Added the end of clause modifier, removed the end of clause instructions.

Rearangement of the CF instruction bits in order to ensure byte alignement.

Updated the interfaces and added a section on exporting rules.

Added CP state report interface. Last version of the spec with the old control flow scheme New control flow scheme

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE			
C 900	9 July, 2003	4 September, 2015	GEN-CXXXXX-REVA	5 of 54			
Rev 2.01 (Lau			d slightly the control flow instructi	ons to			
Date : May 2, 2			ce jumps and calls.				
Rev 2.02 (Lau			the Opcodes. Added type field				
Date : May 13	, 2002		t/pred interface. Added Last field	to the			
			Pinstruction load interface.				
Rev 2.03 (Lau	,		erface updated to include prec				
Date : July 15,	2002		itions. Added the predicate no	stall			
Day 0.04 (Lau		instructi	,	a la a ma a			
Rev 2.04 (Lau			ented the new parameter generation s	cneme			
Date :August 2			oordinates points and lines STs.	o oturol			
Rev 2.05 (Lau Date : Septem		Some interface changes and an architectural change to the auto-counter scheme.					
Rev 2.06 (Lau		Widened the event interface to 5 bits. Some other					
Date : October		little typos corrected.					
Rev 2.07 (Lau		Loops, jumps and calls are now using a 13 bit					
Date : October		address which allows to jump and call and loop					
	,	around any control flow addresses (does not					
		requires to be even anymore).					
Rev 2.08 (Lau	rent Lefebvre)		tion updates after discussion with Cla	у.			
Date : October	r 16, 2002		-	-			
Rev 2.09 (Lau		Correcte	Corrected the SQ→SP staging register interface.				
Date : January							
Rev 2.10 (Lau		Adding I	Adding R500 modifications				
Date : April 8,							
Rev 2.11 (Lau		Adding	Adding SQ->SP updated interfaces				
Date : May 1, :	2003						

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257927

ATI Ex. 2109 IPR2023-00922 Page 217 of 326



1. Overview

The sequencer chooses four ALU threads (two from each bank), a vertex cache and a fetch thread to execute, and executes all of the instructions in a block before looking for a new clause of the same type. Two ALU threads are executed interleaved to hide the ALU latency. The arbitrator will give priority to older threads. There are two separate reservation stations, one for pixel vectors and one for vertices vectors. This way a pixel can pass a vertex and a vertex can pass a pixel.

There are also 2 separate ALU banks from which the SQ picks the ALU threads to be executed in parallel.

To support the shader pipe the sequencer also contains the shader instruction store, constant store, control flow constants and texture state. The height shader pipes also execute the same two instructions thus there is only one sequencer for the whole chip but it issues 2 instructions every four clocks.

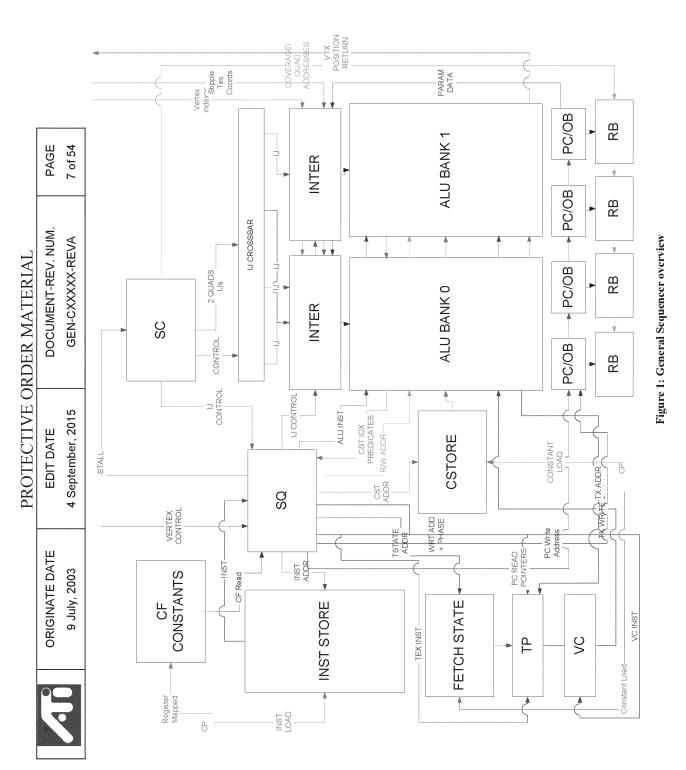
The sequencer first arbitrates between vectors of 64 vertices that arrive directly from primitive assembly and vectors of 16 quads (64 pixels) that are generated in the scan converter.

The vertex or pixel program specifies how many GPRs it needs to execute. The sequencer will not start the next vector until the needed space is available in the GPRs.

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044 0257928

ATI Ex. 2109 IPR2023-00922 Page 218 of 326





AMD1044_0257929

ATI Ex. 2109 IPR2023-00922 Page 219 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
9 July, 2003	4 September, 2015		8 of 54

1.1 Top Level Block Diagram

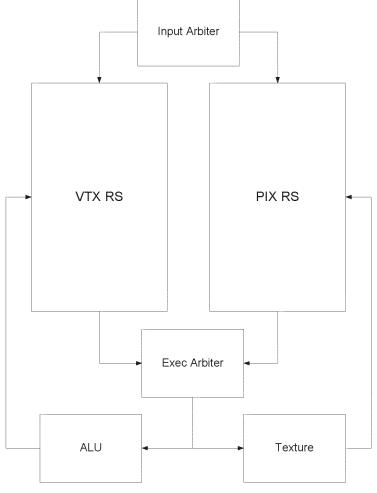


Figure 2: Reservation stations and arbiters

Under this new scheme, the sequencer (SQ) will only use one global state management machine per vector type (pixel, vertex) that we call the reservation station (RS).

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257930

ATI Ex. 2109 IPR2023-00922 Page 220 of 326

	PAGE	9 of 54
ER MATERIAL	DOCUMENT-REV. NUM.	GEN-CXXXX-REVA
PROTECTIVE ORDER MATERL	EDIT DATE	4 September, 2015
Р	ORIGINATE DATE	9 July, 2003

Γ

1.2 Data Flow graph (SP)



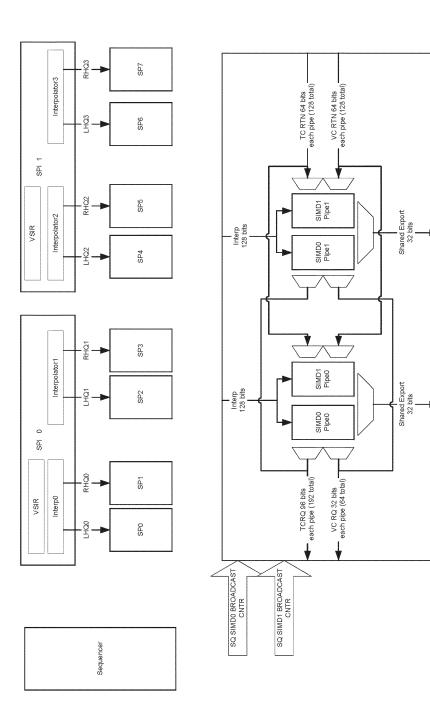


Exhibit 2039 doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

Figure 3: The shader Pipe

AMD1044_0257931

ATI Ex. 2109 IPR2023-00922 Page 221 of 326

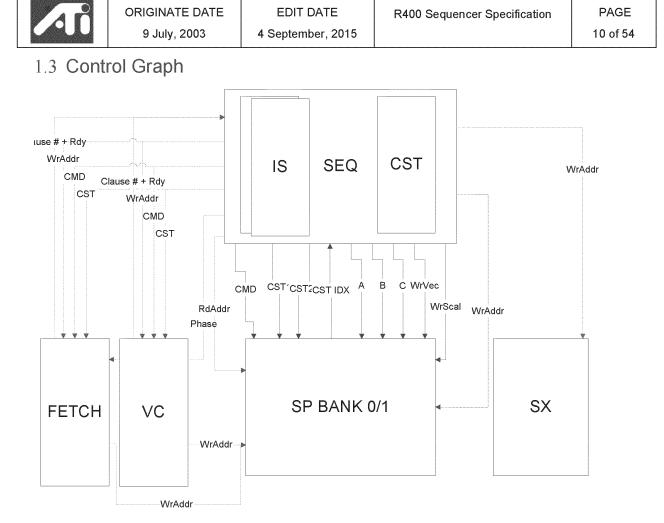


Figure 4: Sequencer Control interfaces

In green is represented the Fetch control interface, in red the ALU control interface, in blue the Interpolated/Vector control interface and in purple is the output file control interface.

2. Interpolated data bus

The interpolators contain an IJ buffer to pack the information as much as possible before writing it to the register file.

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257932

ATI Ex. 2109 IPR2023-00922 Page 222 of 326

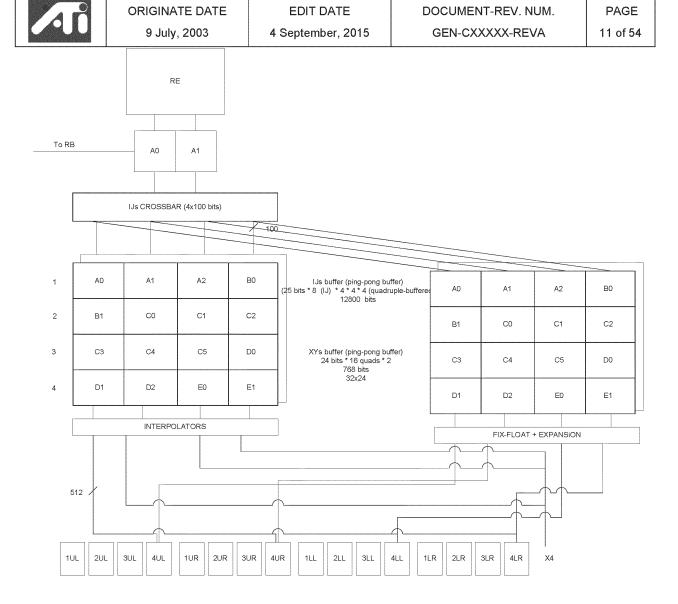


Figure 5: Interpolation buffers

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257933

ATI Ex. 2109 IPR2023-00922 Page 223 of 326

		Γ23					51 -4 <	52- 55	29 59 <	> ၀ အီ	
		T22					<pre>< < < 32- 48- 35 51</pre>	V V 36-52- 39 55	40-56- 43 59	<pre>< < < 44-60- 47 63</pre>	×
		T17 T18 T19 T20 T21 T22 T23					> ¹ ₀ ¹ −	23 ² 0 <	24- 27	31 -28 <	5
		Т20					> °0	> -4	, ⇒ ⇒ 4	12 ¹ 2 <	
		Т19							ОШ	Ш	
		T18								õ	
	4				Хü	Х Ч		8	5	6	
PAGE	12 of 54	T16			ЮШ	ш				BO	N
		T15			ЦО	Щ,	5	D2			Δ
tion		T14	ХБ	Z2		Хa	ß	0 4	C5		
ecifica		T13	Б	D2		8	B1				
R400 Sequencer Specification		T12	Б	D2		8	AO	A1	A2		
eduen		T11		7 2 2	C5 C5		<u> </u>		0 Ш	Ш	
400 S		T10		C4	C5	L C				8	
		Т9		C4	C5			8	5	8	
	015	Т8	≿ຬ	XS	×ַט	C3 X				BO	diaman
EDIT DATE	nber, 2	T7	ប	8	δ	ß	5	D2			à
EDIT	4 September, 2015	T6	ß	00	ъ	C2	Ü	C4	C5		
	4	Т5	×₽			X B X	B B				
ATE		T4	B			BO	AO	A1	A2		
TE D/	9 July, 2003	Т3	ñ			BO	51 4 8- 51	X≺ 52- 55	X ≺ 56- 59	S ° ≺	
ORIGINATE DATE	9 July	Т2	Υd	¥ X	₹ A2		XY 32- 35	39 3 0 ×√	40- 43- 43	ХҮ 44- 47	\succ
OR		ŗ	AO	A1	A2		16- 19	23 ² 0 X	ХҮ 24- 27	31 -28- 33 -28-	×
PC		TO	AO	A1	A2		,×0 2×	¥ 7-4	, ∾ ×	X 12- 15	
	2		o Sp	− SP	S P S	ი ი	O S	- S	N SP	ი ე	

PROTECTIVE ORDER MATERIAL

Exhibit 2039 doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

Figure 6: Interpolation timing diagram

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
9 July, 2003	4 September, 2015	GEN-CXXXXX-REVA	13 of 54

Above is an example of a tile the sequencer might receive from the SC. The write side is how the data get stacked into the XY and IJ buffers, the read side is how the data is passed to the GPRs. The IJ information is packed in the IJ buffer 4 quads at a time or two clocks. The sequencer allows at any given time as many as four quads to interpolate a parameter. They all have to come from the same primitive. Then the sequencer controls the write mask to the GPRs to write the valid data in.

3. Instruction Store

There is going to be two instruction stores for the whole chip. They will each contain 4096 instructions of 96 bits each.

They will be 1 port memories; Ports are allocated in this fashion (but not necessarily in this order):

ALU 0 SIMD0 CF	ALU 0 SIMD1 CF
ALU 0 SIMD0	ALU 0 SIMD1
ALU 1 SIMD0 CF	ALU 1 SIMD1 CF
ALU 1 SIMD0	ALU 1 SIMD1
Fetch CF	Fetch CF
Fetch	Fetch
VC CF	VC CF
VC	VC

Fetch and VC can steal one another's ports with stated resource having priority over its port (this is not really necessary for the R500 but will be for any derivative part because there will only be one instruction store).

Writes are opportunistic.

The instruction store is loaded by the CP thru the register mapped registers.

The VS_BASE and PS_BASE context registers are used to specify for each context where its shader is in the instruction memory.

For the Real time commands the story is quite the same but for some small differences. There are no wrap-around points for real time so the driver must be careful not to overwrite regular shader data. The shared code (shared subroutines) uses the same path as real time.

4. Sequencer Instructions

All control flow instructions instructions are handled by the sequencer only. The ALUs will perform NOPs during this time (MOV PV,PV, PS,PS) if they have nothing else to do.

5. Constant Stores

5.1 Memory organizations

A likely size for the ALU constant store is 1024x128 bits. The read BW from the ALU constant store is 128 bits/clock and the write bandwidth is 32 bits/clock (directed by the CP bus size not by memory ports).

The maximum logical size of the constant store for a given shader is 256 constants. Or 512 for the pixel/vertex shader pair. The size of the re-mapping table is 128 lines (each line addresses 4 constants). The write granularity is 4 constants or 512 bits. It takes 16 clocks to write the four constants. Real time requires 256 lines in the physical memory (this is physically register mapped).

There will be two of those memories and two of each remapping read memories.

The texture state is also kept in a similar memory. The size of this memory is 320x96 bits (128 texture states for regular mode, 32 states for RT). The memory thus holds 128 texture states (192 bits per state). The logical size exposes 32 different states total, which are going to be shared between the pixel and the vertex shader. The size of the re-mapping table to for the texture state memory is 32 lines (each line addresses 1 texture state lines in the real

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
	9 July, 2003	4 September, 2015		14 of 54

memory). The CP write granularity is 1 texture state lines (or 192 bits). The driver sends 512 bits but the CP ignores the top 320 bits. It thus takes 6 clocks to write the texture state. Real time requires 32 lines in the physical memory (this is physically register mapped).

The control flow constant memory doesn't sit behind a renaming table. It is register mapped and thus the driver must reload its content each time there is a change in the control flow constants. Its size is 320*32 because it must hold 8 copies of the 32 dwords of control flow constants and the loop construct constants must be aligned.

The constant re-mapping tables for texture state and ALU constants are logically register mapped for regular mode and physically register mapped for RT operation.

5.2 Management of the Control Flow Constants

The control flow constants are register mapped, thus the CP writes to the according register to set the constant, the SQ decodes the address and writes to the block pointed by its current base pointer (CF_WR_BASE). On the read side, one level of indirection is used. A register (SQ_CONTEXT_MISC.CF_RD_BASE) keeps the current base pointer to the control flow block. This register is copied whenever there is a state change. Should the CP write to CF after the state change, the base register is updated with the (current pointer number +1)% number of states. This way, if the CP doesn't write to CF the state is going to use the previous CF constants.

5.3 Management of the re-mapping tables

5.3.1 R400 Constant management

The sequencer is responsible to manage two re-mapping tables (one for the constant store and one for the texture state). On a state change (by the driver), the sequencer will broadside copy the contents of its re-mapping tables to a new one. We have 8 different re-mapping tables we can use concurrently.

The constant memory update will be incremental, the driver only need to update the constants that actually changed between the two state changes.

For this model to work in its simplest form, the requirement is that the physical memory MUST be at least twice as large as the logical address space + the space allocated for Real Time. In our case, since the logical address space is 512 and the reserved RT space can be up to 256 entries, the memory must be of sizes 1280 and above. Similarly the size of the texture store must be of 32*2+32 = 96 entries and above.

5.3.2 Dirty bits

Two sets of dirty bits will be maintained per logical address. The first one will be set to zero on reset and set when the logical address is addressed. The second one will be set to zero whenever a new context is written and set for each address written while in this context. The reset dirty is not set, then writing to that logical address will not require de-allocation of whatever address stored in the renaming table. If it is set and the context dirty is not set, then the physical address store needs to be de-allocated and a new physical address is necessary to store the incoming data. If they are both set, then the data will be written into the physical address held in the renaming for the current logical address. No de-allocation or allocation takes place. This will happen when the driver does a set constant twice to the same logical address between context changes. NOTE: It is important to detect and prevent this, failure to do it will allow multiple writes to allocate all physical memory and thus hang because a context will not fit for rendering to start and thus free up space.

5.3.3 Free List Block

A free list block that would consist of a counter (called the IFC or Initial Free Counter) that would reset to zero and incremented every time a chunk of physical memory is used until they have all been used once. This counter would be checked each time a physical block is needed, and if the original ones have not been used up, us a new one, else check the free list for an available physical block address. The count is the physical address for when getting a chunk from the counter.

Storage of a free list big enough to store all physical block addresses.

Maintain three pointers for the free list that are reset to zero. The first one we will call write_ptr. This pointer will identify the next location to write the physical address of a block to be de-allocated. Note: we can never free more

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
9 July, 2003	4 September, 2015	GEN-CXXXXX-REVA	15 of 54

physical memory locations than we have. Once recording address the pointer will be incremented to walk the free list like a ring.

The second pointer will be called stop_ptr. The stop_ptr pointer will be advanced by the number of address chunks de-allocates when a context finishes. The address between the stop_ptr and write_ptr cannot be reused because they are still in use. But as soon as the context using then is dismissed the stop_ptr will be advanced.

The third pointer will be called read_ptr. This pointer will point will point to the next address that can be used for allocation as long as the read_ptr does not equal the stop_ptr and the IFC is at its maximum count.

5.3.4 De-allocate Block

This block will maintain a free physical address block count for each context. While in current context, a count shall be maintained specifying how many blocks were written into the free list at the write_ptr pointer. This count will be reset upon reset or when this context is active on the back and different than the previous context. It is actually a count of blocks in the previous context that will no longer be used. This count will be used to advance the write_ptr pointer to make available the set of physical blocks freed when the previous context was done. This allows the discard or de-allocation of any number of blocks in one clock.

5.3.5 Operation of Incremental model

The basic operation of the model would start with the write_ptr, stop_ptr, read_ptr pointers in the free list set to zero and the free list counter is set to zero. Also all the dirty bits and the previous context will be initialized to zero. When the first set constants happen, the reset dirty bit will not be set, so we will allocate a physical location from the free list counter because its not at the max value. The data will be written into physical address zero. Both the additional copy of the renaming table and the context zeros of the big renaming table will be updated for the logical address that was written by set start with physical address of 0. This process will be repeated for any logical address that are not dirty until the context changes. If a logical address is hit that has its dirty bits set while in the same context, both dirty bits would be set, so the new data will be over-written to the last physical address assigned for this logical address. When the first draw command of the context is detected, the previous context stored in the additional renaming table will be copied to the larger renaming table in the current (new) context location. Then the set constant logical address it replaced in the renaming table would be entered at the write_ptr pointer location on the free list and the write_ptr will be incremented. The de-allocation counter for the previous context (eight) will be incremented. This as set states come in for this context one of the following will happen:

- 1.) No dirty bits are set for the logical address being updated. A line will be allocated of the free-list counter or the free list at read_ptr pointer if read_ptr != to stop_ptr .
- Reset dirty set and Context dirty not set. A new physical address is allocated, the physical address in the renaming table is put on the free list at write_ptr and it is incremented along with the de-allocate counter for the last context.
- 3.) Context dirty is set then the data will be written into the physical address specified by the logical address.

This process will continue as long as set states arrive. This block will provide backpressure to the CP whenever he has not free list entries available (counter at max and stop_ptr == read_ptr). The command stream will keep a count of contexts of constants in use and prevent more than max constants contexts from being sent.

Whenever a draw packet arrives, the content of the re-mapping table is written to the correct re-mapping table for the context number. Also if the next context uses less constants than the current one all exceeding lines are moved to the free list to be de-allocated later. This happens in parallel with the writing of the re-mapping table to the correct memory.

Now preferable when the constant context leaves the last ALU clause it will be sent to this block and compared with the previous context that left. (Init to zero) If they differ than the older context will no longer be referenced and thus can be de-allocated in the physical memory. This is accomplished by adding the number of blocks freed this context to the stop_ptr pointer. This will make all the physical addresses used by this context available to the read_ptr allocate pointer for future allocation.

This device allows representation of multiple contexts of constants data with N copies of the logical address space. It also allows the second context to be represented as the first set plus some new additional data by just storing the delta's. It allows memory to be efficiently used and when the constants updates are small it can store multiple

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257937

ATI Ex. 2109 IPR2023-00922 Page 227 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
9 July, 2003	4 September, 2015		16 of 54

context. However, if the updates are large, less contexts will be stored and potentially performance will be degraded. Although it will still perform as well as a ring could in this case.

5.4 Constant Store Indexing

In order to do constant store indexing, the sequencer must be loaded first with the indexes (that come from the GPRs). There are 144 wires from the exit of the SP to the sequencer (9 bits pointers x 16 vertexes/clock).

MOVA R1.X,R2.X // Loads the sequencer with the content of R2.X, also copies the content of R2.X into R1.X ADD R3,R4,C0[R2.X]// Uses the state from the sequencer to add R4 to C0[R2.X] into R3

Note that we don't really care about what is in the brackets because we use the state from the MOVA instruction. R2.X is just written again for the sake of simplicity and coherency.

The storage needed in the sequencer in order to support this feature is 2*64*9 bits = 1152 bits.

The address register is a signed integer, which ranges from -256 to 255.

The address register is not kept across clause boundaries. As such, it must be refreshed after any Serialize (or yield), allocate instruction or resource change. Failure to refresh the address register will result in unpredictable behavior.

5.5 Real Time Commands

The real time commands constants are written by the CP using the register mapped registers allocated for RT. It works is the same way than when dealing with regular constant loads BUT in this case the CP is not sending a logical address but rather a physical address and the reads are not passing thru the re-mapping table but are directly read from the memory. The boundary between the two zones is defined by the CONST_EO_RT control register. Similarly, for the fetch state, the boundary between the two zones is defined by the TSTATE_EO_RT control register.

5.6 Constant Waterfalling

In order to have a reasonable performance in the case of constant store indexing using the address register, we are going to have the possibility of using the physical memory port for read only. This way we can read 1 constant per clock and thus have a worst-case waterfall mode of 1 vertex per clock. There is a small synchronization issue related with this as we need for the SQ to make sure that the constants where actually written to memory (not only sent to the sequencer) before it can allow the first vector of pixels or vertices of the state to go thru the ALUs. To do so, the sequencer keeps 8 bits (one per render state) and sets the bits whenever the last render state is written to memory and clears the bit whenever a state is freed.

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

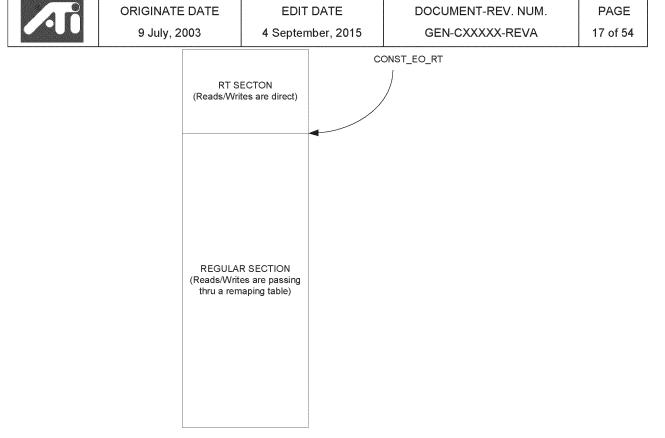


Figure 7: The Constant store

6. Looping and Branches

Loops and branches are planned to be supported and will have to be dealt with at the sequencer level. We plan on supporting constant loops and branches using a control program.

6.1 The controlling state.

The R400 controling state consists of:

Boolean[255:0] Loop_count[7:0][31:0] Loop_Start[7:0][31:0] Loop_Step[7:0][31:0]

That is 256 Booleans and 32 loops.

We have a stack of 4 elements for nested calls of subroutines and 4 loop counters to allow for nested loops.

This state is available on a per shader program basis.

6.2 The Control Flow Program

We'd like to be able to code up a program of the form:

1: Loop 2: Exec TexFetch

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE DATE 9 July, 2003	EDIT DATE 4 September, 2015	R400 Sequencer Specification	PAGE 18 of 54
3:	TexFetch			
4:	ALU			
5:	ALU			
6:	TexFetch			
7:	End Loop			
8:	ALU Export			

But realize that 3: may be dependent on 2: and 4: is almost certainly dependent on 2: and 3:. Without clausing, these dependencies need to be expressed in the Control Flow instructions. Additionally, without separate 'texture clauses' and 'ALU clauses' we need to know which instructions to dispatch to the Texture Unit and which to the ALU unit. This information will be encapsulated in the flow control instructions.

Each control flow instruction will contain 2 bits of information for each (non-control flow) instruction:

a) ALU or Texture

b) Serialize Execution

(b) would force the thread to stop execution at this point (before the instruction is executed) and wait until all textures have been fetched. Given the allocation of reserved bits, this would mean that the count of an 'Exec' instruction would be limited to about 8 (non-control-flow) instructions. If more than this were needed, a second Exec (with the same conditions) would be issued.

Another function that relies upon 'clauses' is allocation and order of execution. We need to assure that pixels and vertices are exported in the correct order (even if not all execution is ordered) and that space in the output buffers are allocated in order. Additionally data can't be exported until space is allocated. A new control flow instruction:

Alloc <buffer select -- position, parameter, pixel or vertex memory. And the size required>.

would be created to mark where such allocation needs to be done. To assure allocation is done in order, the actual allocation for a given thread can not be performed unless the equivalent allocation for all previous threads is already completed. The implementation would also assure that execution of instruction(s) following the serialization due to the Alloc will occur in order -- at least until the next serialization or change from ALU to Texture. In most cases this will allow the exports to occur without any further synchronization. Only 'final' allocations or position allocations are guaranteed to be ordered. Because strict ordering is required for pixels, parameters and positions, this implies only a single alloc for these structures. Vertex exports to memory do not require ordering during allocation and so multiple 'allocs' may be done.

6.2.1 Control flow instructions table

Here is the revised control flow instruction set.

Note that whenever a field is marked as RESERVED, it is assumed that all the bits of the field are cleared (0).

	NOP					
47 44	43	42 0				
0000	Addressing	RESERVED				

This is a regular NOP.

Execute						
47 44	43	40 34	33 28	2716	1512	11 0
0001	Addressing	RESERV	Vertex	Instructions type + serialize (6	Count	Exec Address
	-	ED	Cache	instructions)		

	Execute_End								
47 44	43	40 34	33 28	2716	1512	11 0			
0010	Addressing	RESERV ED	Vertex Cache	Instructions type + serialize (6 instructions)	Count	Exec Address			

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257940

ATI Ex. 2109 IPR2023-00922 Page 230 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
9 July, 2003	4 September, 2015	GEN-CXXXXX-REVA	19 of 54

Execute up to 6 instructions at the specified address in the instruction memory. The Instruction type field tells the sequencer the type of the instruction (LSB) (1 = Texture, 0 = ALU and whether to serialize or not the execution (MSB) (1 = Serialize, 0 = Non-Serialized). If the corresponding VC bit is set then VC is used instead of TP/ALU. If Execute_End this is the last execution block of the shader program.

Vertex Cache	Serialize	Instruction Type (Resource)	
0	0	0	: ALU instruction, not yielding
0	0	1	: Texture instruction, not yielding
0	1	0	: ALU instruction, yielding
0	1	1	: Texture instruction, yielding
1	0	0	: Vertex cache instruction, not yielding
1	0	1	: Vertex cache instruction, not yielding
1	1	0	: Vertex cache instruction, yielding
1	1	1	: Vertex cache instruction, yielding

	Conditional_Execute										
47 44	43	42	41 34	3328	2716	1512	11 0				
0011	Addressing	Condition	Boolean address	Vertex Cache	Instructions type + serialize (6 instructions)	Count	Exec Address				

	Conditional_Execute_End										
47 44	43	42	41 34	3328	2716	1512	11 0				
0100	Addressing	Condition	Boolean address	Vertex Cache	Instructions type + serialize (6 instructions)	Count	Exec Address				

If the specified Boolean (8 bits can address 256 Booleans) meets the specified condition then execute the specified instructions (up to 9 instructions). If the condition is not met, we go on to the next control flow instruction. If Conditional_Execute_End and the condition is met, this is the last execution block of the shader program.

	Conditional_Execute_Predicates										
47 44	43	42	41 36	35 34	3328	2716	1512	11 0			
0101	Addressing	Condition	RESERVED	Predicate vector	Vertex Cache	Instructions type + serialize (6 instructions)	Count	Exec Address			

	Conditional_Execute_Predicates_End											
47 44	43	42	2716	1512	11 0							
0110	Addressing	Condition	RESERVED	Predicate vector	Vertex Cache	Instructions type + serialize (6 instructions)	Count	Exec Address				

Check the AND/OR of all current predicate bits. If AND/OR matches the condition execute the specified number of instructions. We need to AND/OR this with the kill mask in order not to consider the pixels that aren't valid. If the condition is not met, we go on to the next control flow instruction. If Conditional_Execute_Predicates_End and the condition is met, this is the last execution block of the shader program.

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

			NATE DATE	EDIT	DATE	R400 S	PAGE		
	Car	9 Ji	uly, 2003	4 Septem	4 September, 2015				20 of 54
	C		Conditional_Ex	cecute_Predic	ates_No_	Stall			
[47 44	43	42	41 36	35 34	3328	2716	1512	11 0
	1101	Addressing	Condition	RESERVED	Predicate	Vertex	Instructions	Count	Exec Address
					vector	Cache	type +		
							serialize (6		
						instructions)			

·····	Conditional_Execute_Predicates_No_Stall_End										
47 44	43	42	41 36	35 34	3328	2716	1512	11 0			
1110	Addressing	Condition	RESERVED	Predicate vector	Vertex Cache	Instructions type +	Count	Exec Address			
						serialize (6					
						instructions)					

Same as Conditionnal_Execute_Predicates but the SQ is not going to wait for the predicate vector to be updated. You can only set this in the compiler if you know that the predicate set is only a refinement of the current one (like a nested if) because the optimization would still work.

	Loop_Start							
47 44 43 42 21 20 16 1513								
0111	Addressing	RESERVED	loop ID	RESERVED	Jump address			

Loop Start. Compares the loop iterator with the end value. If loop condition not met jump to the address. Forward jump only. Also computes the index value. The loop id must match between the start to end, and also indicates which control flow constants should be used with the loop.

	Loop_End									
4744	43	42	41 36	3534	33 22	21	20 16	1513	12 0	
1000	Addressing	Cond	RESERVED	Predicate	RESERVED	Pred	loop ID	RESERVED	start	
				Vector		break	-		address	

Loop end. Increments the counter by one, compares the loop count with the end value. If loop condition met, continue, else, jump BACK to the start of the loop. If predicate break != 0, then compares predicate vector n (specified by predicate Vector) to condition. If all bits meet condition then break the loop.

The way this is described does not prevent nested loops, and the inclusion of the loop id make this easy to do.

Conditionnal_Call									
47 44	43	42	41 34 33 14 1			12 0			
1001	Addressing	Condition	Boolean address	RESERVED	Force Call	Jump address			

If the condition is met, jumps to the specified address and pushes the control flow program counter on the stack. If force call is set the condition is ignored and the call is made always.

	Return				
47 44 43 42 0					
1010	Addressing	RESERVED			

Pops the topmost address from the stack and jumps to that address. If nothing is on the stack, the program will just continue to the next instruction.

Conditionnal_Jump							
47 44	47 44 43 42 41 34 33 32 14 13 12 0						
1011	Addressing	Condition	Boolean	FW only	RESERVED	Force Jump	Jump address
		suurooree	address	-			-

If force jump is set the condition is ignored and the jump is made always. If FW only is set then only forward jumps are allowed.

Exhibit 2039.doc 84302 Bytes*** [©] ATI Confidential. Reference Copyright Notice on Cover Page [©] ***

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	9 July, 2003	4 September, 2015	GEN-CXXXXX-REVA	21 of 54

Allocate					
47 44	43	4241	40	39 3	20
1100	Debug	Buffer Select	No Serial	RESERVED	Size

Buffer Select takes a value of the following:

01 – position export (ordered export)

10 – parameter cache or pixel export (ordered export)

11 – pass thru (out of order exports).

Size field is only used to reserve space in the export buffer for pass thru exports. Valid values are 1 (1 line) thru 5 (5 lines). It should be determined by the compiler/assembler by taking max index used +1.

If debug is set this is a debug alloc (ignore if debug DB_ON register is set to off).

By default the serial bit is set on an alloc. If the No Serial bit is asserted then the serial bit won't be set in the SQ.

6.2.2 Alloc Statements

Alloc statements are control flow instructions that allocate resources that are required for executable export instructions. An alloc statement can be either a normal yield point, or a partial yield. At a partial yield - hardware releases the gpu so all state (mova, grad etc) is lost but the thread can resume before all pending fetches have completed.

There are three types of allocs:

alloc-position - proceeds the export of position from a vertex shader. A vertex shader must include one alloc of position. A position alloc cannot be used in a pixel shader. all exports for position must execute between the alloc-position and the next yield point or resource change. There is a small performance advantage to placing the alloc-position near the top of the vertex shader. However we don't think this is worth adding an extra instruction or register to the shader.

alloc-interp/color - proceeds interpolator exports in a vertex shader or the color exports in a pixel shader. There can be only one alloc interp/color per shader. The color alloc in a pixel shader must be after any alloc-mem-exports. The actual exports can occur anywhere between the alloc-interp/color and the end of the program. There is a small performance advantage to placing the alloc-interp/color near the bottom of the shader. However we don't think this is worth adding an extra instruction or register to the shader. There is a big performance advantage of having no fetches of any kind after the alloc-interp/color.

alloc-mem-export - proceeds any memory-address, memory-data exports. There can be multiple alloc-mem-export statements in either kind of shader. All exports for mem-exports must execute between the corresponding alloc-mem-export and the next yield point or resource change.

6.3 Implementation

The envisioned implementation has a buffer that maintains the state of each thread. A thread lives in a given location in the buffer during its entire life, but the buffer has FIFO qualities in that threads leave in the order that they enter. Actually two buffers are maintained -- one for Vertices and one for Pixels. The intended implementation would allow for:

16 entries for vertices 48 entries for pixels.

From each buffer, arbitration logic attempts to select 1 thread for the texture unit and 2 (interleaved) thread for the ALU unit. Once a thread is selected it is read out of the buffer, marked as invalid, and submitted to appropriate execution unit. It is returned to the buffer (at the same place) with its status updated once all possible sequential

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257943

ATI Ex. 2109 IPR2023-00922 Page 233 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
9 July, 2003	4 September, 2015		22 of 54

instructions have been executed. A switch from ALU to TEX or visa-versa or a Serialize_Execution modifier forces the thread to be returned to the buffer.

Each entry in the buffer will be stored across two physical pieces of memory - most bits will be stored in a 1 read port device. Only bits needed for thread arbitration will be stored in a highly multi-ported structure. The bits kept in the 1 read port device will be termed 'state'. The bits kept in the multi-read ported device will be termed 'status'.

'State Bits' needed include:

- 1. Control Flow Instruction Pointer (13 bits),
- 2. Execution Count Marker 4 bits),
- 3. Loop Iterators (4x9 bits),
- 4. Loop Counters (4x9 bits),
- 5. Call return pointers (4x13 bits),
- 6. Predicate Bits (64 bits),
- 7. Export ID (4 bits),
- 8. Parameter Cache base Ptr (7 bits),
- 9. GPR Base Ptr (8 bits),
- 10. Context Ptr (3 bits).
- 11. LOD corrections (6x16 bits)
- 12. Valid bits (64 bits)
- 13. RT (1 bit) Signifies that this thread is a Real Time thread. This bit must be sent to the Constant store state machine when reading it.

Absent from this list are 'Index' pointers. These are costly enough that I'm presuming that they are instead stored in the GPRs. The first seven fields above (Control Flow Ptr, Execution Count, Loop Counts, call return ptrs, Predicate bits, PC base ptr and export ID) are updated every time the thread is returned to the buffer based on how much progress has been mode on thread execution. GPR Base Ptr, Context Ptr and LOD corrections are unchanged throughout execution of the thread.

'Status Bits' needed include:

- Valid Thread
- ALU engine needed
- Texture engine needed
- VC engine needed
- Texture Reads are outstanding
- VC Reads are outstanding
- Alu bank (0/1)
- Waiting on Texture Read to Complete
- Allocation Wait (2 bits)
- 00 No allocation needed
- 01 Position export allocation needed (ordered export)
- 10 Parameter or pixel export needed (ordered export)
- 11 pass thru (out of order export)
- Allocation Size (4 bits)
- Position Allocated
- Mem/Color Allocated
- First thread of a new context
- Event thread (NULL thread that needs to trickle down the pipe)
- Last (1 bit)
- Pulse SX (1 bit)

All of the above fields from all of the entries go into the arbitration circuitry. The arbitration circuitry will select a winner for both the Texture Engine and for the ALU engine. There are actually two sets of arbitration -- one for pixels and one for vertices. A final selection is then done between the two. But the rest of this implementation summary only considers the 'first' level selection which is similar for both pixels and vertices.

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	9 July, 2003	4 September, 2015	GEN-CXXXXX-REVA	23 of 54

Texture arbitration requires no allocation or ordering so it is purely based on selecting the 'oldest' thread that requires the Texture Engine.

ALU arbitration is a little more complicated. First, only threads where either of Texture_Reads_outstanding or Waiting_on_Texture_Read_to_Complete are '0' are considered. Then if Allocation_Wait is active, these threads are further filtered based on whether space is available. If the allocation is position allocation, then the thread is only considered if all 'older' threads have already done their position allocation (position allocated bits set). If the allocation is parameter or pixel allocation, then the thread is only considered if it is a parameter or pixel allocation, then the thread is only considered if it is a parameter or pixel or position allocation, has its First_thread_of_a_new_context bit set and would cause ALU interleaving with another thread performing the same parameter or pixel or position allocation. Finally the 'oldest' of the threads that pass through the above filters is selected. If the thread needed to allocate, then at this time the allocation is done, based on Allocation_Size. If a thread has its "last" bit set, then it is also removed from the buffer, never to return.

If I now redefine 'clauses' to mean 'how many times the thread is removed from the thread buffer for the purpose of exection by either the ALU or Texture engine', then the minimum number of clauses needed is 2 -- one to perform the allocation for exports (execution automatically halts after an 'Alloc' instruction) (but doesn't performs the actual allocation) and one for the actual ALU/export instructions. As the 'Alloc' instruction could be part of a texture clause (presumably the final instruction in such a clause), a thread could still execute in this minimal number of 2 clauses, even if it involved texture fetching.

The Texture_Reads_Outstanding and VC_reads_Outstanding bits tell the SQ that a texture or VC read is outstanding. In this case, if we encounter a serial bit we need to wait until both resources are free (pending = 0) in order to proceed.

6.4 Data dependant predicate instructions

Data dependant conditionals will be supported in the R400. The only way we plan to support those is by supporting three vector/scalar predicate operations of the form:

PRED_SETE_PUSH - similar to SETE except that the result is 'exported' to the sequencer. PRED_SETNE_PUSH - similar to SETNE except that the result is 'exported' to the sequencer. PRED_SETGT_PUSH - similar to SETGT except that the result is 'exported' to the sequencer PRED_SETGTE_PUSH - similar to SETGTE except that the result is 'exported' to the sequencer

For the scalar operations only we will also support the two following instructions:

PRED_SETE PRED_SETNE PRED_SETGT PRED_SET_INV PRED_SET_POP PRED_SET_CLR PRED_SET_RESTORE

Details about actual implementation of these opcodes are in the shader pipe architectural spec.

The export is a single bit - 1 or 0 that is sent using the same data path as the MOVA instruction. The sequencer will maintain 1 set of 64 bits predicate vectors (in fact 2 sets because we interleave two programs but only 1 will be exposed) and use it to control the write masking. This predicate is maintained across clause boundaries.

Then we have two conditional execute bits. The first bit is a conditional execute "on" bit and the second bit tells us if we execute on 1 or 0. For example, the instruction:

P0_ ADD_# R0,R1,R2

Is only going to write the result of the ADD into those GPRs whose predicate bit is 0. Alternatively, P1_ADD_# would only write the results to the GPRs whose predicate bit is set. The use of the P0 or P1 without precharging the sequencer with a PRED instruction is undefined.

Exhibit 2039.doc 84302 Bytes*** [©] ATI Confidential. Reference Copyright Notice on Cover Page [©] ***

AMD1044_0257945

ATI Ex. 2109 IPR2023-00922 Page 235 of 326

AÎ	ORIGINATE DATE 9 July, 2003	EDIT DATE 4 September, 2015	R400 Sequencer Specification	PAGE 24 of 54
	5 July, 2005	4 Oeptember, 2015		24 01 34

6.5 HW Detection of PV,PS

Because of the control program, the compiler cannot detect statically dependant instructions. In the case of nonmasked writes and subsequent reads the sequencer will insert uses of PV,PS as needed. This will be done by comparing the read address and the write address of consecutive instructions. For masked writes, the sequencer will insert detect wich channels to read from the GPRs and which ones to read from the PV/PS.

6.6 Register file indexing

Because we can have loops in fetch clause, we need to be able to index into the register file in order to retrieve the data created in a fetch clause loop and use it into an ALU clause. The instruction will include the base address for register indexing and the instruction will contain these controls:

Bit7	Bit 6	
0	0	'absolute register'
0	1	'relative register'
1	0	'previous vector'
1	1	'previous scalar'

In the case of an absolute register we just take the address as is. In the case of a relative register read we take the base address and we add to it the loop_index and this becomes our new address that we give to the shader pipe.

The sequencer is going to keep a loop index computed as such:

Index = Loop_iterator*Loop_step + Loop_start.

We loop until loop_iterator = loop_count. Loop_step is a signed value [-128...127]. The computed index value is a 10 bit counter that is also signed. Its real range is [-256,256]. The tenth bit is only there so that we can provide an out of range value to the "indexing logic" so that it knows when the provided index is out of range and thus can make the necessary arrangements.

6.7 Debugging the Shaders

In order to be able to debug the pixel/vertex shaders efficiently, we provide 2 methods.

6.7.1 *Method 1: Debugging registers*

Current plans are to expose 2 debugging, or error notification, registers:

- 1. address register where the first error occurred
- 2. count of the number of errors

The sequencer will detect the following groups of errors:

- count overflow
- constant indexing overflow
- register indexing overflow

Compiler recognizable errors:

- jump errors
 - relative jump address > size of the control flow program
- call stack
 - call with stack full return with stack empty

With all the other errors, program can continue to run, potentially to worst-case limits.

If indexing outside of the constant or the register range, causing an overflow error, the hardware is specified to return the value with an index of 0. This could be exploited to generate error tokens, by reserving and initializing the 0th register (or constant) for errors.

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

 I				
9 July, 2003	4 September, 2015	GEN-CXXXXX-REVA	25 of 54	
ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	

{ISSUE : Interrupt to the driver or not?}

6.7.2 *Method 2: Exporting the values in the GPRs*

1) The sequencer will have a debug active, count register and an address register for this mode.

Under the normal mode execution follows the normal course.

Under the debug mode it is assumed that the program is always exporting n debug vectors and that all other exports to the SX block (but for position) will be turned off (changed into NOPs) by the sequencer (even if they occur before the address stated by the ADDR debug register).

7. Pixel Kill Mask

A vector of 64 bits is kept by the sequencer per group of pixels/vertices. Its purpose is to optimize the texture fetch requests and allow the shader pipe to kill pixels using the following instructions:

MASK_SETE MASK_SETNE MASK_SETGT MASK_SETGTE

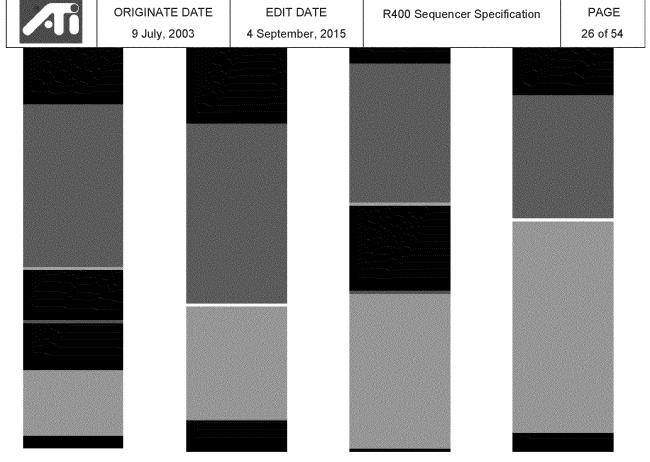
8. Register file allocation

The register file allocation for vertices and pixels can either be static or dynamic. In both cases, the register file in managed using two round robins (one for pixels and one for vertices). In the dynamic case the boundary between pixels and vertices is allowed to move, in the static case it is fixed to 128-VERTEX_REG_SIZE for vertices and PIXEL_REG_SIZE for pixels.

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257947

ATI Ex. 2109 IPR2023-00922 Page 237 of 326



Above is an example of how the algorithm works. Vertices come in from top to bottom; pixels come in from bottom to top. Vertices are in orange and pixels in green. The blue line is the tail of the vertices and the green line is the tail of the pixels. Thus anything between the two lines is shared. When pixels meets vertices the line turns white and the boundary is static until both vertices and pixels share the same "unallocated bubble". Then the boundary is allowed to move again. The numbering of the GPRs starts from the bottom of the picture at index 0 and goes up to the top at index 127.

9. Fetch Arbitration

The fetch arbitration logic chooses one of the n potentially pending fetch clauses to be executed. The choice is made by looking at the Vs and Ps reservation stations and picking the first one ready to execute. Once chosen, the clause state machine will send one 2x2 fetch per clock (or 4 fetches in one clock every 4 clocks) until all the fetch instructions of the clause are sent. This means that there cannot be any dependencies between two fetches of the same clause.

The arbitrator will not wait for the fetches to return prior to selecting another clause for execution. The fetch pipe will be able to handle up to X(?) in flight fetches and thus there can be a fair number of active clauses waiting for their fetch return data.

10. VC Arbitration

The VC arbitration logic chooses one of the n potentially pending VC clauses to be executed. The choice is made by looking at the Vs and Ps reservation stations and picking the first one ready to execute. Once chosen, the clause state machine will send one 2x2 fetch per clock (or 4 fetches in one clock every 4 clocks) until all the fetch instructions of the clause are sent. This means that there cannot be any dependencies between two fetches of the same clause.

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
9 July, 2003	4 September, 2015	GEN-CXXXXX-REVA	27 of 54

The arbitrator will not wait for the fetches to return prior to selecting another clause for execution. The VC pipe will be able to handle up to X(?) in flight VC fetches and thus there can be a fair number of active clauses waiting for their fetch return data.

11. ALU Arbitration

ALU arbitration proceeds in almost the same way than fetch arbitration. The ALU arbitration logic chooses one of the n potentially pending ALU clauses to be executed. The choice is made by looking at the Vs and Ps reservation stations and picking the first one ready to execute. There are two ALU arbitres, one for the even clocks and one for the odd clocks. For example, here is the sequencing of two interleaved ALU clauses (E and O stands for Even and Odd sets of 4 clocks):

Einst0 Oinst0 Einst1 Oinst1 Einst2 Oinst2 Einst0 Oinst3 Einst1 Oinst4 Einst2 Oinst0... Proceeding this way hides the latency of 8 clocks of the ALUs. Also note that the interleaving also occurs across clause boundaries.

12. Handling Stalls

When the output file is full, the sequencer prevents the ALU arbitration logic from selecting the last clause (this way nothing can exit the shader pipe until there is place in the output file. If the packet is a vertex packet and the position buffer is full (POS_FULL) then the sequencer also prevents a thread from entering an exporting clause. The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU arbiter will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

12.1 SP stall conditions

12.1.1 PS Stalls

None.

12.1.2 PV Stalls

None.

13. Content of the reservation station FIFOs

The reservation FIFOs contain the state of the vector of pixels and vertices. We have two sets of those: one for pixels, and one for vertices. They contain 3 bits of Render State 7 bits for the base address of the GPRs, some bits for LOD correction and coverage mask information in order to fetch fetch for only valid pixels, the quad address.

14. The Output File

The output file is where pixels are put before they go to the RBs. The write BW to this store is 256 bits/clock. Just before this output file are staging registers with write BW 512 bits/clock and read BW 256 bits/clock. The staging registers are 4x128 (and there are 16 of those on the whole chip).

15. IJ Format

The IJ information sent by the PA is of this format on a per quad basis:

We have a vector of IJ's (one IJ per pixel at the centroid of the fragment or at the center of the pixel depending on the mode bit). All pixel's parameters are always interpolated at full 20x24 mantissa precision.

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257949

ATI Ex. 2109 IPR2023-00922 Page 239 of 326

	ORIGINATE DATE 9 July, 2003	EDIT DATE 4 September, 20	15	R4	00 Sequenc	er Specification	PAGE 28 of 54
P1 = A + I	$\frac{J}{I(0) * (B - A) + J(0)}$ (1) * (B - A) + J(1) I(2) * (B - A) + J(2)	*(C-A)	P0		P1		
P3 = A + b	J(3) * (B - A) + J(3)	(C - A)	P2		P3		

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

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

Total number of bits : 20*8 + 4*8 + 4*2 = 200.

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 +/- 1024.

15.1 Interpolation of constant attributes

Because of the floating point imprecision, we need to take special provisions if all the interpolated terms are the same or if two of the terms are the same.

16. Staging Registers

In order for the reuse of the vertices to be 14, the sequencer will have to re-order the data sent IN ORDER by the VGT for it to be aligned with the parameter cache memory arrangement. Given the following group of vertices sent by the VGT:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 || 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 || 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 || 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

The sequencer will re-arrange them in this fashion:

0 1 2 3 16 17 18 19 32 33 34 35 48 49 50 51 || 4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 || 8 9 10 11 24 25 26 27 40 41 42 43 56 57 58 59 || 12 13 14 15 28 29 30 31 44 45 46 47 60 61 62 63

The || markers show the SP divisions. In the event a shader pipe is broken, the SQ is responsible to insert padding to account for the missing pipe. For example, if SP1 is broken, vertices 4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 will not be sent by the VGT to the SQ **AND** the SQ is responsible to "jump" over these vertices in order for no valid vertices to be sent to an invalid SP.

The most straightforward, *non-compressed* interface method would be to convert, in the VGT, the data to 32-bit floating point prior to transmission to the VSISRs. In this scenario, the data would be transmitted to (and stored in) the VSISRs in full 32-bit floating point. This method requires three 24-bit fixed-to-float converters in the VGT. Unfortunately, it also requires and additional 3,072 bits of storage across the VSISRs. This interface is illustrated in Figure 9. The area of the fixed-to-float converters and the VSISRs for this method is roughly estimated as 0.759sqmm using the R300 process. The gate count estimate is shown in Figure 8.

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257950

ATI Ex. 2109 IPR2023-00922 Page 240 of 326

	0)	
	n	

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
9 July, 2003	4 September, 2015	GEN-CXXXXX-REVA	29 of 54

8x24-bit	11631	μ^2	$60.57813\mu^2perbit$
Area of 96x8-deep Latch Memory	46524	μ^2	
Area of 24-bit Fix-to-float Converter	4712	μ^2 per conve	erter
Method 1	Block	Quantity	Area
	F2F	3	14136
	8x96 Latch	16_	744384
			758520 µ ²

Figure 8: Area Estimate for VGT to Shader Interface

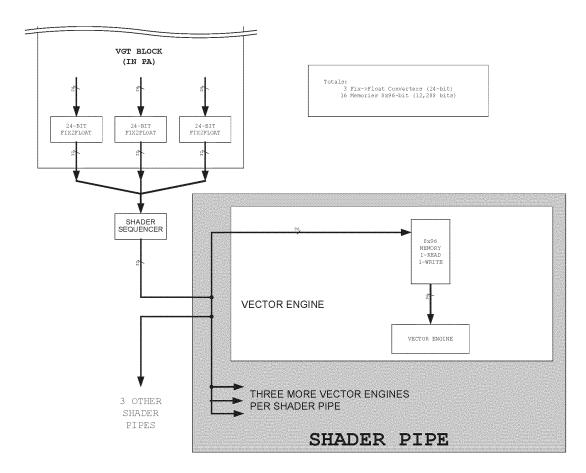


Figure 9:VGT to Shader Interface

17. The parameter cache

The parameter cache is where the vertex shaders export their data. It consists of 16 128x128 memories (1R/1W). The reuse engine will make it so that all vertexes of a given primitive will hit different memories. The allocation

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257951

ATI Ex. 2109 IPR2023-00922 Page 241 of 326

ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
9 July, 2003	4 September, 2015		30 of 54

method for these memories is a simple round robin. The parameter cache pointers are mapped in the following way: 4MSBs are the memory number and the 7 LSBs are the address within this memory.

MEMORY NUMBER	ADDRESS
4 bits	7 bits

The PA generates the parameter cache addresses as the positions come from the SQ. All it needs to do is keep a Current_Location pointer (7 bits only) and as the positions comes increment the memory number. When the memory number field wraps around, the PA increments the Current_Location by VS_EXPORT_COUNT (a snooped register from the SQ). As an example, say the memories are all empty to begin with and the vertex shader is exporting 8 parameters per vertex (VS_EXPORT_COUNT = 8). The first position received is going to have the PC address 00000000000 the second one 00010000000, third one 0010000000 and so on up to 11110000000. Then the next position received (the 17th) is going to have the address 00000001000, the 18th 00010001000, the 19th 00100001000 and so on. The Current_location is NEVER reset BUT on chip resets. The only thing to be careful about is that if the SX doesn't send you a full group of positions (<64) then you need to fill the address space so that the next group starts correctly aligned (for example if you receive only 33 positions then you need to add 2*VS_EXPORT_COUNT to Current_Location and reset the memory count to 0 before the next vector begins).

17.1 Export restrictions

17.1.1 Pixel exports:

Pixels can export 1,2,3 or 4 color buffers to the SX(+z). The exports will be done in order. The exports will always be ordered to the SX.

17.1.2 Vertex exports:

Position or parameter caches can be exported in any order in the shader program. It is always better to export posistion as soon as possible. Position has to be exported in a single export block (no texture instructions can be placed between the exports). Parameter cache exports can be done in any order with texture instructions interleaved. The exports will always be allocated in order to the SX.

17.1.3 Pass thru exports:

Pass thru exports have to be done in groups of the form:

```
Alloc 1 thru 5 (max export offset + 1, for example if using EM4 alloc size 5) Execute ALU(ADDR) ALU(DATA) ALU(DATA) ALU(DATA)...
```

When exporting to more than EM0, one MUST write to EM4 also (the write may be predicated if you don't need the export). This is used to initialize the buffers in the SX.

There cannot be any serialize bits set OR texture Reads between the EA and the last EM.

Memory exports will be surfaced using a macro extension; here is what needs to happen inside the macro:

The macro needs to create a special constant of the form:

Stream ID constant:

- = Integer that holds BaseAddressInBytes/4 in bits (29:0). Bits 31:30 should be 0b01.
- .y = 2**23

.Х

z = Integer that holds register field data. Note that this data must be organized so that it

always represents a 'valid' floating point number, with the relevant bits in (23 - 0); One way of doing this would be to take the 23 bits and add 2**23.

.w = max index value + 2**23

Output to EXaddress:

.x = Base of array (in low 30 bits)/4

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***



ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE		
9 July, 2003 4 September, 2015		GEN-CXXXXX-REVA	31 of 54		
.y = Index value (in low 23 bits)					

- Index value (in low 23 bits
- .z = Register Field data (in low 23 bits) = Max Index value (in low 23 bits) .w

Also Assume that C0:

.х = 0.0 = 1.0 .у

The Macro expansion would be as follows:

MULADD EA = Rindex.xxxx,C0.xyxx,CstreamID; MOV EMx (x = 0 thru 4) = Rdata;

The SX will check for invalid writes and mask out the data so it won't be written to memory. Invalid writes are:

- 1) Index value >= Max Index value
- 2) bit 31 != 0 (negative index)
- 3) bits [30:23] != 23 + IEEE_EXP_BIAS (127) (meaning the index was too big to be represented using 23 bits)

They cannot have texture instructions interleaved in the export block. These exports are not guaranteed to be ordered.

Also, when doing a pass thru export, the shader must still do either a position and PC export (if Vertex) or a color export (if Pixel). The pass thru export can occur anywhere in any shader program and thus can be used to debug. There can be any number of pass thru export blocks throughout the pixel or vertex shader or both.

17.2 Arbitration restrictions

Here are the Sequencer arbitration restrictions:

- 1) Cannot execute a serialized thread if the corresponding texture pending bit and VC pending is set
- 2) Cannot allocate position if any older thread has not allocated position
- Cannot execute a texture clause if texture reads are pending 3)
- 4) Cannot execute a VC clause if VC reads are pending
- Cannot execute last if texture pending (even if not serial) 5)
- Cannot allocate if not last for color exports. 6)
- 7) Cannot allocate if not last for PC exports.

18. Export Types

The export type (or the location where the data should be put) is specified using the destination address field in the ALU instruction. Here is a list of all possible export modes:

18.1 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

84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © *** Exhibit 2039.doc



(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.)

18.2 Pixel Shading

1

2

- Color for buffer 0 (primary)
- Color for buffer 1
- Color for buffer 2 - Color for buffer 3
- 3 Color for buffer 4:15 - Empty
- 4:15 Empty 16 - Buffer 0 Color/Fog (primary)
- 17 Buffer 1 Color/Fog (prim
- 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 'alpha' component)
- 62:63 Empty

19. Special Interpolation modes

19.1 Real time commands

We are unable to use the parameter memory since there is no way for a command stream to write into it. Instead we need to add three 4x128 memories (one for each of three vertices x 4 interpolants). These will be mapped onto the register bus and written by type 0 packets, and output to the the parameter busses (the sequencer and/or PA need to be able to address the reatime parameter memory as well as the regular parameter store. This mode is triggered by the primitive type: REAL TIME. The actual memories are in the in the SX blocks. The parameter data memories are hooked on the RBBM bus and are loaded by the CP using register mapped memory.

19.2 Sprites/ XY screen coordinates/ FB information

XY screen coordinates may be needed in the shader program. This functionality is controlled by the param_gen register (in SQ) in conjunction with the SND_XY register (in SC) and the param_gen_pos. Also it is possible to send the faceness information (for OGL front/back special operations) to the shader using the same control register. Here is a list of all the modes and how they interact together:

The Data is going to be written in the register specified by the param_gen_pos register.

Param_Gen disable, snd_xy disable = No modification Param_Gen disable, snd_xy enable = No modification Param_Gen enable, snd_xy disable = Sign(faceness)garbage,(Sign Point)garbage,Sign(Line)s, t Param_Gen enable, snd_xy enable = Sign(faceness)screenX,(Sign Point)screenY,Sign(Line)s, t

In other words,

The generated vector is (X in RED, Y in GREEN, S in BLUE and T in ALPHA): X,Y,S,T

PGenReg.X = screen X biased 2^23 (assumes pixel center at 0.0), sign bit encodes faceness (0=frontface, 1=backface)

PGenReg.Y = screen Y biased 2^23 (assumes pixel center at 0.0), sign encodes is point primitive (0=not point, 1=is point)

PGenReg.Z = parametric S coordinate [0..1], sign encodes is line primitive (0=not line, 1=is line)

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
9 July, 2003	4 September, 2015	GEN-CXXXXX-REVA	33 of 54

PGenReg.W = parametric T coordinate [0..1]

Constant C0.X = 2^23 (debias for D3D) C0.Y = $2^23 - 0.5$ (debias for OGL which has pixel centers at 0.5)

To generate useable XY: For D3D: ADD ScreenXYReg.xy__ = abs(PGenReg), -C0.xxxx For OGL ADD ScreenXYReg.xy__ = abs(PGenReg), -C0.yyyy Note abs has to be done on PGenReg

To access faceness. Must ALWAYS use (or pos/neg test against) PGenReg.X. < 0.0 is backface >= 0.0 is frontface

```
To access parametric ST.
Same as before simply take abs before access.
realS = abs(PGenReg.Z)
realT = abs(PGenReg.W)
```

To access primitive type +/-ZERO cannot be differentiated in shader pipe so a RECIP_CLAMPED instruction must be done first before testing isLine. isPoint = PGenReg.Y (if <0.0 then point primitive) isLine = RECIP_CLAMPED PGenReg.Z (if <0.0 then line primitive) if ((isPoint>=0.0) && (isLine>=0.0)) then triangle primitive

19.3 Auto generated counters

In the cases we are dealing with multipass shaders, the sequencer is going to generate a vector count to be able to both use this count to write the 1st pass data to memory and then use the count to retrieve the data on the 2nd pass. The count is always generated in the same way but it is passed to the shader in a slightly different way depending on the shader type (pixel or vertex). This is toggled on and off using the GEN_INDEX_PIX/VTX register. The sequencer is going to keep two counters, one for pixels and one for vertices. Every time a full vector of vertices or pixels is written to the GPRs the counter is incremented. Every time a RST_PIX_COUNT or RST_VTX_COUNT events are received, the corresponding counter is reset. While there is only one count broadcast to the GPRs, the LSB are hardwired to specific values making the index different for all elements in the vector. Since the count must be different for all pixels/vertices and the 4 LSBs (16 positions) are hardwired to the corresponding shader unit the SQ has two choices:

1) Maintain a 17 bit counter that counts the vectors of 64. In this case the phase must be appended to the count before the count is broadcast to the SPs:

Countar (17 hite)	Dhace (2 hite)	Hardwirod (1 bita)
Counter (17 bits)	Phase (2 Dits)	Haldwired (4 bits)

2) Maintain a 21 bits counter that counts sub-vectors of 16. In this case only the counter is sent to the Sps:

	Counter (19 bits)	Hardwired (4 bits)
--	-------------------	--------------------

19.3.1 Vertex shaders

In the case of vertex shaders, if GEN_INDEX_VTX is set, the data will be put into the x field of the third register (it means that the compiler must allocate 3 GPRs in all multipass vertex shader modes).

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
200	9 July, 2003	4 September, 2015		34 of 54

19.3.2 Pixel shaders

In the case of pixel shaders, if GEN_INDEX_PIX is set, the data will be put in the x field of the param_gen_pos+1 register.

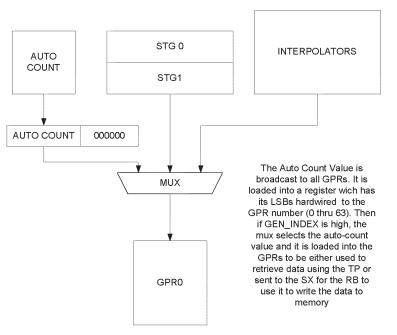


Figure 10: GPR input mux Control

20. State management

Every clock, the sequencer will report to the CP the oldest states still in the pipe. These are the states of the programs as they enter the last ALU clause.

20.1 Parameter cache synchronization

In order for the sequencer not to begin a group of pixels before the associated group of vertices has finished, the sequencer will keep a 6 bit count per state (for a total of 8 counters). These counters are initialized to 0 and every time a vertex shader exports its data TO THE PARAMETER CACHE, the corresponding pointer is incremented. When the SC sends a new vector of pixels with the SC_SQ_new_vector bit asserted, the sequencer will first check if the count is greater than 0 before accepting the transmission (it will in fact accept the transmission but then lower its ready to receive). Then the sequencer waits for the count to go to one and decrements it. The sequencer can then issue the group of pixels to the interpolators. Every time the state changes, the new state counter is initialized to 0.

21. XY Address imports

The SC will be able to send the XY addresses to the GPRs. It does so by interleaving the writes of the IJs (to the IJ buffer) with XY writes (to the XY buffer). Then when writing the data to the GPRs, the sequencer is going to interpolate the IJ data or pass the XY data thru a Fix \rightarrow float converter and expander and write the converted values to the GPRs. The Xys are currently SCREEN SPACE COORDINATES. The values in the XY buffers will wrap. See section 19.2 for details on how to control the interpolation in this mode.

21.1 Vertex indexes imports

In order to import vertex indexes, we have 16 8x96 staging registers. These are loaded one line at a time by the VGT block (96 bits). They are loaded in floating point format and can be transferred in 4 or 8 clocks to the GPRs.

Exhibit 2039.doc 84302 Bytes*** C ATI Confidential. Reference Copyright Notice on Cover Page C ***

AMD1044_0257956

ATI Ex. 2109 IPR2023-00922 Page 246 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
900	9 July, 2003	4 September, 2015	GEN-CXXXXX-REVA	35 of 54	

22. Registers

Please see the auto-generated web pages for register definitions.

23. Interfaces

23.1 External Interfaces

Whenever an x is used, it means that the bus is broadcast to all units of the same name. For example, if a bus is named SQ \rightarrow SPx it means that SQ is going to broadcast the same information to all SP instances.

23.2 SC to SP Interfaces

23.2.1 SC_SP#

There is one of these interfaces at front of each of the SP (buffer to stage pixel interpolators). This interface transmits the I,J data for pixel interpolation. For the entire system, two quads per clock are transferred to the 4 SPs, so each of these 4 interfaces transmits one half of a quad per clock. The interface below describes a half of a quad worth of data.

The actual data which is transferred per quad is

Ref Pix I => S4.20 Floating Point I value *4 Ref Pix J => S4.20 Floating Point J value *4

This equates to a total of 200 bits which transferred over 2 clocks and therefor needs an interface 100 bits wide

Additionally, X,Y data (12-bit unsigned fixed) is conditionally sent across this data bus over the same wires in an additional clock. The X,Y data is sent on the lower 24 bits of the data bus with faceness in the msb. Transfers across these interfaces are synchronized with the SC SQ IJ Control Bus transfers.

The data transfer across each of these busses is controlled by a IJ_BUF_INUSE_COUNT in the SC. Each time the SC has sent a pixel vector's worth of data to the SPs, he will increment the IJ_BUF_INUSE_COUNT count. Prior to sending the next pixel vectors data, he will check to make sure the count is less than MAX_BUFER_MINUS_2, if not the SC will stall until the SQ returns a pipelined pulse to decrement the count when he has scheduled a buffer free. Note: We could/may optimize for the case of only sending only IJ to use all the buffers to pre-load more. Currently it is planned for the SP to hold 2 double buffers of I,J data and two buffers of X,Y data, so if either X,Y or Centers and Centroids are on, then the SC can send two Buffers.

In at least the initial version, the SC shall send 16 quads per pixel vector even if the vector is not full. This will increment buffer write address pointers correctly all the time. (We may revisit this for both the SX,SP,SQ and add a EndOfVector signal on all interfaces to quit early. We opted for the simple mode first with a belief that only the end of packet and multiple new vector signals should cause a partial vector and that this would not really be significant performance hit.)

Name	Bits	Description
SC_SP#_data	100	IJ information sent over 2 clocks (or X,Y in 24 LSBs with faceness in upper bit)
		Type 0 or 1, First clock I, second clk J
		Field ULC URC LLC LRC
		Bits [63:39] [38:26] [25:13] [12:0]
		Format SE4M20 SE4M20 SE4M20 SE4M20
		Туре 2
		Field Face X Y
		Bits [24] [23:12] [11:0]
		Format Bit Unsigned Unsigned
SC_SP#_valid	1	Valid
SC_SP#_last_quad_data	1	This bit will be set on the last transfer of data per quad.

Exhibit 2039.doc 84302 Bytes*** [©] ATI Confidential. Reference Copyright Notice on Cover Page [©] ***

AMD1044_0257957

ATI Ex. 2109 IPR2023-00922 Page 247 of 326

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
	9 July, 2003	4 September, 2015		36 of 54
SC_SP#_type	2	0 -> Indicates centroids 1 -> Indicates centers 2 -> Indicates X,Y Data and faceness on data bus The SC shall look at state data to determine how many types to send for interpolation process.		o send for the

The **#** is included for clarity in the spec and will be replaced with a prefix of u**#**_ in the verilog module statement for the SC and the SP block will have neither because the instantiation will insert the prefix.

23.2.2 SC_SQ

This is the control information sent to the sequencer in order to synchronize and control the interpolation and/or loading data into the GPRs needed to execute a shader program on the sent pixels. This data will be sent over two clocks per transfer with 1 to 16 transfers. Therefore the bus (approx 108 bits) could be folded in half to approx 54 bits.

Name	Bits	Description
Name SC_SQ_data	Bits 46	Control Data sent to the SQ 1 clk transfers Event - valid data consist of event_id and state_id. Instruct SQ to post an event vector to send state id and event_id through request fifo and onto the reservation stations making sure state id and/or event_id gets back to the CP. Events only follow end of packets so no pixel vectors will be in progress. Empty Quad Mask – Transfer Control data consisting of pc_dealloc or new_vector. Receipt of this is to transfer pc_dealloc or new_vector without any valid quad data. New vector will always be posted to request fifo and pc_dealloc will be attached to any pixel vector outstanding or posted in request fifo if no valid quad outstanding. 2 clk transfers Quad Data Valid – Sending quad data with or without new_vector or pc_dealloc. New vector will be posted to request fifo with or without a pixel vector and pc_dealloc will be posted with a pixel vector unless none is in progress. In this case the pc_dealloc will be posted in the request queue. Filler quads will be transferred with
SC_SQ_valid	1	The Quad mask set but the pixel corresponding pixel mask set to zero. SC sending valid data, 2 nd clk could be all zeroes

 $\ensuremath{\mathsf{SC}_\mathsf{SQ}_\mathsf{data}}\xspace$ – first clock and second clock transfers are shown in the table below.

Name	BitField	Bits	Description

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE	DATE	EDIT DATE 4 September, 2015		DOCUMENT-REV. NUM.	PAGE			
	9 July, 20	03			GEN-CXXXXX-REVA	37 of 54			
1 st Clock Trans	sfer					1			
SC_SQ_event		0	1	This transfer is a 1	clock event vector Force quad_mask	=			
					new_vector=pc_dealloc=0				
SC_SQ_event	_id	[5:1]	4	This field identifies the event 0 => denotes an End Of State Event 1 => TBD					
SC_SQ_state_	id	[8:6]	3	State/constant poir	iter (6*3+3)				
SC_SQ_pc_de	ealloc	[11:9]	3	Deallocation token	for the Parameter Cache				
SC_SQ_new_v	vector	12	1		The SQ must wait for Vertex shader done count > 0 and after dispatching the Pixel Vector the SQ will decrement the count.				
SC_SQ_quad		[16:13]	4		Quad Write mask left to right SP0 => SP3				
SC_SQ_end_c		17	1	End Of the primitive					
SC_SQ_pix_m	iask	[33:18]	16	Valid bits for all pix	els SP0=>SP3 (UL,UR,LL,LR)				
SC_SQ_provo		[35:34]	2	Provoking vertex for flat shading					
SC_SQ_lod_c	orrect_0	[44:36]	9	LOD correction for quad 0 (SP0) (9 bits per quad)					
SC_SQ_lod_c	orrect_1	[53:45]	9	LOD correction for quad 1 (SP1) (9 bits per quad)					
2nd Clock Trai	TATA AND DESCRIPTION OF A	peca.manumerco.com							
SC_SQ_lod_c		[8:0]	9	LOD correction for quad 2 (SP2) (9 bits per quad)					
SC_SQ_lod_c		[17:9]	9	LOD correction for quad 3 (SP3) (9 bits per quad)					
SC_SQ_pc_pt		[28:18]	11	Parameter Cache pointer for vertex 0					
SC_SQ_pc_pt		[39:29]	11	Parameter Cache pointer for vertex 1					
SC_SQ_pc_pt		[50:40]	11	Parameter Cache pointer for vertex 2					
SC_SQ_prim_	type	[53:51]	3	Stippled line and R alternate buffer	eal time command need to load tex c	ords from			
				000: Sprite (point)					
				001: Line					
				010: Tri_rect					
				100: Realtime Sprit					
				101: Realtime Line					
				110: Realtime Tri_r	ect				

Name	Bits	Description
SQ_SC_free_buff	1	Pipelined bit that instructs SC to decrement count of buffers in use.
SQ_SC_dec_cntr_cnt	1	Pipelined bit that instructs SC to decrement count of new vector and/or event
		sent to prevent SC from overflowing SQ interpolator/Reservation request fifo.

The scan converter will submit a partial vector whenever:

- 1.) He gets a primitive marked with an end of packet signal.
- 2.) A current pixel vector is being assembled with at least one or more valid quads and the vector has been marked for deallocate when a primitive marked new_vector arrives. The Scan Converter will submit a partial vector (up to 16quads with zero pixel mask to fill out the vector) prior to submitting the new_vector marker\primitive.

(This will prevent a hang which can be demonstrated when all primitives in a packet three vectors are culled except for a one quad primitive that gets marked pc_dealloc (vertices maximum size). In this case two new_vectors are submitted and processed, but then one valid quad with the pc_dealloc creates a vector and then the new would wait for another vertex vector to be processed, but the one being waited for could never export until the pc_dealloc signal made it through and thus the hang.)

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***



Name	Direction	Bits	Description
SQ_SXx_interp_cyl_wrap	SQ→SXx	4	Which channel needs to be cylindrical wrapped
SQ_SXx_wrap_count	SQ->SXx	2	Cylindrical wrap count
SQ_SPx_auto_count	SQ->SPx	19	Auto generated count for VTx and Pixels
SQ_SPx_interp_param_gen	SQ→SPx	1	Generate Parameter
SQ_SPx_interp_prim_type	SQ→SPx	2	Bits [1:0] of primitive type sent by SC
SQ_SPx_interp_buff_swap	SQ→SPx	1	Swap IJ buffers
SQ_SPx_interp_IJ_line	SQ→SPx	2	IJ line number
SQ_SPx_interp_mode	SQ→SPx	1	Center/Centroid sampling
SQ_SXx_pc_ptr0	SQ→SXx	11	Parameter Cache Pointer
SQ_SXx_pc_ptr1	SQ→SXx	11	Parameter Cache Pointer
SQ_SXx_pc_ptr2	SQ→SXx	11	Parameter Cache Pointer
SQ_SXx_rt_sel	SQ→SXx	1	Selects between RT and Normal data (Bit 2 of prim type)
SQ_SX0_pc_wr_en	SQ→SX0	8	Write enable for the PC memories
SQ_SX1_pc_wr_en	SQ→SX1	8	Write enable for the PC memories
SQ_SXx_pc_wr_addr	SQ→SXx	7	Write address for the PCs
SQ_SXx_pc_channel_mask	SQ→SXx	4	Channel mask
SQ_SXx_pc_ptr_valid	SQ→SXx	1	Read pointers are valid.
SQ_SPx_interp_valid	SQ→SPx	1	Interpolation control valid

23.2.4 SQ to SP: Staging Register Data

This is a broadcast bus that sends the VSISR information to the staging registers of the shader pipes.

Name	Direction	Bits	Description
SQ_SPx_vsr_data	SQ→SPx	96	Pointers of indexes or HOS surface information
SQ_SPx_vsr_wrt_addr	SQ→SPx	3	Staging register write address
SQ_SPx_vsr_rd_addr	SQ→SPx	3	Staging register read address
SQ_SP0_vsr_valid	SQ→SP0	1	Data is valid
SQ_SP1_vsr_valid	SQ→SP1	1	Data is valid
SQ_SP2_vsr_valid	SQ→SP2	1	Data is valid
SQ_SP3_ vsr_ valid	SQ→SP3	1	Data is valid
SQ_SPx_vsr_read	SQ→SPx	1	Increment the read pointers

23.2.5 VGT to SQ : Vertex interface

23.2.5.1 Interface Signal Table

The area difference between the two methods is not sufficient to warrant complicating the interface or the state requirements of the VSISRs. <u>Therefore, the POR for this interface is that the VGT will transmit the data to the</u> <u>VSISRs (via the Shader Sequencer) in full, 32-bit floating-point format.</u> The VGT can transmit up to six 32-bit floating-point values to each VSISR where four or more values require two transmission clocks. The data bus is 96 bits wide. In the case where an event is sent the 5 LSBs of VGT_SQ_vsisr_data contain the eventID.

Name	Bits	Description					
VGT_SQ_vsisr_data	96	Pointers of indexes or HOS surface information					
VGT_SQ_event	1	VGT is sending an event					
VGT_SQ_vsisr_continued	1	0: Normal 96 bits per vert 1: double 192 bits per vert					
VGT_SQ_end_of_vtx_vect	1	Indicates the last VSISR data set for the current process vector (for double vector					
		data, "end_of_vector" is set on the first vector)					
VGT_SQ_indx_valid	1	Vsisr data is valid					
VGT_SQ_state	3	Render State (6*3+3 for constants). This signal is guaranteed to be correct when "VGT_SQ_vgt_end_of_vector" is high.					
VGT_SQ_send	1	Data on the VGT_SQ is valid receive (see write-up for standard R400 SEND/RTR interface handshaking)					
SQ_VGT_rtr	1	Ready to receive (see write-up for standard R400 SEND/RTR interface handshaking)					

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
9 July, 2003	4 September, 2015	GEN-CXXXXX-REVA	39 of 54	

23.2.5.2 Interface Diagrams

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257961

ATI Ex. 2109 IPR2023-00922 Page 251 of 326

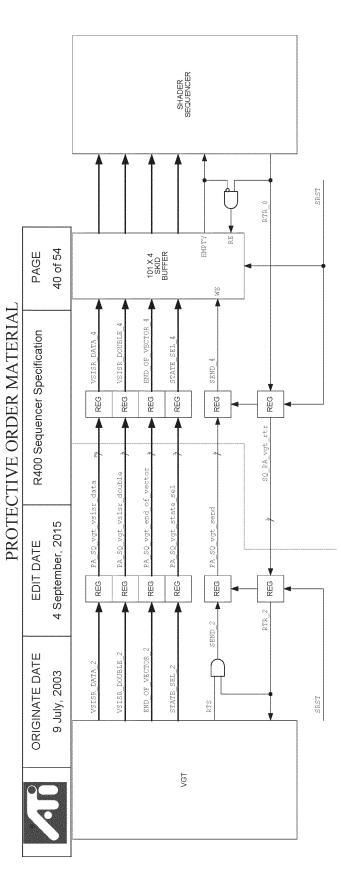


Exhibit 2039.doc 84502 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257962

ATI Ex. 2109 IPR2023-00922 Page 252 of 326

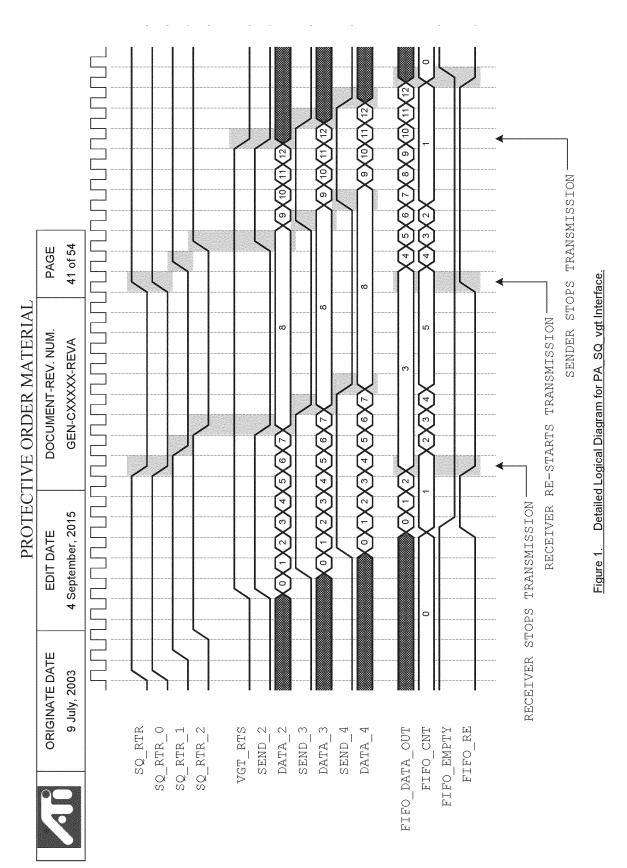


Exhibit 2039.doc 84.302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***



23.2.6 SQ to SX: Control bus

Name	Direction	Bits	Description
SQ_SXx_exp_type	SQ→SXx	2	00: Pixel without z (1 to 4 buffers) 01: Pixel with z (1 to 4 buffers) 10: Position (1 or 2 results) 11: Pass thru (1 to 5 results aligned)
SQ_SXx_exp_number	SQ→SXx	2	Number of locations needed in the export buffer (encoding depends on the type see bellow).
SQ_SXx_exp_alu_id	SQ→SXx	4	ALU ID. Revolving ID 0 thru 15. Memory exports have to increment this count by 4 or 8 depending on the size requested. Other type of exports increment the ID by 1.
SQ_SXx_exp_valid	SQ→SXx	1	Valid bit
SQ_SXx_exp_state	SQ→SXx	3	State Context
SQ_SXx_free_done	SQ→SXx	1	Pulse that indicates that the previous export is finished from the point of view of the SP. This does not necessarily mean that the data has been transferred to RB or PA, or that the space in export buffer for that particular vector thread has been freed up.
SQ_SXx_free_alu_id	SQ→SXx	4	ALU ID that was used at allocate time.

Depending on the type the number of export location changes:

- Type 00 : Pixels without Z
 - 00 = 1 buffer
 - \circ 01 = 2 buffers
 - \circ 10 = 3 buffers
 - 11 = 4 buffer
 - Type 01: Pixels with Z
 - \circ 00 = 2 Buffers (color + Z)
 - \circ 01 = 3 buffers (2 color + Z)
 - \circ 10 = 4 buffers (3 color + Z)
 - \circ 11 = 5 buffers (4 color + Z)
- Type 10 : Position export
- \circ 00 = 1 position
 - \circ 01 = 2 positions
 - o 1X = Undefined
- Type 11: Pass Thru
 - \circ 00 = 4 buffers
 - \circ 01 = 8 buffers
 - 10 = Undefined
 - 11 = Undefined

23.2.7 SX to SQ : Output file control

Below the thick black line is the end of transfer packet that tells the SX that a given export is finished. The report packet will always arrive either before or at the same time than the next export to the same ALU id.

Name	Direction	Bits	Description
SXx_SQ_pix_free_count0	SXx→SQ	6	How many slots
OVv. CO mily asymptot walled	6VV 60	4	E

Name	Direction	Bits	Description
SXx_SQ_pix_free_count0	SXx→SQ	6	How many slots where just freed in the SX for bank0
SXx_SQ_pix_count0_valid	SXx→SQ	1	Free_count0 is valid
SXx_SQ_pix_free_count1	SXx→SQ	6	How many slots where just freed in the SX for bank1
SXx_SQ_pix_count1_valid	SXx→SQ	1	Free_count1 is valid
SXx_SQ_pos_free_count0	SXx→SQ	4	How many slots where just freed in the SX for bank0
SXx_SQ_pos_count0_valid	SXx→SQ	1	Free_count0 is valid
SXx_SQ_pos_free_count1	SXx→SQ	4	How many slots where just freed in the SX for bank1

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE	ORIGINATE DATE		r dat	E	DOCUMENT-REV. NUM.	PAGE
	9 July, 20	03	4 Septer	4 September, 2015		GEN-CXXXXX-REVA	43 of 54
SXx_SQ_pos_	_count1_valid	SXx→SQ		1	Free_c	count1 is valid	
SXx_SQ_men	n_export_free	SXx_→SQ		1	Freed	a memory export slot	

23.2.8 SQ to TP: Control bus

Once every clock, the fetch unit sends to the sequencer on which RS line it is now working and if the data in the GPRs is ready or not. This way the sequencer can update the fetch valid bits flags for the reservation station. The sequencer also provides the instruction and constants for the fetch to execute and the address in the register file where to write the fetch return data.

Name	Direction	Bits	Description
TPx_SQ_data_rdy	$TPx \rightarrow SQ$	1	Data ready
TPx_SQ_rs_line_num	$TPx \rightarrow SQ$	6	Line number in the Reservation station
TPx_SQ_type	$TPx \rightarrow SQ$	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_TPx_send	SQ→TPx	1	Sending valid data
SQ_TPx_const	SQ→TPx	48	Fetch state sent over 4 clocks (192 bits total)
SQ_TPx_instr	SQ→TPx	24	Fetch instruction sent over 4 clocks
SQ_TPx_end_of_group	SQ→TPx	1	Last instruction of the group
SQ_TPx_Type	SQ→TPx	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_TPx_gpr_phase	SQ→TPx	2	Write phase signal
SQ_TP0_lod_correct	SQ→TP0	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP0_pix_mask	SQ→TP0	4	Pixel mask 1 bit per pixel
SQ_TP1_lod_correct	SQ→TP1	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP1_pix_mask	SQ→TP1	4	Pixel mask 1 bit per pixel
SQ_TP2_lod_correct	SQ→TP2	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP2_pix_mask	SQ→TP2	4	Pixel mask 1 bit per pixel
SQ_TP3_lod_correct	SQ→TP3	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP3_pix_mask	SQ→TP3	4	Pixel mask 1 bit per pixel
SQ_TPx_rs_line_num	SQ→TPx	6	Line number in the Reservation station
SQ_TPx_write_gpr_index	SQ->TPx	7	Index into Register file for write of returned Fetch Data
SQ_TPx_ctx_id	SQ→TPx	3	The state context ID (needed for multisample resolves)
SQ_TPx_SIMD	SQ->TPx	1	Tells the TP from which SIMD the data is coming from.

23.2.9 SQ to VC: Control bus

Once every clock, the VC unit sends to the sequencer on which RS line it is now working and if the data in the GPRs is ready or not. This way the sequencer can update the fetch valid bits flags for the reservation station. The sequencer also provides the instruction and constants for the fetch to execute and the address in the register file where to write the fetch return data.

Name	Direction	Bits	Description
VCx_SQ_data_rdy	VCx→ SQ	1	Data ready
VCx_SQ_rs_line_num	$VCx \rightarrow SQ$	6	Line number in the Reservation station
VCx_SQ_type	VCx→ SQ	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_VCx_send	SQ→VCx	1	Sending valid data
SQ_VCx_const	SQ→VCx	48	Fetch state sent over 4 clocks (192 bits total)
SQ_VCx_instr	SQ→VCx	24	Fetch instruction sent over 4 clocks
SQ_VCx_end_of_group	SQ→VCx	1	Last instruction of the group
SQ_VCx_Type	SQ→VCx	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_VCx_gpr_phase	SQ→VCx	2	Write phase signal
SQ VC0 pix mask	SQ→VC0	4	Pixel mask 1 bit per pixel
SQ VC1 pix mask	SQ→VC1	4	Pixel mask 1 bit per pixel
SQ_VC2_pix_mask	SQ→VC2	4	Pixel mask 1 bit per pixel
SQ_VC3_pix_mask	SQ→VC3	4	Pixel mask 1 bit per pixel
SQ_VCx_rs_line_num	SQ→VCx	6	Line number in the Reservation station

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE D	ATE	EDIT DATE			R400 Sequencer Specification	PAGE
	9 July, 200	3	4 Septemb	er, 2015			44 of 54
 SQ_VCx_write	e_gpr_index	SQ->V	Cx	7 1	nd	ex into Register file for write of returne	d Fetch Data
SQ_VCx_SIM	D	SQ->V	′Cx	1 -	ſell	s the VC from which SIMD the data is	coming from.

23.2.10 TP to SQ: Texture stall

The TP sends this signal to the SQ and the SPs when frees up a buffer.

Name	Direction	Bits	Description
TP_SQ_fetch_dec	$TP \rightarrow SQ$	1	Just freed a slot in the TP.

23.2.11 VC to SQ: Vertex Cache stall

The VC sends this signal to the SQ and the SPs when frees up a buffer. There are 2 types of buffers, Mega and Mini and a signal for both.

Name	Direction	Bits	Description
VC_SQ_fetch_dec_mega	$VC \rightarrow SQ$	1	Freed a Mega slot in the VC.
VC_SQ_fetch_dec_mini	VC→ SQ	1	Freed a Mini slot in the VC.

23.2.12 SQ to SP: GPR and auto counter

I

Name	Direction	Bits	Description
SQ_SPx_simd0_gpr_wr_addr	SQ→SPx	7	Write address
SQ_SPx_simd0_gpr_rd_addr	SQ→SPx	7	Read address
SQ_SPx_simd0_gpr_rd_en	SQ→SPx	1	Read Enable
SQ_SP0_simd0_gpr_pspv_wr_en	SQ→SP0 (SP1)	4	Write Enable for the GPRs of SP0-1 for PS and PV
SQ_SP2_simd0_gpr_pspv_wr_en	SQ→SP2 (SP3)	4	Write Enable for the GPRs of SP2-3 for PS and PV
SQ_SP4_simd0_gpr_pspv_wr_en	SQ→SP4 (SP5)	4	Write Enable for the GPRs of SP4-5 for PS and PV
SQ_SP6_simd0_gpr_pspv_wr_en	SQ→SP6 (SP7)	4	Write Enable for the GPRs of SP6-7 for PS and PV
SQ_SP0_simd0_gpr_int_wr_en	SQ→SP0	1	Write Enable for the GPRs of SP0 for Inputs (interp/vtx)
SQ_SP2_simd0_gpr_int_wr_en	SQ→SP2	1	Write Enable for the GPRs of SP1 for Inputs (interp/vtx)
SQ_SP4_simd0_gpr_int_wr_en	SQ→SP4	1	Write Enable for the GPRs of SP2 for Inputs (interp/vtx)
SQ_SP6_simd0_gpr_int_wr_en	SQ→SP6	1	Write Enable for the GPRs of SP3 for Inputs (interp/vtx)
SQ_SPx_gpr_phase	SQ→SPx	2	The phase mux (arbitrates between inputs, ALU SRC reads and writes)
SQ_SPx_simd0_channel_mask	SQ→SPx	4	The channel mask for SIMD0
SQ_SPx_gpr_input_sel	SQ→SPx	2	When the phase mux selects the inputs this tells from which source to read from: Interpolated data, VSR, autogen counter.
SQ_SPx_auto_count	SQ→SPx	21	Auto count generated by the SQ, common for all shader pipes
SQ_SPx_simd0_fetch_swizzle	SQ <u>→</u> SPx	6	Swizzle code for the TP request (2 bits per channel ignore W as it is not used). Bits [10] X mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W Bits [32] Y mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W Bits [54] Z mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W
SQ_SPx_tp_fetch_simd_sel	SQ→SPx	1	TP Resource coming from: 0: SIMD0 1: SIMD1
SQ_SPx_vc_fetch_simd_sel	SQ _{→→} SPx	1	VC Resource coming from: 0: SIMD0 1: SIMD1
SQ_SPx_simd1_gpr_wr_addr	SQ→SPx	7	Write address

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257966

ATI Ex. 2109 IPR2023-00922 Page 256 of 326

		ORIGINATE DATE		EDIT DA	TE		DOCUMENT-REV. NUM.	PAGE
	<u>°UU</u>	9 July, 2003		4 September	er, 2015		GEN-CXXXXX-REVA 4	5 of 54
3	SQ_SPx_s	simd1_gpr_rd_addr	SQ-	→SPx	7	Re	ad address	
		simd1_gpr_rd_en	SQ-	→SPx	1	Re	ad Enable	
\$	SQ_SP0_s	simd1_gpr_pspv_wr_en	SQ-	→SP0 (SP1)	4	Wr	ite Enable for the GPRs of SP0-1 for PS ar	d PV
3	SQ_SP2_s	simd1_gpr_pspv_wr_en	SQ-	→SP2 (SP3)	4	Wr	ite Enable for the GPRs of SP2-3 for PS ar	d PV
		simd1_gpr_pspv_wr_en		→SP4 (SP5)	4	Wr	ite Enable for the GPRs of SP4-5 for PS ar	d PV
\$	SQ_SP6_s	simd1_gpr_pspv_wr_en	SQ-	→SP6 (SP7)	4	Wr	ite Enable for the GPRs of SP6-7 for PS ar	d PV
3	SQ_SPx_	simd1_channel_mask	SQ-	→SPx	4	Th	e channel mask for SIMD1	
\$	SQ_SPx_s	simd1_fetch_swizzle	SQ-	→SPx	6	Sw	vizzle code for the TP request (2 bits per o	hannel
							ore W as it is not used).	
							s [10] X mode select:	
						0=	GPR_X 1=GPR_Y 2=GPR_Z 3=GP	R_W
						Bit	s [32] Y mode select:	
						0=	GPR_X 1=GPR_Y 2=GPR_Z 3=GP	R_W
						Bit	s [54] Z mode select:	
						0=	GPR_X 1=GPR_Y 2=GPR_Z 3=GP	R_W
	SQ SPO s	simd1_gpr_int_wr_en	SQ-	→SP0	1	Wr	ite Enable for the GPRs of SP0-1 for	Inputs
						(in	terp/vtx)	
	SQ_SP2_s	simd1_gpr_int_wr_en	SQ-	→SP2	1	Wr	ite Enable for the GPRs of SP2-3 for	Inputs
						(in	terp/vtx)	
	SQ_SP4_s	simd1_gpr_int_wr_en	SQ-	→SP4	1	Wr	ite Enable for the GPRs of SP4-5 for	Inputs
						(in	terp/vtx)	
3	SQ_SP6_s	simd1_gpr_int_wr_en	SQ-	→SP6	1	Wr	ite Enable for the GPRs of SP6-7 for	Inputs
	_					(in	terp/vtx)	

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257967

ATI Ex. 2109 IPR2023-00922 Page 257 of 326

1000	6
	a A

23.2.13 SQ to SPx:

Name	Direction	Bits	Description					
SQ_SPx_instr_start	SQ→SPx	1	Instruction start Transferred over 4 cycles					
SQ_SPx_simd0_instruct	SQ→SPx	24						
			0: SRC A Negate Argument Modifier 0:0					
			SRC A Abs Argument Modifier 1:1					
			SRC A Swizzle 9:2					
			Vector Dst 15:10					
			Per channel Select 23:16					
			00: GPR					
			01: PV					
			10: PS					
			11: Constant (if 11 has to be 11 for					
			channels)					
			·					
			- 1: SRC B Negate Argument Modifier 0:0					
			SRC B Abs Argument Modifier 1:1					
			SRC B Swizzle 9:2					
			Scalar Dst 15:10					
			Per channel Select 23:16					
			00: GPR					
			01: PV					
			10: PS					
			11: Constant (if 11 has to be 11 for					
			channels)					
			-					
			2: SRC C Negate Argument Modifier 0:0					
			SRC C Abs Argument Modifier 1:1					
			SRC C Swizzle 9:2					
			Unused 15:10					
			Per channel Select 23:16					
			00: GPR					
			01: PV					
			10: PS					
			11: Constant (if 11 has to be 11 for					
			channels)					
			-					
			3: Vector Opcode 4:0					
	1	1						
			Scalar Opcode 10:5					
			Scalar Opcode 10:5 Vector Clamp 11:11					
			Scalar Opcode10:5Vector Clamp11:11Scalar Clamp12:12					
			Scalar Opcode10:5Vector Clamp11:11Scalar Clamp12:12					
			Scalar Opcode10:5Vector Clamp11:11Scalar Clamp12:12Vector Write Mask16:13					
			Scalar Opcode10:5Vector Clamp11:11Scalar Clamp12:12Vector Write Mask16:13Scalar Write Mask20:17					
3Q SP0 simd0 pred override	SQ→SP0 (SP1)	4	Scalar Opcode10:5Vector Clamp11:11Scalar Clamp12:12Vector Write Mask16:13Scalar Write Mask20:17Unused23:21					
SQ_SP0_simd0_pred_override	SQ→SP0 (SP1)	4	Scalar Opcode10:5Vector Clamp11:11Scalar Clamp12:12Vector Write Mask16:13Scalar Write Mask20:17Unused23:21SP0 receives 2 bits and SP1 two bits as well.					
SQ_SP0_simd0_pred_override	SQ→SP0 (SP1)	4	Scalar Opcode10:5Vector Clamp11:11Scalar Clamp12:12Vector Write Mask16:13Scalar Write Mask20:17Unused23:21SP0 receives 2 bits and SP1 two bits as well.0: Use per channel RGBA field (enables the per chan					
SQ_SP0_simd0_pred_override	SQ→SP0 (SP1)	4	Scalar Opcode10:5Vector Clamp11:11Scalar Clamp12:12Vector Write Mask16:13Scalar Write Mask20:17Unused23:21SP0 receives 2 bits and SP1 two bits as well.0: Use per channel RGBA field (enables the per chanlogic).					
SQ_SP0_simd0_pred_override	SQ→SP0 (SP1)	4	Scalar Opcode 10:5 Vector Clamp 11:11 Scalar Clamp 12:12 Vector Write Mask 16:13 Scalar Write Mask 20:17 Unused 23:21 SP0 receives 2 bits and SP1 two bits as well. 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the					
			Scalar Opcode 10:5 Vector Clamp 11:11 Scalar Clamp 12:12 Vector Write Mask 16:13 Scalar Write Mask 20:17 Unused 23:21 SP0 receives 2 bits and SP1 two bits as well. 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged					
		4	Scalar Opcode 10:5 Vector Clamp 11:11 Scalar Clamp 12:12 Vector Write Mask 16:13 Scalar Write Mask 20:17 Unused 23:21 SP0 receives 2 bits and SP1 two bits as well. 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the					
			Scalar Opcode 10:5 Vector Clamp 11:11 Scalar Clamp 12:12 Vector Write Mask 16:13 Scalar Write Mask 20:17 Unused 23:21 SP0 receives 2 bits and SP1 two bits as well. 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan					
			Scalar Opcode 10:5 Vector Clamp 11:11 Scalar Clamp 12:12 Vector Write Mask 16:13 Scalar Write Mask 20:17 Unused 23:21 SP0 receives 2 bits and SP1 two bits as well. 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic).					
			Scalar Opcode 10:5 Vector Clamp 11:11 Scalar Clamp 12:12 Vector Write Mask 16:13 Scalar Write Mask 20:17 Unused 23:21 SP0 receives 2 bits and SP1 two bits as well. 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the					
SQ_SP2_simd0_pred_override	SQ→SP2 (SP3)	4	Scalar Opcode 10:5 Vector Clamp 11:11 Scalar Clamp 12:12 Vector Write Mask 16:13 Scalar Write Mask 20:17 Unused 23:21 SP0 receives 2 bits and SP1 two bits as well. 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use GPR for PV or PS settings. LET the per chan logic). 1: Use GPR for PV or PS settings. LET the per chan logic).					
SQ_SP2_simd0_pred_override	SQ→SP2 (SP3)		Scalar Opcode 10:5 Vector Clamp 11:11 Scalar Clamp 12:12 Vector Write Mask 16:13 Scalar Write Mask 20:17 Unused 23:21 SP0 receives 2 bits and SP1 two bits as well. 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic).					
SQ_SP2_simd0_pred_override	SQ→SP2 (SP3)	4	Scalar Opcode 10:5 Vector Clamp 11:11 Scalar Clamp 12:12 Vector Write Mask 16:13 Scalar Write Mask 20:17 Unused 23:21 SP0 receives 2 bits and SP1 two bits as well. 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic).					
SQ_SP0_simd0_pred_override SQ_SP2_simd0_pred_override SQ_SP4_simd0_pred_override	SQ→SP2 (SP3)	4	Scalar Opcode 10:5 Vector Clamp 11:11 Scalar Clamp 12:12 Vector Write Mask 16:13 Scalar Write Mask 20:17 Unused 23:21 SP0 receives 2 bits and SP1 two bits as well. 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic). 1: Use GPR for PV or PS settings. LET the (constant) go thru unchanged 0: Use per channel RGBA field (enables the per chan logic).					

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

ORIGINATE DATE		E	EDIT	DATE		DOCUMENT-REV. NUM.			
<u> 200</u>	9 July, 2003		4 September, 201			015 GEN-CXXXXX-REVA 47 of 5			
SQ_SPx_s	simd0_pred_override simd0_stall simd1_instruct	SQ-	→SP6 (SP7) →SPx →SPx	4	logic) 1: U (cons Stall	e per channel RGBA field (enables the se GPR for PV or PS settings. L .tant) go thru unchanged signal sferred over 4 cycles			
	Sind I_IIStruct	54		24	0: SR SR Ve Pe chant - 1: SR SR SR SC	C A Negate Argument Modifier 0:0 C A Abs Argument Modifier 1:1 C A Swizzle 9:2 ctor Dst 15:10 er channel Select 23:16 00: GPR 01: PV 10: PS 11: Constant (if 11 has to	be 11 for all		
					SR SR Un	C C Negate Argument Modifier 0:0 C C Abs Argument Modifier 1:1 C C Swizzle 9:2 used 15:10 er channel Select 23:16 00: GPR 01: PV 10: PS 11: Constant (if 11 has to			
					- 3: Ve Sc Ve Sc Ve Sc	ctor Opcode 4:0 alar Opcode 10:5 ctor Clamp 11:11 alar Clamp 12:12 ctor Write Mask 16:13 alar Write Mask 20:17 used 23:21			
SQ_SP0_s	simd0_pred_override	SQ-	→SP0 (SP1)	4	SP0 0: Us logic) 1: U (cons	receives 2 bits and SP1 two bits as well e per channel RGBA field (enables the se GPR for PV or PS settings. L .tant) go thru unchanged	per channel ET the 11		
SQ_SP2_s	simd0_pred_override	SQ-	→SP2 (SP3)	4	0: Us logic) 1: U	e per channel RGBA field (enables the			
SQ_SP4_s	simd0_pred_override	SQ-	→SP4 (SP5)	4	0: Us logic) 1: U	e per channel RGBA field (enables the			

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

48	ORIGINATE DATE EDIT D		DATE	R400 Sequencer Specification PAC	€ E
9 July, 2003 4 September		oer, 2015	48 of	54	
SQ_SP6_s	simd0_pred_override	SQ→SP6 (SP7)	4	0: Use per channel RGBA field (enables the per chalogic).1: Use GPR for PV or PS settings. LET th (constant) go thru unchanged	
SQ_SPx_s	imd1_stall	SQ→SPx	1	Stall signal	
SQ SPX 6	export simd sel	SQ->SPx	1	Which SIMD engine is exporting.	

23.2.14 SQ to SX: write mask interface (must be aligned with the SP data)

Name	Direction	Bits	Description
SQ_SX0_write_mask	SQ→SX0	8	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. This is for the data coming of SP0 and SP2.
SQ_SX1_ write_mask	SQ→SX1	8	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. This is for the data coming of SP1 and SP3.
SQ_SXx_channel_mask	SQ->SXx	4	This is the per channel export mask. It is computed by doing vector_mask scalar_mask bit 14 of the alu instruction.
SQ_SX0_kill_mask	SQ->SX0	8	These are the valid bits coming straight from the reservation stations.
SQ_SX1_kill_mask	SQ->SX1	8	These are the valid bits coming straight from the reservation stations.

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257970

ATI Ex. 2109 IPR2023-00922 Page 260 of 326



EDIT DATE

23.2.15 SP to SQ: Constant address load/ Predicate Set/Kill set

Name	Direction	Bits	Description
SP0_SQ_simd0_const_addr	(SP1) SP0→SQ	36	Constant address load 18 bits from SP0 and 18 from SP4.
SP0 SQ simd0 valid	SP0→SQ	1	Data valid
SP2 SQ simd0 const addr	(SP3) SP2→SQ	36	Constant address load
SP2_SQ_simd0_valid	SP2→SQ	1	Data valid
SP4 SQ simd0 const addr	(SP5) SP4→SQ	36	Constant address load
SP4 SQ simd0 valid	SP4→SQ	1	Data valid
SP6 SQ simd0 const addr	(SP7) SP6→SQ	36	Constant address load
SP6_SQ_simd0_valid	SP6→SQ	1	Data valid
SP0_SQ_simd0_pred_kill_vector	(SP1) SP0→SQ	4	Data (predicates or kill/mask) 2 bits from SP0 and 2 bits from SP4
SP0_SQ_simd0_pred_kill_valid	SP0->SQ	1	Data valid
SP0_SQ_simd0_pred_kill_type	SP0->SQ	1	0: predicate vector 1: kill/mask vector
SP2 SQ simd0 pred kill vector	(SP3) SP2→SQ	4	Data (predicates or kill/mask)
SP2_SQ_simd0_pred_kill_valid	SP2->SQ	1	Data valid
SP2_SQ_simd0_pred_kill_type	SP2->SQ	1	0: predicate vector 1: kill/mask vector
SP4_SQ_simd0_pred_kill_vector	(SP5) SP4→SQ	4	Data (predicates or kill/mask)
SP4_SQ_simd0_pred_kill_valid	SP4->SQ	1	Data valid
SP4_SQ_simd0_pred_kill_type	SP4->SQ	1	0: predicate vector
			1: kill/mask vector
SP6 SQ simd0 pred kill vector	(SP7) SP6→SQ	4	Data (predicates or kill/mask)
SP6_SQ_simd0_pred_kill_valid	SP6->SQ	1	Data valid
SP6_SQ_simd0_pred_kill_type	SP6->SQ	1	0: predicate vector 1: kill/mask vector
SP0_SQ_simd1_const_addr	(SP1) SP0→SQ	36	Constant address load 18 bits from SP0 and 18 from SP4.
SP0 SQ simd1 valid	SP0→SQ	1	Data valid
SP2_SQ_simd1_const_addr	(SP3) SP2→SQ	36	Constant address load
SP2 SQ simd1 valid	SP2→SQ	1	Data valid
SP4_SQ_simd1_const_addr	(SP5) SP4→SQ	36	Constant address load
SP4_SQ_simd1_valid	SP4→SQ	1	Data valid
SP6_SQ_simd1_const_addr	(SP7) SP6→SQ	36	Constant address load
SP6 SQ simd1 valid	SP6→SQ	1	Data valid
SP0_SQ_simd1_pred_kill_vector	(SP1) SP0→SQ	4	Data (predicates or kill/mask) 2 bits from SP0 and 2 bits from SP4
SP0_SQ_simd1_pred_kill_valid	SP0->SQ	1	Data valid
SP0_SQ_simd1_pred_kill_type	SP0->SQ	1	0: predicate vector 1: kill/mask vector
SP2_SQ_simd1_pred_kill_vector	(SP3) SP2→SQ	4	Data (predicates or kill/mask)
SP2 SQ simd1 pred kill valid	SP2->SQ	1	Data valid
SP2_SQ_simd1_pred_kill_type	SP2->SQ	1	0: predicate vector 1: kill/mask vector
SP4 SQ simd1 pred kill vector	(SP5) SP4→SQ	4	Data (predicates or kill/mask)
SP4 SQ simd1 pred kill valid	SP4->SQ	1	Data valid
SP4_SQ_simd1_pred_kill_type	SP4->SQ	1	0: predicate vector 1: kill/mask vector
SP6_SQ_simd1_pred_kill_vector	(SP7) SP6→SQ	4	Data (predicates or kill/mask)
SP6_SQ_simd1_pred_kill_vector	SP6->SQ	1	Data valid
SP6_SQ_simd1_pred_kill_type	SP6->SQ	1	0: predicate vector
	0.0.004		1: kill/mask vector

Because of the sharing of the bus none of the MOVA, PREDSET or KILL instructions may be coissued.

23.2.16 SQ to SPx: constant broadcast

Name	Direction	Bits	Description
	•		

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

	ORIGINATE D	ATE	EDIT DATE			R400 Sequencer Specification	PAGE
	9 July, 200	3	4 Septemb	er, 201	5		50 of 54
	simd0_const	SQ→S		128		nstant broadcast	
SQ_SPx_s	simd1_const	SQ→S	Px	128	Cor	nstant broadcast	
SQ_SPx_s	simd0_const_sel	SQ→S		2	for 1 0 : 1 1: V 2: V	e the incoming constant instead of the the next group of 16. Normal mode Vaterfall on SRCA Vaterfall on SRCB Vaterfall on SRCC	registered one
SQ_SPx_s	SQ_SPx_simd1_const_sel SQ→SPx		2	for 1 0 : 1 1: V 2: V	e the incoming constant instead of the the next group of 16. Normal mode Vaterfall on SRCA Vaterfall on SRCB Vaterfall on SRCC	registered one	

23.2.17 SQ to CP: RBBM bus

Name	Direction	Bits	Description
SQ_RBB_rs	SQ→CP	1	Read Strobe
SQ_RBB_rd	SQ→CP	32	Read Data
SQ_RBBM_nrtrtr	SQ→CP	1	Optional
SQ_RBBM_rtr	SQ→CP	1	Real-Time (Optional)

23.2.18 CP to SQ: RBBM bus

Name	Direction	Bits	Description
rbbm_we	CP→SQ	1	Write Enable
rbbm_a	CP→SQ	15	Address Upper Extent is TBD (16:2)
rbbm_wd	CP→SQ	32	Data
rbbm_be	CP→SQ	4	Byte Enables
rbbm_re	CP→SQ	1	Read Enable
rbb_rs0	CP→SQ	1	Read Return Strobe 0
rbb_rs1	CP→SQ	1	Read Return Strobe 1
rbb_rd0	CP→SQ	32	Read Data 0
rbb_rd1	CP→SQ	32	Read Data 0
RBBM_SQ_soft_reset	CP→SQ	1	Soft Reset

23.2.19 SQ to CP: State report

Name	Direction	Bits	Description
SQ_CP_vs_event	SQ→CP	1	Vertex Shader Event
SQ_CP_vs_eventid	SQ→CP	5	Vertex Shader Event ID
SQ_CP_ps_event	SQ→CP	1	Pixel Shader Event
SQ_CP_ps_eventid	SQ→CP	5	Pixel Shader Event ID

23.3 Example of control flow program execution

We now provide some examples of execution to better illustrate the new design.

Given the program:

Alu 0 Alu 1 Tex 0 Tex 1 Alu 3 Serial Alu 4 Tex 2 Alu 5 Alu 6 Serial

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***



Tex 3 Alu 7 Alloc Position 1 buffer Alu 8 Export Tex 4 Alloc Parameter 3 buffers Alu 9 Export 0 Tex 5 Alu 10 Serial Export 2 Alu 11 Export 1 End

Would be converted into the following CF instructions:

Execute 0 Alu 0 Alu 0 Tex 0 Tex 1 Alu 0 Alu 0 Tex 0 Alu 1 Alu 0 Tex Execute 0 Alu Alloc Position 1 Execute 0 Alu 0 Tex Alloc Param 3 Execute_end 0 Alu 0 Tex 1 Alu 0 Alu

And the execution of this program would look like this:

Put thread in Vertex RS:

Control Flow Instruction Pointer (12 bits), (CFP) Execution Count Marker (3 or 4 bits), (ECM) Loop Iterators (4x9 bits), (LI) Call return pointers (4x12 bits), (CRP) Predicate Bits(4x64 bits), (PB) Export ID (1 bit), (EXID) GPR Base Ptr (8 bits), (GPR) Export Base Ptr (7 bits), (EB) Context Ptr (3 bits).(CPTR) LOD correction bits (16x6 bits) (LOD)

State Bits										
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD	
0	0	0	0	0	0	0	0	0	0	

Valid Thread (VALID) Texture/ALU engine needed (TYPE) Texture Reads are outstanding (PENDING) Waiting on Texture Read to Complete (SERIAL) Allocation Wait (2 bits) (ALLOC) 00 – No allocation needed 01 – Position export allocation needed (ordered export) 10 – Parameter or pixel export needed (ordered export) 11 – pass thru (out of order export) Allocation Size (4 bits) (SIZE) Position Allocated (POS_ALLOC) First thread of a new context (FIRST) Last (1 bit), (LAST)

Status Bits

VALID	TUDE	DENIDINIO	OFDIAL		0175	BOO ALLOO	FIDOT	1 4 0 7		
VALID	IYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST		
1	ALU	0	0	0	0	0	1	0		

Then the thread is picked up for the execution of the first control flow instruction:

Exhibit 2039.doc 84302 Bytes*** [©] ATI Confidential. Reference Copyright Notice on Cover Page [©] ***

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
	9 July, 2003	4 September, 2015		52 of 54
17	1 O D 1 O m O m.	1 77. 0 77. 0 77.	- 0 77 1 77 0 77	

Execute 0 Alu 0 Alu 0 Tex 0 Tex 1 Alu 0 Alu 0 Tex 0 Alu 1 Alu 0 Tex

It executes the first two ALU instructions and goes back to the RS for a resource request change. Here is the state returned to the RS:

State Bits											
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD		
0	2	0	0	0	0	0	0	0	0		

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	0	0	0	0	0	1	0

Then when the texture pipe frees up, the arbiter picks up the thread to issue the texture reads. The thread comes back in this state:

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
0	4	0	0	0	0	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	1	0	0	0	1	0

Because of the serial bit the arbiter must wait for the texture to return and clear the PENDING bit before it can pick the thread up. Lets say that the texture reads are complete, then the arbiter picks up the thread and returns it in this state:

State Bits											
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD		
0	6	0	0	0	0	0	0	0	0		

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	0	0	0	0	0	1	0

Again the TP frees up, the arbiter picks up the thread and executes. It returns in this state:

State Bits											
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD		
0	7	0	0	0	0	0	0	0	0		

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	0	0	0	1	0

Now, even if the texture has not returned we can still pick up the thread for ALU execution because the serial bit is not set. The thread will however come back to the RS for the second ALU instruction because it has the serial bit set.

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AMD1044_0257974

ATI Ex. 2109 IPR2023-00922 Page 264 of 326

		GINATE DAT July, 2003		EDIT DATE 4 September, 2015		DOCUM GEN-C	PAGE 53 of 54				
State Bits											
CFP E	CM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD		
0 8		0	0	0	0	0	0	0	0		

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	1	0	0	0	1	0

As soon as the TP clears the pending bit the thread is picked up and returns:

State Bits											
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD		
0	9	0	0	0	0	0	0	0	0		

Status Bits

					COMPANY AND			
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	0	0	0	0	0	1	0

Picked up by the TP and returns: Execute 0 Alu

State Bits

ALL O MANAGER COMPANY								COROLAND IN COLOR OF THE OWNER	
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
1	0	0	0	0	0	0	0	0	0

Status Bits

L.	otatus pits								
	VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
	1	ALU	1	0	0	0	0	1	0

Picked up by the ALU and returns (lets say the TP has not returned yet): Alloc Position 1

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
2	0	0	0	0	0	0	0	0	0

Status Rits

otatus bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	01	1	0	1	0

If the SX has the place for the export, the SQ is going to allocate and pick up the thread for execution. It returns to the RS in this state:

Execute 0 Alu 0 Tex

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
3	1	0	0	0	0	0	0	0	0

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

		TE DATE , 2003	EDIT D/ 4 Septembe		R400	R400 Sequencer Specification			E 54
 Status Bits									
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS ALLOC	FIRST	LAST	

Now, since the TP has not returned yet, we must wait for it to return because we cannot issue multiple texture requests. The TP returns, clears the PENDING bit and we proceed:

0

1

0

1

0

0

Alloc Param 3

1

TEX

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
4	0	0	0	0	1	0	0	0	0

Status Bits								
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	0	10	3	1	1	0

Once again the SQ makes sure the SX has enough room in the Parameter cache before it can pick up this thread.

Execute_end 0 Alu 0 Tex 1 Alu 0 Alu

1

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
5	1	0	0	0	1	0	100	0	0

Status Bits	Γ	Status	Bits
-------------	---	--------	------

No. of the second s					SHOW NEEDS WITH SHOW	And the second		
VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	TEX	1	0	0	0	1	1	0

This executes on the TP and then returns:

State Bits									
CFP	ECM	LI	CRP	PB	EXID	GPR	EB	CPTR	LOD
5	2	0	0	0	1	0	100	0	0

Sta	tus	Bits	

VALID	TYPE	PENDING	SERIAL	ALLOC	SIZE	POS_ALLOC	FIRST	LAST
1	ALU	1	1	0	0	1	1	1

Waits for the TP to return because of the textures reads are pending (and SERIAL in this case). Then executes and does not return to the RS because the LAST bit is set. This is the end of this thread and before dropping it on the floor, the SQ notifies the SX of export completion.

24. Open issues

Need to do some testing on the size of the register file as well as on the register file allocation method (dynamic VS static).

Saving power?

Exhibit 2039.doc 84302 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***

AP	ORIGINATE DATE	EDIT DA	TE	DOCUMENT-REV. NUM.	PAGE
	13 November, 2000	4 September	r, 2015	GEN-CXXXXX-REVA	1 of 16
\uthor:	Steve Morein	•			
sue To:			Copy No		
Jue 10.			oopy No		
	R400	Architec	ture	Proposal	
			0.4		
		ver	0.1		
verview: Th	he is a proposal for the overa	II architecture of	the R400.	It is also just a proposal, and nothir	ng is decided
	et.				-
AUTOMATI Document	ICALLY UPDATED FIELDS:	FireWire HD:r40	lenec		
) Top Level Spec			
		APPRO	OVALS		
	Name/Dept			Signature/Date	
emarks:					
				NFORMATION THAT CO	
SUBST				EREST OF ATI TECHNOL	OGIES
	INC. THROUGH L	JNAUTHOR	IZED US	SE OR DISCLOSURE.	
Convright 00	00 ATI Technologica Inc. Al	l righto rocanic -	The met-	rial in this document constitutes	unnuhliched
				rial in this document constitutes an provide notice that ATI owns a cop	
Inpublished	work. The copyright notice	is not an admi	ssion that	publication has occurred. This w	ork contains
				t of this document may be used, re ssion of ATI Technologies Inc."	produced, or
anonnilleu III	any form of by any means with		men permit	Solution Artificonnologies inc.	
Exhibit 2040).doc 26858 Bytes*** © All Confid	aential. Keterend	ce Copyrig	ht Notice on Cover Page © *** _{09/04/1}	15 04:43 PM

ATI 2040 LG v. ATI IPR2015-00325 AMD1044_0257977

> ATI Ex. 2109 IPR2023-00922 Page 267 of 326



Table Of Contents

1. FEA	TURES	6
1.1 AC	3P8x and possibly serial AGP	6
1.2 12	8 Bit memory interface	6
1.3 Ne	early transparent dual chip	6
1.4 Ur	nified processing pipe	6
1.5 Fr	ont end scaling	6
	ontrol processor	
1.7 Re	eal-Time drawing command ability	7
1.8 3 E) Features	7
1.8.1	Noise Textures	7
1.8.2	Shadow buffers	7
1.8.3	Anti-Aliasing	7
1.8.4	Texture compression	7
1.8.5	Z compression	7
1.8.6	Filtering	
1.8.7	Curved Surface Support	
	gh color depth	
	Performance	
	ΞΑ	
	IEDULE	
)CESS	
	AL CHIP	
	IERAL RENDERING OPERATION	
	nified Shader	
) Rendering	
	Rendering	
	eal-time Rendering	
	PLAY OPERATION	
	ORT BLOCK DESCRIPTIONS	
	′S	
9.1.1	HBIU1	
9.1.2 9.1.3	HDP MISC	
9.1.3	MISC	
9.1.4	VIP	
		5
9.1.6 9.1.7		5
9.1.7	ClockGen	
9.1.8	CIOCKGEIT	
9.1.9	RBBM	
9.1.10	КЪВІИ	
	splay	
	fx1	
9.3.1	PrimitiveAssembly/vertex cache	
9.3.2	Raster Engine.	
		0
Exhibit 2040.	doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:43 PM	

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE		
	13 November, 2000	4 September, 2015	GEN-CXXXXX-REVA	3 of 16		
9.3.3	Sequencer					
9.3.4	Datapath					
9.3.5	TextureEngine					
9.3.6	RenderBackend					
10. TOP LEVEL INTERCONNECTIONS						
10.1 Fir	10.1 First Level Sub Heading					
10.1.1						

Exhibit 2040.doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15 04:43 PM}

AMD1044_0257979

ATI Ex. 2109 IPR2023-00922 Page 269 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
	13 November, 2000	4 September, 2015	Go to "File -> Properties ->	4 of 16	
Revision Changes:					
Rev 0.0 (Steve Morein) Date: November 6, 2000 Initial revision.		Doct	ument started		

Rev 0.01 Steve Morein Date: November 10,2000 Document continued

Exhibit 2040.doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:43 PM

AMD1044_0257980

ATI Ex. 2109 IPR2023-00922 Page 270 of 326



ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
13 November, 2000	4 September, 2015	GEN-CXXXXX-REVA	5 of 16	

Introduction

This document outlines a proposal for the r400 architecture.

A minor note: in the middle of writing this I decided that it makes the most sense to call the "pixel" pipelines shader pipeline since they handle vertices and pixels. I have not gone through this to make sure that my usage is consistent.

Exhibit 2040.doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:43 PM

AMD1044_0257981

ATI Ex. 2109 IPR2023-00922 Page 271 of 326



1. Features

1.1 AGP8x and possibly serial AGP

The R400 will at a minimum support AGP4x and AGP8x interfaces. We may also support 3.3V i/o including AGP2x and 3.3V PCI. We need to consider how we interface to LDT (AMD) and possibly the Motorola rapid I/O that may be used in future Apple Designs (G5).

1.2 128 Bit memory interface

We are thinking of only supporting a 128 bit interface to memory. The memory will be configured as four channels of 32 bits each. The atomic fetch until will be 256 bits in expectation that some high speed memories will use a prefetch-8 architecture. Logic in the memory controller will optimize down to 128 bit writes when possible on DDR or prefetch 4 memories. Memories up to 500 MHz will be supported (1 gigabit data rate).

Memory is the most open issue on the R400. We need to develop a roadmap ASAP for how memory will develop, and this may significantly affect our plans.

1.3 Nearly transparent dual chip

To be able to address the very high end desktop/enthusiast market we will support a glueless two chip design instead of a 256 bit bus. Unlike previous dual chip designs we have done, this is targeted to be a mainstream product. This implies that it can easily be WHQL'd, and can accelerate all applications and benchmarks, not just a subset of full screen apps. A separate document outlines the two proposals we are looking at for the dual chip design.

There will be costs added to the base chip to support this. Design time, pins, and area will be impacted by adding this support.

1.4 Unified processing pipe

The most ambitious feature in this design is the "truly unified pipe": a single programmable pipeline is used for 2D, Video, 3D vertex, and 3D pixel operations. The unified pipeline does all of its calculations in 32 bit floating point, the same as the existing vertex transform in previous chip, and the next step in the precision of the color/pixel calculations which have increased from 8 bits (R100), through 16 bits (R200), to the 20 bits in the R300.

There is an area cost to the unified pipeline since we are forced to go to 32 bit precision for color, when application requirements may need less (22 to 24 bits). However the unified pipeline results in a single math/register structure compared to the separate structures in a more traditional design. It is hoped that by only needing to design the one structure we can make the investment in design time and effort to really optimize the area.

Some of the benefits to merging the pipelines include allowing the vertex operations to do texture fetches, which we could not afford add logic to the transform pipe to do, a single programming model for both operations, more precision on color than we would normally provide, and the ability to support significantly more registers and instructions in pixel shaders.

One important benefit is load balancing. In the current pipeline when the app it transform bound the pixel pipeline is idle some significant portion of the time, and when the app is raster bound the transform hardware idle. The unified pipeline presented here dynamically allocates its processing power between transform and raster.

1.5 Front end scaling

We will remove the back end scaling capability from the display, and replace it with a non-scaling overlay. This will require us to be able to implement scaling using the unified pipeline. Key features that will need to be supported are large filter kernels, de-interlacing, frame rate conversion, and good support for YUV and color conversion.

Exhibit 2040.doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:43 PM

AMD1044_0257982

ATI Ex. 2109 IPR2023-00922 Page 272 of 326



ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
13 November, 2000	4 September, 2015	GEN-CXXXXX-REVA	7 of 16	

1.6 Control processor

To allow us to emulate a backend scaler and to enable new applications the control processor will be enhanced with event based streams. These are secondary, real time, command streams that start execution when an event happens.

1.7 Real-Time drawing command ability

To allow for the emulation of backend scaling as well as support new features we need to be able to interrupt the 3D pipe and be able to execute high priority commands with low latency. At the moment it appears far to difficult to be able to insert a new command at the top of the 3D pipeline and meet latency requirements (which I believe we wish to define as around a 1/16 of a frame refresh). This would require us to be able to interrupt triangles in the midst of rasterizing, inset vertices in the midst of a large vertex array, and other nasty things. I think instead we can get by with a second rasterizer which drives the pixel pipelines. Setup would be done with software, but since the majority of the real time rasterization is expected to be simple

1.8 3D Features

There are a number of new 3D features we are considering for inclusion. Additional features may be added, and some of these may be dropped.

1.8.1 Noise Textures

Perlin style noise is useful for a number of applications. It is generated on chip and consumes no external memory bandwidth. It also larger than any physical texture can be: 256x256x256 lattice points, and still has detail when the resolution is 4Kx4Kx4K. There is an opportunity to get this adopted as part of dx9.

1.8.2 Shadow buffers

John Carmack is using shadow volumes to generate shadow effects in doom3. Shadow volumes are very poor way to use modern 3D pipelines. (will add more detail here later). Shadow buffers have two key limitations: very high resolutions are required to avoid aliasing, and traditional shadow buffers can not be mip-mapped so filtering is real problem. Through a combination of the z techniques we have developed and, hopefully, deep shadow buffers, we can solve both of these problems and widely enable shadow buffers.

1.8.3 Anti-Aliasing

We want to further improve the anti-aliasing used in the R300 by reducing the needed memory, and possibly increasing the number of samples per pixel. The goal is more than fifty percent of the performance and less than three times the memory of anti-aliased rendering. We should also look into improved methods.

1.8.4 *Texture compression*

To further reduce bandwidth we need to improve texture compression. We need to achieve both better compression that S3TC, and have a high enough quality that textures that would lose too much detail with S3TC can be compressed. Both of these goals do not need to be achieved simultaneously on all textures. We also need to look at compression of non-traditional surfaces such as normal maps.

1.8.5 Z compression

We will build on the R300 slope based compression but we are looking at supporting maxmin for cachelines that do not compress with slopes (either too many slopes per cacheline, or the pixel shader modifies the z value)

Exhibit 2040.doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15 04:43 PM}

AMD1044_0257983

ATI Ex. 2109 IPR2023-00922 Page 273 of 326



1.8.6 *Filtering*

As part of the move to front end scaling we need better than bi-linear filters. Goals are : arbitrary sized separable filters and 4x4 bi-cubic. Being able to support programmable weights is nice.

1.8.7 Curved Surface Support

We need to figure out how we are going to support curved surfaces in this architecture. I think that we can find a way to use the wide ability of a vertex shader to implement acceleration for subdivision surfaces, but the vertex only level of processing in the shader pipeline means that something ahead of it needs to set up the surface. At one point I imagined that we could use the sibyte processor as a CP, which would have the power to do the curved surface setup. That is obviously no longer possible.

1.9 High color depth

We will support a 64 bit color buffer (16:16:16:16), the exact format (fixed, floating, etc.?) has yet to be decided

1.10 Performance

I think we can increase the clock speed from 300 MHz to 500MHz.

Historically the goal has been to double speed in each generation, assuming a constant clock speed. However since we are considering the dual chip solution for the very high end we may not need to be 2x the speed of the R300. Our use of a 128 bit memory bus instead of 256 bits will impose a potentially lower bandwidth.

That said I would still like to aim for 2x the internal processing capability of the R300:

	R300	R400
Clock speed	300 MHz	500 MHz
Pixels per clock	8	16
Bi tex fetches per pixel	2	2
ALU ops/clk (mac's)	64 (dedicated)	192 (shared)
Peak Tri/Sec	150	250
Peak xform fp ops/clk	16? (dedicated)	192 (shared)

We may need to reduce this performance goal to meet our area goal.

2. Area

The area goal for the R400 is 10mm on a side in .13 micron CMOS There will probably be a lower cost version with a target area of 8.5 on a side.

3. Schedule

Tapeout	April 2, 2002
Samples	May, 2002
Production	Nov, 2002

4. Process

At the moment this looks like an easy choice: .13 will be in production for over a year, and .10 does not show up until the very end of 2002 according to the TSMC and UMC roadmaps.

We will probably want to be in a flip chip packaging approach to meet power distribution goals. It will also reduce the cost of the dual chip option by making the extra pins needed for the interface cheaper. Is there a way to have an

Exhibit 2040.doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:43 PM

AMD1044_0257984

ATI Ex. 2109 IPR2023-00922 Page 274 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
13 November, 2000	4 September, 2015	GEN-CXXXXX-REVA	9 of 16	

option to wire bond it also? Possibly without the dual chip interface, and with less pwr/gnd forcing a lower clock speed. This may make sense for a lower cost sku.

5. Dual Chip

<need to copy the dual chip notes over and add to them>

6. General rendering operation

6.1 Unified Shader

The unified shader is a simd/vector engine that performs the same instructions on four sets of four (16 total) elements. For pixel shader operations the elements are pixels with the sets of four required to be $2x^2$ footprints. For vertex shader operations the sixteen elements are sixteen vertices. The basic element is a 4 value vector – frequently interpreted as x,y,z,w or r,g,b,a.

The user model for the unified shader is composed of a variable number of general purpose registers, a subset of which are usually initialized with data. An ALU can do simple math, conditional moves, and permutations on the registers, and the ability to do a limited number of memory reads using the texture cache. The number of register is variable, and the number of registers required for an operation are specified when the task is submitted to the unified shader. The unified shader will not start the task until there is enough free room for the tasks registers.

The unified shader is based on the R300 partially unified shader.

6.2 3D Rendering

For 3D rendering data is passed twice through the unified shader- once to transform the vertices and a second time to determine the color of the pixels.

The input to the 3D pipe is expected to be indexed vertex arrays. Linear vertex arrays can easily be supported by the CP generating sequential indices. Inline vertex data is an open issue, I would prefer to write it to memory and then fetch it as a vertex array rather than add a direct path.

The stream of indices is sent to the Primitive Assembly block by the CP. The front of the primitive assembly block maintains the tag for the vertex cache; The vertex cache stores transformed vertices. As misses are detected in the tag, the indices that miss are placed into 16 entry vectors. Each vector contains a state pointer, a pointer to the vertex shader to be used, and the 16 indices to vertices that need to be transformed. When either a vector is filled with 16 entries or a state change happens (so that the next vertex does not share the state and vertex shader with the previous vertex) the vector is issued to one of the "shader" pipelines for transformation. Which of the four shader pipelines it is issued to determined either by some effort of load balancing or a simple round robin. All that is submitted to the pixel pipeline is the state, the vertex program, and the indices. The shader pipeline will fetch the vertex array data through the cache infrastructure that is also used for texture fetches. After the tag the indices (actually now the indices into the vertex cache) are placed into a latency FIFO to hide the latency of transforming the vertices.

The shader pipeline receives the vector of 16 indices from the primitive assembly block. The shader pipeline operates, when rendering pixels, by processing a vector of four 2x2 pixel footprints, A total of 16 pixels. For vertex processing each of the pixels is replaced with a vertex. The vertex program includes information of how many local variables it will need. The rasterizer waits until that many local variables are free, (as each executing thread in the shader pipeline terminates it frees its local variables). With the proposed shader datapath the maximum number of local variables per vertex is 256. However this leaves no ability to hide latency, 16 to 32 local variables will probably maximize latency hiding and therefore performance. The vertex shader program can use all the capabilities of the shader pipeline including texture fetches and dependent lookups. At the end of the vertex program, the transformed coordinates must be output. One output will be the x,y,z,w position which we be stored in the position cache of the vertex cache. The vertex program may also output a number of parameter values (colors, texture coordinates, other

Exhibit 2040.doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:43 PM

AMD1044_0257985

ATI Ex. 2109 IPR2023-00922 Page 275 of 326

13 November, 2000 4 September, 2015 Go to "File -> Properties ->	10 of 16

interpolated inputs into the pixel shader). The parameter values must be output as a multiple of four 128 bit words, as the parameter cache is designed for this.

The primitive assembly block reads the indices back out of the latency FIFO and accesses the position cache portion of the vertex cache. It assembles the vertices into primitives (lines, triangles, rectangles, quads?, points, ?). Baricentric values are assigned to the vertices, and will be used later in the rasterizer to interpolate the parameters. The parameters are not accessed by the primitive assembly logic, which only works from the position data. The primitive is clipped against both the viewing volume as well as user clip planes, with fractional baricentric coordinates assigned to the clipped primitive sections. The primitive goes through the perspective divide and the viewport transform. The resulting screen space primitive is setup (plane equations for 1/W, Z, and the baricentric coordinates). The resulting primitive data, including the indices back into the parameter portion of the vertex cache are broadcast to the four pipes. The final time that an index is output that access the oldest vertex cache line, a token is also sent. When all of the four pipelines return the token the primitive assembly block can free that cacheline and allow it to be used for a new vector of vertices. The performance goal in the primitive assembly block is a triangle every two clocks.

Each pipe has a FIFO in front of the rasterize to load balance. Each pipe will handles 16x16 sections which are interleaved between the pipes. To maximize the effective size of the FIFO we will probably cull the triangle list before the FIFO. The rasterizer will request the parameter data from the parameter cache for the primitives. A small latency hiding FIFO will hide the latency of the access to the parameter cache. The parameter cache is 512 bits wide, and the interfaces from the parameter cache to the rasterizer are 128 bits wide, this allows the parameter cache to output one pipelines request per clock, which is serialized over four clocks, keeping all four interfaces busy. The rasterizer keeps a small cache of three to four vertices, this allow only the new parameter to be fetched when adjacent triangles are processed. The parameter cache interface imposes a second performance limits, in the worst case each polygon covers all four pipelines and there are no vertices shared from triangle to triangle. In this case the peak performance is (500 MHz / (4 pipelines * 3 vertices) = (500/12) = 41.6 million triangles per second. In the best case triangles are perfectly stripped and never cross over pipeline boundaries. In this case the peak performance (If we ignore the setup limit) is 500 million triangles per second. As a practical manner we should be able to approach the setup limit of 250 Million triangles per second.

The rasterizer also contains a portion of the hierarchical Z memory. We are looking into moving this into a cache based approach, but that is far for certain at this point. We would like to be able to do heirz culling at a speed in excess of 64 pixel per clock per pipelines (256 pixels per clock total). We are also going to consider some of the improved latency heriz options to improve culling efficiency.

The rasterizer will generate four pixels per clock if there are no more than eight interpolated parameters. The rasterizer generates vectors of four 2x2 footprints (16 pixels). Each 2x2 footprint must be screen aligned and from the same triangle (with a single shared z slope). The four footprints only need to share the same state and shader program.

Before starting the processing of a vector the rasterizer (which includes the sequencer for the shader pipeline) checks to make sure that there are enough free registers in the shader pipeline for the pixel shader program. If not, it stalls until there are enough. The rasterizer also needs to arbitrate between the three streams of vectors to be shaded: the vertex stream, the pixel stream, and the real time stream. I think it will be sufficient for the real time stream to have priority over the vertex stream which has priority over the pixel stream. This will meet the realtime demands, and keep the vertex cache filled.

The vector is then processed by the shader pipeline. We will probably support up to eight sequentially dependent texture fetches. (to use the R300 terminology, eight clauses). 16 (8?) textures are supported, but each texture can be accessed multiple times by a single pixel shader which can provide a different address each time. This is especially useful for complex filters.

The output of the pixel shader is the final color of the fragment. The pixel shader may also replace the Z value. Fog and stippling must be done in the pixel shader program.

The render backend does the z compare, stencil operation and color alpha blend.

The texture fetch path has a number of design options. One option is an approach where the local, multiported, texture cache is small (1 to 4 KB), and contains uncompressed color in a canonical format (32 bits per pixel) and uses a 4x2 or 4x4 cacheline. This is backed up by a large (>16KB) L2 cache which also stored uncompressed 8x8 cachelines. The decompression logic lives between the memory controller and the L2 cache.

Exhibit 2040.doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:43 PM

AMD1044_0257986

ATI Ex. 2109 IPR2023-00922 Page 276 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
13 November, 2000	4 September, 2015	GEN-CXXXXX-REVA	11 of

An alternative design uses the L2 cache to contain data in memory format (compressed) which is decompressed as needed to fulfill L1 texture cache misses. This will increase the effective size of the L2. The L2 cache is distributed, with 1/4 of it residing in each memory controller. The Texture decompression logic can either be located in each shader pipeline, or exist as a shared block(s) that receive data from all four memory controller and send the decompressed 4x4 cachelines to each shader pipeline. The unified decompression block will result in better performance, and possibly less area, at the cost of some of the scalability.

Assuming that we chose the L2 in memory controller and the unified decompression logic, the texture path would work as follows:

In a four pipeline design there are two texture decompression blocks, one for the "left" texture units in each shader pipeline, and the second for the "right" texture units. In the two pipeline, lower cost, version of the chip only a single decompression pipeline is used, serving the left and right texture units.

The L1 texture cache receives a texture request from its shader pipeline. The usual tag and latency FIFO is used to generate the misses. These are sent to the shared texture decompression block, which looks up the texture to find the physical address and then sends the request to the L2 cache in the memory controller. The L2 also has a latency FIFO and tag, and will return the data in order (but there is no order guaranteed between the data returning from each L2). The decompression block has a buffer which is used to place the data from the memory controllers back in order. The decompression logic decompresses the texture and returns, in order, the 4x4 cachelines that the L1 caches are requesting. Most of the compression techniques we are considering are based on an 8x8 tile (or 4x4x4), when necessary the decompression logic will decompress an entire 64 pixel tile and only return the requested 16 pixels to the L1 cache. This will tend to increase the bandwidth between the decompression logic and the L2 cache as 8x8 blocks are repeatedly requested to provide different 4x4 subtiles to the L1. The L2 cache will prevent the repeated reads from going to memory, and we will probably implement an "L0" style cache in front of the L2 to also catch the redundant requests.

Each memory controller will have two 64 bit read return buses, one to each of the two decompression blocks, each decompression blocks drives a separate 128 bit bus to each of the four shader pipelines. This will tend to have better utilization and load balancing than having the memory controller drive a 32 bit bus to the decompression logic in each shader pipeline. While the total number of wires is similar (128 bits per memory controller, 128 bits into each texture cache) we are less likely to leave the texture pipes starved when there is some imbalance.

6.3 2D Rendering

2D rendering is implemented in the 3D pipeline. The first reason for the change is performance; The current 2D pipeline can render 128 bits per clock (16 8 bit pixels, 8 16 bit pixels, 4 32 bit pixels). The 3D pipe can render 16 pixels per clock, and the pixels can be 8 to 64 bits wide. Secondly routing the three busses needed by the 2D engine (src read, dest read, dest write) has a certain cost, and complicated the design of the chip. If we attempt to improve the performance of the 2D pipe these busses will increase in size, further complicating the design. Another reason is dual chip rendering, it would be nice if 2D as well as 3D operations increase in speed. (2D operations include things like color clears and texture uploads that do show up in 3D benchmarks.

There are four issues currently known that will affect placing the 2D commands into the 3D pipelines:

- 1) Command compatibility
- 2) Hostpath blits
- 3) ROP3
- 4) Overlapping blits.

We wish to be as compatible as possible with the existing PM4 2D model. This will require that the CP be enhanced, allowing it to translate 2D commands into commands understood by the 3D pipe. We will ad interfaces to the 3D pipe to make this easier but there will still be significant amount of work that still needs to be done by the CP.

Host path blits can no longer work as they do now. Four pipelines will be attempting to execute the requested command in parallel, walking the area to be drawn in some tightly tiled pattern to optimize memory and cache performance. This bears little resemblance to the single linear stream of data from embedded into the PM4 command stream. In addition the shader pipelines are heavily optimized for a pull, "reverse" mapping model, and not a push, "forward" model. The basic solution to this problem is for the 3D pipes to pull the source data directly out of a

Exhibit 2040.doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:43 PM

AMD1044_0257987

ATI Ex. 2109 IPR2023-00922 Page 277 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
13 November, 2000	4 September, 2015	Go to "File -> Properties ->	12 of 16

command/data buffer, in whatever order, and in whatever parallel streams exist. Whether we give the PM4 engine the ability to skip over the hostpath data, or force the driver to move hostpath data into a second command ring still needs to be decided. It is possible that one or more changes may be needed to the driver for this.

2D supports a ROP3 operation that requires the destination color as well as two sources: the source, and the pattern. To support this the pattern color is output by the pixel shader on the Z output which is not used by 2D operations. The render backend now has all three needed sources.

Overlapping blits is an ugly problem that I have not yet found an acceptable solution to. More work needed.

One benefit to these changes is the 2D operations will also be accelerated by the second chip in a dual chip board.

6.4 Real-time Rendering

7. Display operation

The display must be able to display from microtiled surfaces and overlays. This will generally force us to adopt line buffers.

The display should support at least two outputs, ideally we will be able to support two high resolution outputs and a low resolution output (TV out)

We will drop support for overlay scaling, and therefore supporting an overlay on all displays becomes affordable, fixing a "bug" that our current dual display products suffer from.

We will place the overlay line buffers in the memory controllers, this changes the interface from the memory controllers to the display from a wide "bursty" interface to a narrow continuos interface.

<this rest of this was copied out of another document and needs some editing to fit in this document. Bear with us as
construction of this document continues>

Video buffer operation:

At beginning of even scanline (scanline 0) 1/2 of line buffer is filled.

The upper half of the buffer is read out, at the same time as the second half of the buffer is filled. When the scanout reaches the midpoint 3/4 of the buffer is filled. When the scanout reaches the end of the scan the buffer is filled. The speed at which the buffer is filled must be greater than 1/2 of the rate at which the buffer is scanned out.

For the odd scanlines, the buffer is completely filled at the start of scanout (as a result of the even scanline finishing properly). As the lower half is scanned out, reads are issued to fetch the data for the next pairs of scanlines. At the end of the odd scanline, the buffer is expected to contain half of the data for the next scanline.

Another way of looking at this is as follows:

At the beginning of the odd scanline the scan buffer is filled. As each word is read from the buffer and sent to the display logic, a request is made to the memory controller to fill in the data. It is not necessary for all the data for the next scanline pair to be fetched by the time that the scanline reaches the end, the real requirement is for the last word in the scanline buffer to get there just before it is read, at the end of the next even scanline.

We will support a single non-scaled overlay per each display.

Some bandwidth numbers In reality we do not need to deal with quite as much bandwidth as the FIFO in the display can hide the horizontal retrace. 350 MHz primary fetch, 32 bit data: 350 MHz primary display, 32 bits 350 MHz overlay fetch

Exhibit 2040.doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15 04:43 PM}

AMD1044_0257988

ATI Ex. 2109 IPR2023-00922 Page 278 of 326



350 MHz overlay display total: 350 MHz * 16 bytes (128 bits)

Since we support dual monitor, this is doubled.

One design option is to split the display scanline into pieces and move them into the memory controller. This greatly reduces the exposed bandwidth in the system (reducing power and routing problems)

If we assume that there are four memory controllers, each with 2GB/s of memory bandwidth then the following will work:

core clock speed: >= 1x the memory clock memclk = 500 MHz coreclk = 500 MHz

each memory interface is 32 bits at a DDR rate, and the fetch granularity is 256 bits. Therefore if data was continuously received into the display FIFO 64 bits would be received every clock. A 256 bit interface at the core clock rate is more than adequate.

the memory size needed for two 2048 displays is : 2048*4 bytes * 2 scanlines * 4 buffers is 64KB. So each buffer is 16KB (128 Kbit). With a 256 bit interface the memory is 256x512 single ported.

For writes into the write buffer as a result of memory fetches, a small buffer reorders data between pairs of 256 bit words so that while what is read from memory is 256 bits containing two vertically stacked 128 bit words, what is written is two 128 bit words that are on the same scanline.

The interface from the memory controller to the display only needs to be big enough for the sustained bandwidth, and not the peak memory speed bandwidth. A 16 bit interface to each display seems like more than enough. (compare this to the rage 6 with two 128 bit busses between the memory controller and display)

A couple more notes:

for the most part the memory format is stored in the scanline buffer. The exception is 64 bit, which we would like to convert to something like 11:11:10 or 8:8:8:8 This may mean some sort of gamma circuit in the memory controller.

A LUT would exist in the display for gamma de-correction and pallet support.

The interleave between the memory controllers would have to be compatible with the tiling and still give good performance.

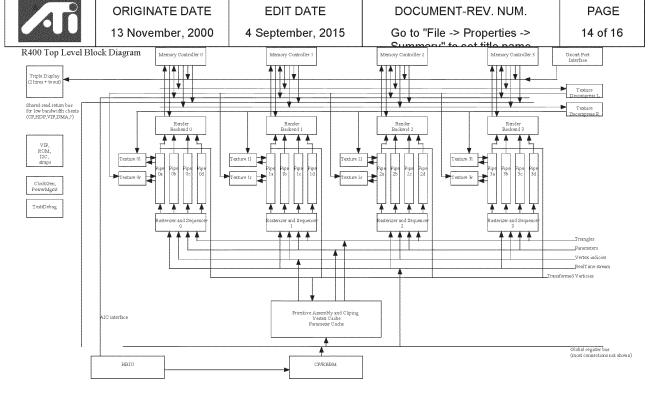
A big question is how does this work in a two chip board? I had been thinking about interleaving on a fine basis between the chips with a display controller in one chip fetching from both, but this somewhat flips that around. We need to route the video signals as an extra channel between the chips, this will add complexity, but it actually is less bandwidth since the overlay is combined first.

8. Block diagram

Exhibit 2040.doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:43 PM

AMD1044_0257989

ATI Ex. 2109 IPR2023-00922 Page 279 of 326



9. Short Block descriptions

9.1 SYS

The system blocks support the chip, but are not graphics specific.

9.1.1 HBIU

The HBIU is the interface to the host bus. It implements four interfaces: register/read write, HDP for host access to memory and the

Exhibit 2040.doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15 04:43 PM}

AMD1044_0257990

ATI Ex. 2109 IPR2023-00922 Page 280 of 326

- 9.1.2 HDP
- 9.1.3 M/SC
- 9.1.4 *Rom*
- 9.1.5 VIP
- 9.1.6 /2C
- 9.1.7 ?
- 9.1.8 ClockGen
- 9.1.9 CP
- 9.1.10 *RBBM*
- 9.1.11 MC

The memory controller is distributed, each of the four memory channels has a separate memory conttoller. Each memory control contain a part of the L2/Line buffer memory. This large buffer serves a number of purposes in the graphics chip, including L2 cache for textures and verticies.

- 9.2 Display
- 9.3 Grfx
- 9.3.1 PrimitiveAssembly/vertex cache
- 9.3.2 Raster Engine
- 9.3.3 Sequencer
- 9.3.4 Datapath
- 9.3.5 TextureEngine
- 9.3.6 RenderBackend
- 10. Top Level Interconnections
 - 10.1 First Level Sub Heading
 - 10.1.1 Second Level Sub Heading'

Exhibit 2040.doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:43 PM

AMD1044_0257991

ATI Ex. 2109 IPR2023-00922 Page 281 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
13 November, 2000	4 September, 2015	Go to "File -> Properties ->	16 of 16	

Exhibit 2040.doc 26858 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15 04:43 PM}

AMD1044_0257992

ATI Ex. 2109 IPR2023-00922 Page 282 of 326

	ORIGINATE DATE	EDIT DA	TE	DOCUMENT-REV. NUM.	PAGE
	11 March, 2001	4 Septembe	r, 2015	GEN-CXXXXX-REVA	1 of 32
Author:	Steve Morein				
Issue To:			Copy No:		
				п ф ^а н d п	
	R400 I	op Leve	el Spe	cification	
		ver	0.2		
Overview: This	s replaces the R400 architect	ture specification	I.		
AUTOMATIC	CALLY UPDATED FIELDS:				
Document L	.ocation: Docu	iment1			
Current Intra	anet Search Title : R400) Top Level Spec			
	Name/Dept	APPR	OVALS	Signature/Date	
	·			Ŭ	
Remarks:					
		S CONFIDE	NTIAL II	NFORMATION THAT COL	ILD BE
				REST OF ATI TECHNOL	
00001/				E OR DISCLOSURE.	
			2000 K		
				ial in this document constitutes an provide notice that ATI owns a cop	
unpublished w	vork. The copyright notice	is not an adm	ission that	publication has occurred. This we	ork contains
				of this document may be used, rep sion of ATI Technologies Inc."	produced, or
	any form of by any mound wit		and bounds	e.e or a tradition of the second	
			• •		
Exhibit 2041.D	OC 48154 Bytes*** © ATI Confi	dential. Referen	ce Copyrig	ht Notice on Cover Page © *** _{09/04/1}	^{5 12:48 PM} ATI 204
					A11 204

IPR2015-00325

AMD1044_0257993

ATI Ex. 2109 IPR2023-00922 Page 283 of 326



Table Of Contents

1.			TURES	
1.1			8x	
1.2	~		Bit Memory Interface	
1.3			ed Processing pipe t end scaling	
1.5			-Time drawing command ability	
1.6			eatures	
	1.6	5.1	Noise Textures	
	1.6	5.2	Shadow buffers	
	1.6	5.3	Sort Independent Transparency	. 8
	1.6	5.4	Anti-Aliasing	. 8
	1.6	5.5	Texture compression	. 8
	1.6	5.6	Z compression	. 8
	1.6	5.7	Texture Filtering	. 8
	1.6	5.8	Curved Surface Support	. 9
	1.6	5.9	Displacement maps	. 9
1.7	7		color depth	
2.				
3. 4.			EDULE CESS	
		FIV		• V
		GEN	ERAL CHIP OPERATION	. 9
5 .			ERAL CHIP OPERATION	
5. 5.1 5.2	2	Unifie 3D R	ed Shader endering	. 9 10
5. 5.1 5.2 5.3	2	Unifie 3D R Real	ed Shader endering Time Rendering	. 9 10 12
5. 5.1 5.2 5.3 5.4	2 3	Unifie 3D R Real State	ed Shader endering Time Rendering Management	. 9 10 12 13
5. 5.1 5.2 5.3 5.4 5.5	2	Unifie 3D R Real State Bad I	ed Shader endering Time Rendering Management Data	. 9 10 12 13 13
5. 5.1 5.2 5.3 5.4 5.5 5.6	2	Unifie 3D R Real State Bad I Displ	ed Shader endering Time Rendering Management Data ay operation	. 9 10 12 13 13
5. 5.1 5.2 5.3 5.4 5.5	2	Unifie 3D R Real State Bad I Displ BLO	ed Shader endering Time Rendering Management Data	. 9 10 12 13 13 14 15
5. 5.1 5.2 5.3 5.4 5.5 5.6 6.	2	Unifie 3D R Real State Bad I Displ BLO BLO	ed Shader	. 9 10 12 13 13 14 15 16
 5.1 5.2 5.3 5.4 5.5 5.6 6. 7. 8.1 		Unifie 3D R Real State Bad I Displ BLO BLO HBIL	ed Shader	. 9 10 12 13 13 14 15 16
5. 5.1 5.2 5.3 5.4 5.5 5.6 6. 7. 8. 8.1	2 3 5 5	Unifie 3D R Real State Bad I Displ BLO BLO HBIL	ed Shader	. 9 10 12 13 13 14 15 16
5.1 5.2 5.3 5.4 5.5 5.6 6. 7. 8.1 8.1	2 3 5 5 8.1 8.1	Unifie 3D R Real State Bad I Displ BLO BLO BLO HBIL	ed Shader endering Time Rendering Management Data Data Ay operation CK DIAGRAM CK DIAGRAM CK DESCRIPTONS I – host bus interface unit	. 9 10 12 13 13 14 15 16 16
5. 5.1 5.2 5.3 5.4 5.5 5.6 6. 7. 8. 8.1	2 3 5 5 8.1 8.1 8.1	Unifie 3D R Real State Bad I Displ BLOO BLOO HBIU .1	ed Shader	. 9 10 12 13 13 14 15 16 16 16 16
5. 5.1 5.2 5.3 5.4 5.5 5.6 6. 7. 8. 8.1 8.1 8.2	2 3 5 5 8.1 8.1 8.1	Unifie 3D R Real State Bad I Displ BLO BLO HBIU .1	ed Shader endering Time Rendering Management Data ay operation CK DIAGRAM CKS CK DESCRIPTONS U – host bus interface unit Description Major interfaces. Block diagram. control processor.	. 9 10 12 13 13 14 15 16 16 16 16 17
5. 5.1 5.2 5.3 5.4 5.5 5.6 6. 7. 8.1 8.1 8.2 8.2	2 5 5 8.1 8.1 8.1 2 8.2	Unifie 3D R Real State Bad I Displ BLO BLO BLO HBIU .1 .2 .1 .2 .1	ed Shader	. 9 10 12 13 13 14 15 16 16 16 16 17
5. 5.1 5.2 5.3 5.4 5.5 5.6 6. 7. 8. 8.1 8.2 8.2	2 5 5 8.1 8.1 8.1 8.2 8.2	Unifie 3D R Real State Bad I Displ BLO BLO BLO BLO (BLO (BLO (BLO (BLO (BLO (BLO (CP - 2.1	ed Shader endering Time Rendering Management Data ay operation CK DIAGRAM CKS CK DESCRIPTONS U – host bus interface unit Description Major interfaces Block diagram control processor	. 9 10 12 13 14 15 16 16 16 17 18 18
5. 5.1 5.2 5.3 5.4 5.5 5.6 6. 7. 8.1 8.1 8.2 8.2 8.2	2 5 5 8.1 8.1 8.1 8.2 8.2 8.2	Unifie 3D R Real State Bad I Displ BLOO BLOO HBIU 1.1 1.2 1.3 CP – 2.1 2.2	ed Shader endering Time Rendering Management Data ay operation CK DIAGRAM CKS U – host bus interface unit Description Major interfaces. Block diagram control processor. Description	. 9 10 12 13 14 15 16 16 17 18 18 18 18
5. 5.1 5.2 5.3 5.4 5.5 5.6 6. 7. 8. 8.1 8.2 8.2 8.2 8.3	2 5 5 8.1 8.1 8.1 8.2 8.2 8.2	Unifie 3D R Real State Bad I Displ BLO BLO HBIU .1 .2 .3 CP – 2.1 2.2 RBBI	ed Shader endering Time Rendering Management Data ay operation CK DIAGRAM CKS CK DESCRIPTONS U – host bus interface unit Description Major interfaces Block diagram control processor	. 9 10 12 13 14 15 16 16 16 17 18 18 19 19

Exhibit 2041.DCC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
<u>~~</u> UU	11 March, 2001	4 September, 2015	GEN-CXXXXX-REVA	3 of 32
8.3.2	Major interfaces			
8.3.3	Block diagram			
8.3.4	RBBM operation			
8.4 CLK	•			
8.4.1	Description			21
8.4.2	Major interfaces			21
8.4.3	Block diagram			
8.5 TC -	- test controller			
8.5.1	Description			22
8.5.2	Major interfaces			
8.5.3	Block diagram			
	 Video input port 			22
8.6.1	Description			
8.6.2	Major interfaces			22
8.6.3	-			
8.7.1	Description			22
8.7.2	Major interfaces			
8.7.3				
8.8.1				
8.8.2	Major interfaces			22
8.8.3				
8.9.1				
8.9.2	*			
8.9.3				
8.10.1				
8.10.2				
8.10.3	9			
8.11.1	•			
8.11.2				
8.11.3	-			
8.12.1				
8.12.2	Major interfaces			

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

8.12.3

8.13

AMD1044_0257995

ATI Ex. 2109 IPR2023-00922 Page 285 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	11 March, 2001	4 September, 2015	R400 Top Level Spec	4 of 32
8.13.1	Description	•••••••••••••••••••••••••••••••••••••••		
8.13.2	Major interfaces			
8.13.3				
8.14.1				
8.14.2				
8.14.3				
8.15 RE - 8.15.1	-			
8.15.2				
	w.			
8.15.3 8.16 SP -				
8.16.1				
8.16.2				
8.16.3	19			
	-			
8.17.1	Description			
8.17.2	Major interfaces			
8.17.3				
8.18.1	·			
8.18.2	-			
8.18.3	-			
8.19 MC - 8.19.1				
8.19.2				
8.19.2				
9.1.1 c	Data formats			
9.1.2 F	Register Bus			
9.1.3 E	Block Communication	protocol		
9.2 Softwa	are			

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15 12:48 PM}

AMD1044_0257996

ATI Ex. 2109 IPR2023-00922 Page 286 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
	11 March, 2001	4 September, 2015	GEN-CXXXXX-REVA	5 of 32	
Desistent					

Revision Changes:

Rev 0.0 (Steve Morein) Date: March 11, 2001 Initial revision.

Date March 14,2001

Document recreated from earlier documents

Finally got back to editing it.

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15 12:48 PM}

AMD1044_0257997

ATI Ex. 2109 IPR2023-00922 Page 287 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
11 March, 2001	4 September, 2015	R400 Top Level Spec	6 of 32

Introduction

The R400 will be the high end standalone graphics chip product when it is introduced. It will be followed very rapidly with two variants: The RV400, aimed at the volume PC space The R450, aimed at a volume high end market. The targets for the three chips are:

Part	Clock Speed	pixels/clk	texture fetches/clk	alu ops/clk	Memory width	Memory speed	die size	Tapeout
R400	400 MHz	8	16	32	256	400MHz	11.5	July,2002
RV40 0	500 MHz	4	8	16	128	500 MHz	8.5	Nov 2002
R450	500 MHz	8	16	32	256?	500 MHz	9.5	Feb 2003

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

AMD1044_0257998

ATI Ex. 2109 IPR2023-00922 Page 288 of 326



1. Features

1.1 AGP 8x

The chip will support the 32 bit AGP interface at speeds up to 8x. I expect that we will need to support AGP 1x and 2x which require 3.3 Volt I/0 (AGP 4x is 1.5v and AGP 8x is 750mv). AGP fast writes are supported for access to the frame buffer.

Open issue: 64 bit address space support.

1.2 256 Bit Memory Interface

The R400 and R450 support four memory channels, which can be 32 or 64 bits wide; the maximum memory bus width is a total of 256 bits. The RV400 supports two memory channels and a maximum total width of 128 bits.

All channels need to be configured identically, 1, 2 or 4 channels can be configured.

Memory standards supported:

1/0	Voltage	Memory type	Speed
SSTL2.5	2.5	DDR	100 to 500 MHz
SSTL1.8	1.8	DDR/infineon	300 to 500 MHz
Elpida	1.8 (1.5?)	Elpida	300 to 400 MHz
Infineon	1.2, 1.0 V	Infinion e-dram	500 MHz

No support for SSTL3.3, or SDRAM (LVTTL – 3.3V) is planned.

1.3 Unified Processing pipe

The most ambitious feature in this design is the "truly unified pipe": a single programmable pipeline is used for 2D, Video, 3D vertex, and 3D pixel operations. The unified pipeline does all of its calculations in 32 bit floating point, the same as the existing vertex transform in previous chip, and the next step in the precision of the color/pixel calculations which have increased from 8 bits (R100), through 16 bits (R200), to the 20 bits in the R300.

There is an area cost to the unified pipeline since we are forced to go to 32 bit precision for color, when application requirements may need less (22 to 24 bits). However the unified pipeline results in a single math/register structure compared to the separate structures in a more traditional design. It is hoped that by only needing to design the one structure we can make the investment in design time and effort to really optimize the area.

Some of the benefits to merging the pipelines include allowing the vertex operations to do texture fetches, which we could not afford add logic to the transform pipe to do, a single programming model for both operations, more precision on color than we would normally provide, and the ability to support significantly more registers and instructions in pixel shaders.

One important benefit is load balancing. In the current pipeline when the app it transform bound the pixel pipeline is idle some significant portion of the time, and when the app is raster bound the transform hardware idle. The unified pipeline presented here dynamically allocates its processing power between transform and raster.

1.4 Front end scaling

We will remove the back end scaling capability from the display, and replace it with a non-scaling overlay. This will require us to be able to implement scaling using the unified pipeline. Key features that will need to be supported are large filter kernels, de-interlacing, frame rate conversion, and good support for YUV and color conversion.

1.5 Real-Time drawing command ability

To allow for the emulation of backend scaling as well as support new features we need to be able to interrupt the 3D pipe and be able to execute high priority commands with low latency. The point of interruption is in the primitive

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

AMD1044_0257999

ATI Ex. 2109 IPR2023-00922 Page 289 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
11 March, 2001	4 September, 2015	R400 Top Level Spec	8 of 32

assembly, the maximum latency will be about the time it takes to render 4096 pixels. The real time commands are inserted into the 3D pipeline after transform, clipping, and setup. Those function need to be performed by the driver. There are also limits on the number of constant registers available.

1.6 3D Features

There are a number of new 3D features we are considering for inclusion. Additional features may be added, and some of these may be dropped.

1.6.1 Noise Textures

Perlin style noise is useful for a number of applications. It is generated on chip and consumes no external memory bandwidth. It also larger than any physical texture can be: 256x256x256 lattice points, and still has detail when the resolution is 4Kx4Kx4K. There is an opportunity to get this adopted as part of dx9.

1.6.2 Shadow buffers

John Carmack is using shadow volumes to generate shadow effects in doom3. Shadow volumes are very poor way to use modern 3D pipelines. (will add more detail here later). Shadow buffers have two key limitations: very high resolutions are required to avoid aliasing, and traditional shadow buffers can not be mip-mapped so filtering is real problem. We are able to solve the first problem through a combination of our improved anti-aliasing Z compression, and a new method of implementing the shadow map probe.

1.6.3 Sort Independent Transparency

We are currently looking into how best to support sort independent transparency. The two plans are either the dual Z buffer approach, or the approach described in <need to decide where the email should be placed so others can see>

1.6.4 Anti-Aliasing

The changes from the R300 include an increased number of samples per pixel, probably eight, and support for an allocated frame buffer size smaller than the worst case maximum.

1.6.5 Texture compression

To further reduce bandwidth we need to improve texture compression. We need to achieve both better compression that S3TC, and have a high enough quality that textures that would lose too much detail with S3TC can be compressed. Both of these goals do not need to be achieved simultaneously on all textures. We also need to look at compression of non-traditional surfaces such as normal maps. Advances here are dependent on the availability of resources to work on this. If we are unable to find resources we will support the s3tc compression currently in D3D.

1.6.6 Z compression

<larry needs to give me a paragraph here>

1.6.7 Texture Filtering

The texture pipes can fetch a 2x2 region from the texture map and filter it.

The data per pixel can either be four eight bit values, two sixteen bit values, or one 32 value. All data needs to be fixed point.

Linear filters are completely built in, and it takes 1 cycle for bi-linear, 2 for tri-linear, four for quadra-linear (filtered mipmapping of volume textures). Variable depth anisotropy is supported in hardware with the texture pipe calculating the number of samples needed. Optionally the pixel shader can calculate the number of samples, and how to increment the texture address, and provide this to the texture pipe.

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15} 12:48 PM

AMD1044_0258000

ATI Ex. 2109 IPR2023-00922 Page 290 of 326

<u>A</u> P	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	11 March, 2001	4 September, 2015	GEN-CXXXXX-REVA	9 of 32

1.6.8 Curved Surface Support

We will support curved surfaces through combination of vertex shader code and a tessellation engine to generate new vertices.

The tessellation engine generated new vertex indices from a input vertex index array. The new indices contain both the coordinate in parametric space of the vertex, and the indices to the surface, or to data from which the surface can be derived. More information is available in the programming guide.

1.6.9 *Displacement maps*

The tessellation engine for curved surfaces can dice triangles into micropolygons, the vertex shaders for the vertices can then access into a displacement map and change the location of the points.

1.7 High color depth

We will support a 64 bit color buffer (16:16:16:16), we will support two formats: sRGB64 and a floating point format.. <need to insert format details.

2. Performance

The basic performance is:

R400 MHz fill rate bi-linear equiv peak tri/sec

	MHz	Fill rate	Bi-linear texture fetches	Peak tri/sec
R400	400	3.2 gigapixel	6.4 Billion	400 Million
RV400	500	2.0	4.0	500 Million
R450	500	4.0	8.0	500 Million

Under normal conditions, and when not further limited by memory bandwidth we expect to be > 75% efficient.

3. <u>Schedule</u>

	Tapeout	Samples	Production
R400	July, 2002	Oct, 2002	Dec, 2002
RV400	Nov, 2002	Jan, 2003	March, 2003
R450	Jan, 2003	April 2003	May 2003

4. Process

At the moment this looks like an easy choice: .13 will be in production for over a year, and .10 does not show up until the very end of 2002 according to the TSMC and UMC roadmaps.

We will probably want to be in a flip chip packaging approach to meet power distribution goals. With the 256 bit bus we will have at least 600 signal I/O's (404 in memory). We may be as much as 10A at 1V for average power, which will require very good power distribution, area bond flip chip is probably the only option.

5. General Chip operation

5.1 Unified Shader

The unified shader is a simd/vector engine that performs the same instructions on four sets of four (16 total) elements. For pixel shader operations the elements are pixels with the sets of four required to be 2x2 footprints. For

Exhibit 2041.DCC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

AMD1044_0258001

ATI Ex. 2109 IPR2023-00922 Page 291 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
11 March, 2001	4 September, 2015	R400 Top Level Spec	10 of 32

vertex shader operations the sixteen elements are sixteen vertices. The basic element is a 4 value vector – frequently interpreted as x,y,z,w or r,g,b,a.

The user model for the unified shader is composed of a variable number of general purpose registers, a subset of which are usually initialized with data. An ALU can do simple math, conditional moves, and permutations on the registers, and the ability to do a limited number of memory reads using the texture cache. The number of register is variable, and the number of registers required for an operation are specified when the task is submitted to the unified shader. The unified shader will not start the task until there is enough free room for the task's registers.

The unified shader is based on the R300 pixel shader.

5.2 3D Rendering

For 3D rendering data is passed twice through the unified shader- once to transform the vertices and a second time to determine the color of the pixels.

The input to the 3D pipe is expected to be indexed vertex arrays. Linear vertex arrays can easily be supported by the CP generating sequential indices. Inline vertex data is an open issue, I would prefer to write it to memory and then fetch it as a vertex array rather than add a direct path.

The stream of indices is sent to the Primitive Assembly block by the CP. The front of the primitive assembly block maintains the tag for the vertex cache; The vertex cache stores transformed vertices. As misses are detected in the tag, the indices that miss are placed into 16 entry vectors. Each vector contains a state pointer, a pointer to the vertex shader to be used, and the 16 indices to vertices that need to be transformed. When either a vector is filled with 16 entries or a state change happens (so that the next vertex does not share the state and vertex shader with the previous vertex) the vector is issued to one of the "shader" pipelines for transformation. Which of the four shader pipelines it is issued to determined either by some effort of load balancing or a simple round robin. All that is submitted to the pixel pipeline is the state, the vertex program, and the indices. The shader pipeline will fetch the vertex array data through the cache infrastructure that is also used for texture fetches. After the tag the indices (actually now the indices into the vertex cache) are placed into a latency FIFO to hide the latency of transforming the vertices.

The shader pipeline receives the vector of 16 indices from the primitive assembly block. The shader pipeline operates, when rendering pixels, by processing a vector of four 2x2 pixel footprints, a total of 16 pixels. For vertex processing each of the pixels is replaced with a vertex. The vertex program includes information of how many local variables it will need. The rasterizer waits until that many local variables are free, (as each executing thread in the shader pipeline terminates it frees its local variables). With the proposed shader data path the maximum number of local variables per vertex is 256. However this leaves no ability to hide latency, 16 to 32 local variables will probably maximize latency hiding and therefore performance. The vertex shader program can use all the capabilities of the shader pipeline including texture fetches and dependent lookups. At the end of the vertex program, the transformed coordinates must be output. One output will be the x, y, z, w position which we be stored in the position cache of the vertex cache. The vertex program may also output a number of parameter values (colors, texture coordinates, other interpolated inputs into the pixel shader). The parameter values must be output as a multiple of four 128 bit words, as the parameter cache is designed for this.

The primitive assembly block reads the indices back out of the latency FIFO and accesses the position cache portion of the vertex cache. It assembles the vertices into primitives (lines, triangles, rectangles, quads?, points, ?). Baricentric values are assigned to the vertices, and will be used later in the rasterizer to interpolate the parameters. The parameters are not accessed by the primitive assembly logic, which only works from the position data. The primitive is clipped against both the viewing volume as well as user clip planes, with fractional baricentric coordinates assigned to the clipped primitive sections. The primitive goes through the perspective divide and the viewport transform. The resulting screen space primitive is setup (plane equations for 1/W, Z, and the baricentric coordinates). The resulting primitive data, including the indices back into the parameter portion of the vertex cache are broadcast to the four pipes. The final time that an index is output that access the oldest vertex cache line, a token is also sent. When all of the four pipelines return the token the primitive assembly block can free that cacheline and allow it to be used for a new vector of vertices. The performance goal in the primitive assembly block is a triangle every two clocks. An alternative option is for the vertex shader to generate screen coordinates and clip codes. If a primitive needs to be clipped, which can not be determined until primitive assembly, then the vertices are reverse transformed back into clip space by logic in the primitive assembly block, clipped, and then transformed back into screen space.

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15} 12:48 PM

AMD1044_0258002

ATI Ex. 2109 IPR2023-00922 Page 292 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
11 March, 2001	4 September, 2015	GEN-CXXXXX-REVA	11 of

To help meet marketing BS numbers we can look into doing backface culling at a rate of one triangle per clock. This will boost us to peak bs number of 500 million triangles per second.

Each pipe has a FIFO in front of the rasterizer to load balance. Each pipe will handle 16x16 tiles of the screen which are interleaved between the pipes. To maximize the effective size of the FIFO we will probably cull the triangle list before the FIFO. The rasterizer will request the parameter data from the parameter cache for the primitives. A small latency hiding FIFO will hide the latency of the access to the parameter cache. The parameter cache is 512 bits wide, and the interfaces from the parameter cache to the rasterizer are 128 bits wide, this allows the parameter cache to output one pipelines request per clock, which is serialized over four clocks, keeping all four interfaces busy. The rasterizer keeps a small cache of three to four vertices, this allow only the new parameter to be fetched when adjacent triangles are processed. The parameter cache interface imposes a second performance limits, in the worst case each polygon covers all four pipelines and there are no vertices shared from triangle to triangle. In this case the peak performance is (500 MHz / (4 pipelines * 3 vertices) = (500/12) = 41.6 million triangles per second. In the best case triangles are perfectly stripped and never cross over pipeline boundaries. In this case the peak performance (If we ignore the setup limit) is 500 million triangles per second. As a practical manner we should be able to approach the setup limit of 250 Million triangles per second.

The rasterizer also contains a portion of the hierarchical Z memory. We are looking into moving this into a cache based approach, but that is far for certain at this point. We would like to be able to do hierarchical z culling at a speed in excess of 64 pixel per clock per pipelines (256 pixels per clock total). We are also going to consider some of the improved latency hierarchical Z options to improve culling efficiency.

The rasterizer will generate four pixels per clock if there are no more than eight interpolated parameters. The rasterizer generates vectors of four 2x2 footprints (16 pixels). Each 2x2 footprint must be screen aligned and from the same triangle (with a single shared z slope). The four footprints only need to share the same state and shader program.

Before starting the processing of a vector the rasterizer (which includes the sequencer for the shader pipeline) checks to make sure that there are enough free registers in the shader pipeline for the pixel shader program. If not, it stalls until there are enough. The rasterizer also needs to arbitrate between the three streams of vectors to be shaded: the vertex stream, the pixel stream, and the real time stream. I think it will be sufficient for the real time stream to have priority over the vertex stream which has priority over the pixel stream. This will meet the real-time demands, and keep the vertex cache filled.

The vector is then processed by the shader pipeline. We will probably support up to eight sequentially dependent texture fetches. (to use the R300 terminology, eight clauses). 16 (8?) textures are supported, but each texture can be accessed multiple times by a single pixel shader which can provide a different address each time. This is especially useful for complex filters.

The output of the pixel shader is the final color of the fragment. The pixel shader may also replace the Z value. Fog and stippling must be done in the pixel shader program.

The render backend does the z compare, stencil operation and color alpha blend.

The texture fetch path has a number of design options. One option is an approach where the local, multiported, texture cache is small (1 to 4 KB), and contains uncompressed color in a canonical format (32 bits per pixel) and uses a 4x2 or 4x4 cacheline. This is backed up by a large (>16KB) L2 cache which also stored uncompressed 8x8 cachelines. The decompression logic lives between the memory controller and the L2 cache.

An alternative design uses the L2 cache to contain data in memory format (compressed) which is decompressed as needed to fulfill L1 texture cache misses. This will increase the effective size of the L2. The L2 cache is distributed, with 1/4 of it residing in each memory controller. The Texture decompression logic can either be located in each shader pipeline, or exist as a shared block(s) that receive data from all four memory controller and send the decompressed 4x4 cachelines to each shader pipeline. The unified decompression block will result in better performance, and possibly less area, at the cost of some of the scalability.

Assuming that we chose the L2 in memory controller and the unified decompression logic, the texture path would work as follows:

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
11 March, 2001	4 September, 2015	R400 Top Level Spec	12 of 32

In a four pipeline design there are two texture decompression blocks, one for the "left" texture units in each shader pipeline, and the second for the "right" texture units. In the two pipeline, lower cost, version of the chip only a single decompression pipeline is used, serving the left and right texture units.

The L1 texture cache receives a texture request from its shader pipeline. The usual tag and latency FIFO is used to generate the misses. These are sent to the shared texture decompression block, which looks up the texture to find the physical address and then sends the request to the L2 cache in the memory controller. The L2 also has a latency FIFO and tag, and will return the data in order (but there is no order guaranteed between the data returning from each L2). The decompression block has a buffer which is used to place the data from the memory controllers back in order. The decompression logic decompresses the texture and returns, in order, the 4x4 cachelines that the L1 caches are requesting. Most of the compression techniques we are considering are based on an 8x8 tile (or 4x4x4), when necessary the decompression logic will decompress an entire 64 pixel tile and only return the requested 16 pixels to the L1 cache. This will tend to increase the bandwidth between the decompression logic and the L2 cache as 8x8 blocks are repeatedly requested to provide different 4x4 subtiles to the L1. The L2 cache will prevent the repeated reads from going to memory, and we will probably implement an "L0" style cache in front of the L2 to also catch the redundant requests.

Each memory controller will have two 64 bit read return buses, one to each of the two decompression blocks, each decompression blocks drives a separate 128 bit bus to each of the four shader pipelines. This will tend to have better utilization and load balancing than having the memory controller drive a 32 bit bus to the decompression logic in each shader pipeline. While the total number of wires is similar (128 bits per memory controller, 128 bits into each texture cache) we are less likely to leave the texture pipes starved when there is some imbalance.

5.3 Real Time Rendering

The real time rendering interface allows primitives to be inserted into the rendering pipeline at a very late stage, therefore providing very low latency. The expected use is for scale blits timed by the display refresh, this suggests a small number of large primitives. We take advantage of this to simplify hardware by forcing the interface to be post setup, a real-time primitive needs to be transformed and setup by software.

Real time primitives also do not have access to the state management hardware used by non-real-time 3D commands. A single set of state registers, some constant registers, and one full parameter set is available. The real-time command stream will generally need to wait for the current real-time drawing operation to complete before it can start the next real-time command. The driver can statically allocate some of the physical constant registers to the real-time stream, these are not available to the RBBM for renaming use, and are written by the real-time command stream will generally need to non-real-time commands, for normal operation it is entirely managed by hardware. For real time rendering there will be dedicated space for three vertices, each with sixteen 128 bit interpolants. If the real-time primitive requires more than eight interpolants there will only be enough room for one primitive at a time, even if they need the same state and constants, if less than eight interpolants are needed then there is room to manually double buffer the interpolants, and allow pipelining of primitives. The real time command stream will still need to manually check that the pipeline has finished with the previous primitive, before writing new data to the parameter memory for the next primitive, while the pipeline works on the current primitive.

For example, the a drawing command in a real-time command buffer might look like this:

Wait for realtime pipe idle // make sure no real-time command is in the pipeline Write state reg m in context 7 with data // set rendering state for command Write state reg m in context 7 with data // set rendering state for command Write state reg m in context 7 with data // set rendering state for command Write state reg m in context 7 with data // set rendering state for command Write const reg at physical address k // write constant register Write const reg at physical address k+1 // write constant register Write const reg at physical address k+2 // etc. Write vertex 0, parameter 0, in real time parameter store Write vertex 1, parameter 0, in real time parameter store Write vertex 2, parameter 0, in real time parameter store Write setup primitive to primitive assembly (scan converter) Write initiator register, tag command with 0 Write vertex 0, parameter 8, in real time parameter store Write vertex 1, parameter 8, in real time parameter store

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
11 March, 2001	4 September, 2015	GEN-CXXXXX-REVA	13 of	

Write vertex 2, parameter 8, in real time parameter store

Write setup primitive to primitive assembly (scan converter) // this assumes we double buffer the primitive registers Write initiator register, tag command with 1 Wait_for_realtime_command_0_not_in_pipe Write vertex 0, parameter 0, in real time parameter store Write vertex 1, parameter 0, in real time parameter store Write vertex 2, parameter 0, in real time parameter store Write setup primitive to primitive assembly (scan converter) Write initiator register, tag command with 0 Wait_for_realtime_command_1_not_in_pipe Write vertex 0, parameter 8, in real time parameter store Write vertex 1, parameter 8, in real time parameter store Write vertex 2, parameter 8, in real time parameter store Write vertex 2, parameter 8, in real time parameter store Write vertex 1, parameter 8, in real time parameter store Write vertex 2, parameter 8, in real time parameter store Write vertex 2, parameter 8, in real time parameter store Write vertex 1, parameter 8, in real time parameter store Write vertex 2, parameter 8, in real time parameter store Write vertex 2, parameter 8, in real time parameter store Write vertex 2, parameter 8, in real time parameter store Write setup primitive to primitive assembly (scan converter) // this assumes we double buffer the primitive registers Write initiator register, tag command with 1 Wait_for_realtime_command_0_not_in_pipe

5.4 State Management

State management differs from previous ATI chips.

There are eight sets of state registers in the chip. Each pixel or triangle is tagged with which state it is supposed to use. Most of this is hidden from the programmer by the RBBM, which implements the in-order semantics that are normally used. States 0 to 6 are managed by the RBBM for high performance 3D/2D/video rendering. State 7 is reserved for real time commands, and the real time command stream must ensure that the state is not changed while the pipeline is active.

Each register is therefore mapped in to the register space nine times: once for the current state, plus eight additional times to provide access to all existing state. This is only true for the normal pipeline state registers, the constant registers used by the pixel/vertex shaders are handled by a separate, related mechanism.

There are two options for the update of the state registers. The first option is to implement a broadside state copy, which copies the contents of the previous current state to the new current state before the first state write happens to the new state. This is somewhat costly in hardware. The second option is for the state updates done by the driver to be "complete", write the minimum set of state registers that completely defines the new rendering state, this avoids the need for the hw broadside copy.

The constant registers are implemented using a renaming scheme that avoids the need to do a broadside copy when changing state. It also does not use storage for each state, when two state contexts have the same value in the same register, the renaming logic points them at the same physical register.

Since the registers that are most frequently changed are located in the constant memory of the R400 (vertex array pointers, and texture pointers) we may wish to separate updates to the constant registers from general state register updates.

5.5 Bad Data

Bad data can exist for a number of reasons. When a vertex shader does an access to an address which is not permitted (or does not exist) we need a way to avoid hanging, and make debugging possible; A similar issue exists for pixel shaders that do bad texture accesses.

We currently handle a limited form of this: a triangle than contain a vertex which contains (or generates) a NaN or INF is not drawn, it is simply culled at setup.

We will extend this as follows:

For a vertex fetch that goes out of range (or times out) a flag in the vertex is set which will cause that vertex to be treated as if it contained a NaN. A debugging flag will also be set, and if we can find an easy way to do it, the index of the offending vertex will also be stored.

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

AMD1044_0258005

ATI Ex. 2109 IPR2023-00922 Page 295 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
11 March, 2001	4 September, 2015	R400 Top Level Spec	14 of 32

For a texture fetch a similar strategy will apply: A bad access will set a flag that will cause the pixel to be dropped. The debugging mode will force the pixel to pass the Z test, and override the color output from the pixelshader with an ugly shade of green.

5.6 Display operation

The display must be able to display from microtiled surfaces and overlays. This will generally force us to adopt line buffers.

The display should support at least two outputs, ideally we will be able to support two high resolution outputs and a low resolution output (TV out)

We will drop support for overlay scaling, and therefore supporting an overlay on all displays becomes affordable, fixing a "bug" that our current dual display products suffer from.

The memory for the line buffers is shared with the L2 texture cache. This allows use a memory size that is closer the maximum requirement of either function, instead of the sum of the maximum requirement.

The maximum resolution color format is 64 bit color for the primary surface and 32 bit color for the overlay For two 2560 pixel wide line buffers we need

2560 pixels2560two lines2two displays296 bits of color96(32 overlay + 64 bits primary)total bits960K bits, 120 Kbytes

The L2 memory will probably be 128Kbytes, which will leave only 8KB for the texture L2 cache when driving the above display. However, the above case is driving two multi-megapixel displays with the worst case color depth. It works, but 3D performance suffers.

A slightly more normal case might be two 1600x1200 displays, with the same color depth:

2560 pixels1600two lines2two displays296 bits of color96(32 overlay + 64 bits primary)total bits600K bitswhich leaves >54Kbytes for the L2 Cache

A benchmark case, one display no overlay, 32 bit color: 2560 pixels 1280 two lines 2 two displays 1 96 bits of color 32 (32 overlay + 64 bits primary) total bits 81K bits which leaves >110Kbytes for the L2 cache.

The line buffers are two scan lines high:

At beginning of an even scan line (scan line 0) 1/2 of line buffer is filled. The upper half of the buffer is read out, at the same time as the second half of the buffer is filled. When the scanout reaches the midpoint 3/4 of the buffer is filled. When the scanout reaches the end of the scan the buffer is filled. The speed at which the buffer is filled must be greater than 1/2 of the rate at which the buffer is scanned out.

For the odd scanlines, the buffer is completely filled at the start of scanout (as a result of the even scanline finishing properly). As the lower half is scanned out, reads are issued to fetch the data for the next pairs of scanlines. At the end of the odd scan line, the buffer is expected to contain half of the data for the next scan line.

Another way of looking at this is as follows:

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

AMD1044_0258006

ATI Ex. 2109 IPR2023-00922 Page 296 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
11 March, 2001	4 September, 2015	GEN-CXXXXX-REVA	15 of

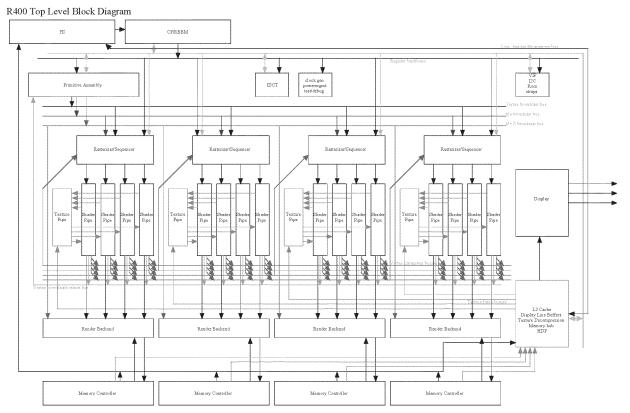
At the beginning of the odd scan line the scan buffer is filled. As each word is read from the buffer and sent to the display logic, a request is made to the memory controller to fill in the data. It is not necessary for all the data for the next scan line pair to be fetched by the time that the scan line reaches the end, the real requirement is for the last word in the scan line buffer to get there just before it is read, at the end of the next even scan line.

The display lives mostly in the core clock domain. There is a FIFO per pixel clock that crosses into the DAC/TMDS clock domain.

If we are able to implement the time interleaved display block then the display will merge and color convert two pixels per clock in the core clock domain. Whichever display FIFO is closest to empty will get the priority to be filled in the next time slot. The sum of the pixel clocks (display0, display 1, tvout) must be less than 2x the core clock. We should be able to cheat slightly and use some of the horizontal retrace time to fill the display fifo's, this will relax slightly the 2x core clock limit.

Since the 3D pipe is capable of real-time events, such as display triggered scale-blits, we may wish to reconsider the location of several operations that are currently in the display. TV out scaling, and ratiometric scaling for LCD panels may be more cheaply implemented using the 3D pipe instead of dedicated hardware.

6. Block Diagram



7. Blocks

HBIU – host bus interface unit CP – control processor RBBM – register interface manager CLK – clock generator TC – test controller VIP – video input port

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:43 PM

AMD1044_0258007

ATI Ex. 2109 IPR2023-00922 Page 297 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
11 March, 2001	4 September, 2015	R400 Top Level Spec	16 of 32	

ROM – boot rom

I2C – I2C interface

DU – Display

MH – Memory Hub

- HDP Host Data Path
- IDCT Mpeg decoder
- PA Primitive Assembly
- TD Texture Decompression
- RE Raster Engine
- SP Shader Pipe TP – Texture Pipe
- RB Render Backend
- MC Memory Controller

The blocks are combined into a smaller number of blocks for layout:

Layout block	subblocks	Instances R400/450	Instances RV400	Notes
HI	HI	1	1	
CP	CP RBBM CLK Reset	1	1	
Misc	VIP ROM I2C TC	1	1	
DU	DU	1	1	Display
TD	TD MH HDP	1	1	L:2 Cache
PA	PA	1	1	
RE	RE	4	2	
SP	SP	16	8	
TP	TP	4	2	
RB	RB	4	2	
MC	MC	4	2	

8. Block descriptions

8.1 HBIU - host bus interface unit

The HBIU interfaces the graphics chip to the system AGP bus.

8.1.1 Description

The HBIU implements the following buses: PCI slave PCI master AGP fast writes AGP reads AGP writes

64 bit support?

8.1.2 Major interfaces

The following busses connect the HBIU to the rest of the chip:

Bus Chip client Bus client Description
--

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15} 12:48 PM

	ORIGINATE DATE	EDIT DATE		DOCUMENT-REV. NUM.	PAGE	
	11 March, 2001	4 September, 20	015	GEN-CXXXXX-REVA	17 of	
 Host register	CP/RBBM	PCI Slave	CPU	reads and writes to chip registers		
Host Data		PCI Slave AGP fast write	CPU	reads and writes to video memory		
AIC Write		PCI Master (writes) AGP writes	Prima writes	rily blits to system memory, and contro	l semapho	re
AIC Read		PCI Master (reads) AGP reads	PA in State/ Verte:	M4 reads dex reads vertex program loading k loads texture		
Oddites for VG	A					

8.1.3 Block diagram

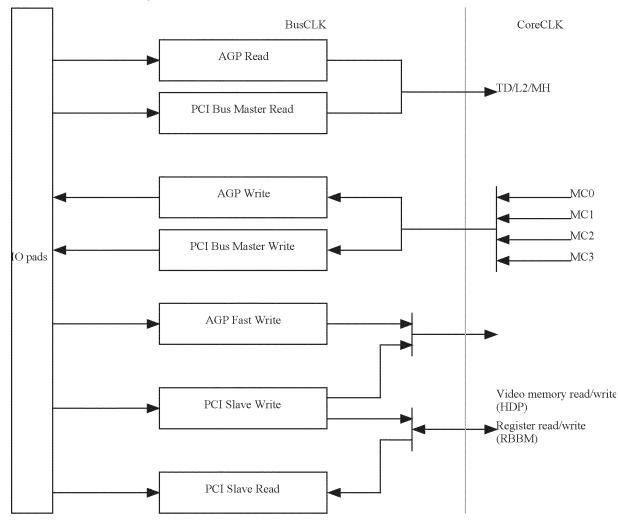


Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

AMD1044_0258009

ATI Ex. 2109 IPR2023-00922 Page 299 of 326



8.2 CP – control processor

8.2.1 Description

The control processor executes the pm4 display list from memory, driving the operation of the rest of the chip. It also implements the real-time event commands.

Currently the CP is based on a custom processor, which has a very limited instruction set and is really only capable of executing the existing program. It is not expected to be capable of doing the translation of 2D packets to the preferred hardware interface, or be able to implement the real time commands.

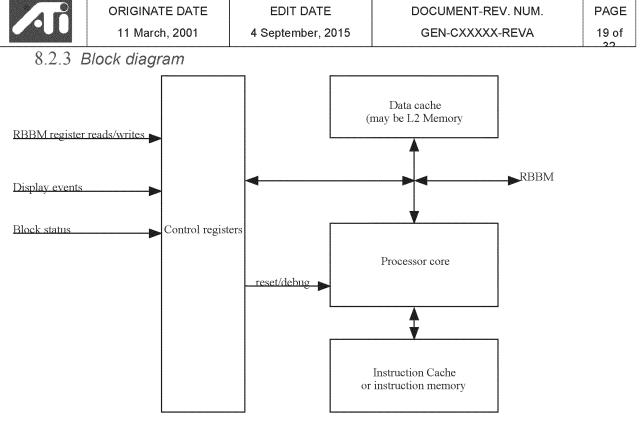
An alternative is to base the CP on a more generic RISC processor. It appears that this will save area, and make it possible to write the CP control program in C. The ARC core, for example, is less than 20K gates.

One key change that enables us to consider a processor core instead of the custom PM4 engine is that data is no longer embedded in the command stream. In the R128 to R300 index, vertex, and host-blit data is embedded in the primary ring buffer and the indirect buffer. In the R400 index data is fetched by a dedicated DMA engine in the PA block, and vertex and host blit data is fetched through the texture cache. This allows us to optimized a single path for the data rather than need to optimize both the DMA and PM4 paths. With the CP no longer needing to be able to copy data at 32 bits per instruction (read and write), a less specialized processor can be used.

8.2.2 Major interfaces

Bus	Description
RBBM->CP	Register read write,
	Used for reset and debugging of CP, and access to control registers
CP→RBBM	Register writes, and reads
	Register access that occur as a result of executing the control program
СР→МН	Memory reads and writes.
	Read PM4 buffers, write semaphores to communicate with driver
Display→CP	Source of real time events to trigger real time commands, also delays in command queue based on
	display status. Current scan line is most common type of data
All block→CP	Blocks status.
	Used for wait for idle and power down

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM



8.3 RBBM - register interface manager

8.3.1 Description

The RBBM of the R400 is vastly simplified compared to previous versions.

The key differences are:

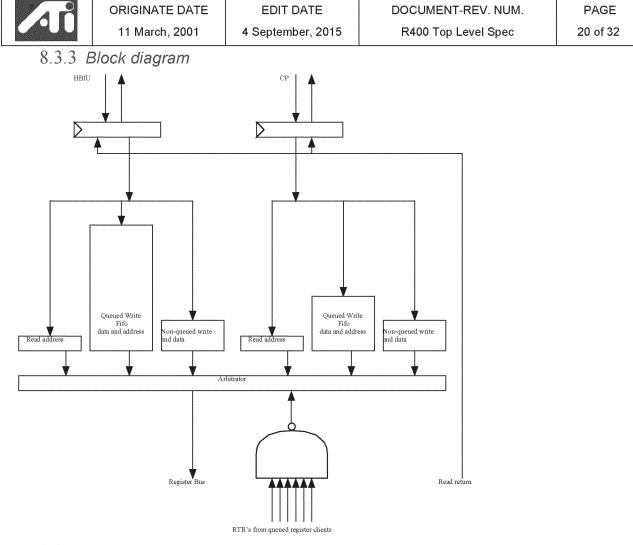
- 1) A much simpler register decoding scheme that does not need the RBBM to be aware of autoreg files.
- 2) A simpler register bus protocol that (for most registers) does not involve any feedback signals to the RBBM
- 3) Support for simple pipelining of the register bus to meet timing goals.
- 4) Much of the synchronization logic that was in the RBBM is now the domain of the CP, this means that bypassing the CP is not a viable production driver mode, but it really is not viable now.
- 5) Power Saving needs some adjustment (since the RBBM is no longer aware of when a block is activated.
- 6) All register bus connections are now single cycle, register to register which will simplify timing.

However the management of state changes has been moved from the blocks in the 3D pipe to the RBBM. The RBBM detects when a state block is no longer in use, tracks the blocks that are not is use, and allocates them to new primitives as needed.

8.3.2 Major interfaces

Bus	Description			
HBIU→RBBM	Register read/write			
CP→RBBM	Register writes resulting from interpretation of command packets			
RBBM-register bus	The purpose of the block			
RBBM→CLK	Power management			
RBBM→all	Soft/hard reset			

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM



8.3.4 RBBM operation

This is copied from the current RBBM spec, at some point most of it will be moved back there.

The RBBM has merges register writes and reads from the HBIU and the CP and broadcasts them to the rest of the blocks in the chip. That is all it needs to do.

Registers can either be queued or un-queued. In general queued register writes are initiator registers, or order critical state registers. The RBBM distinguishes between the two types of registers by their address only. The upper ?Kbytes of the register space are queued registers, the remainder is un-queued.

Both the CP and the host can generate both types of register writes.

Un-queued register writes can and will pass queued registers writes. If it is important for un-queued register writes to be held off by a queued register write the host or cp must not send the un-queued register write until the host or cp has determined that the queued register write has completed (usually by a spin lock on a semaphore).

Queued registers are maintained in order from the viewpoint of each originator. I.e. all of the CP's queued writes will complete in order, and all of the hosts will complete in order. There is no ordering between the CP and host- the writes from both clients may become interleaved.

The global register bus is as follows:

Name	Direction	bits	Description	
	•			÷

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

	ORIGINATE DATE		EDIT DATE	DOCUMENT-REV. NUM.	PAGE	
	11 March, 2001		4 September, 2015	GEN-CXXXXX-REVA	21 of	
WE	RBBM→	1	Write enable, address and	l data are valid		
Addr[19:2]	RBBM→	18	Register address			
Wm[3:0]	RBBM→	4	Register write mask, shou	Id be ignored by most clients		
Wd[31:0]	RBBM→	32	Data			
RE	RBBM→	1	Read Enable, address is v	valid		
Rd[31:0]	→RBBM	32	Read data returned			
RRn	→RBBM	1	Read return strobe (active	low)		

The protocol for a write is simple:

On a rising edge , if WE is high then the data and address is valid.

There are no completion signals, there is no way to abort a write.

Handshake signals for queued registers will be described later and are separate from the register bus.

The read protocol is somewhat more complex.

A read request is sent out when RE is high. The address holds the address of the read request. RE and addr will only be valid for one clock cycle.

Some number of clock cycles later the RBBM will receive the return data back, when RR is low.

The read return "bus" (Rd and RRn) is the logical AND of all the clients that can respond to a read request. All clients but the client that is responding to the read request drive a logical 1 on the bus. The wiring of the read return bus is a tree of point to point connections, and each node one or more sub-busses are AND'd together, registered, and driven on to the RBBM. This is the same as an OR tree, but the signal is inverted. Since a read return cycle is surrounded by idle cycles the only critical transition is high to low for the Rd signals (possible timing help, at the cost of needing to tell static timing to ignore low to high transitions).

Only one read is outstanding at any time. Reads will pass queued writes, (should/can they pass all writes?) The RBBM will timeout on a read after 64? clocks. It is critical that no client respond latter than 64 clock as the RBBM may timeout on the read, issue another and interpret the very late response of the first read as the second read.

If a read times out the RBBM will return dummy data (such as '0xDEADBEEF') to the requestor and mark in a debug register than an error happened.

Both queued and non-queued register writes are broadcast on the same bus. To implement the queued registers the RBBM looks at the status of all of the RTR signals from the clients that contain queued registers. Only if all are high will any queued register be allowed to issue from the RBBM. Note that these signals are registered on the boundary of the RBBM, and the register bus is also registered. This means that there is at least a two clock latency responding to a RTR signal deasserting. Since the clients will also be registered this means that a client will receive four or more queued register writes after asking them to stop. It is the clients responsibility to have enough buffering so that no register writes are lost.

8.4 CLK – clock generator

8.4.1 Description

The clock generator block generates the many clocks used by the R400:

Clock	speed range	
AGP	133 to 533	AGP clock
Sclk	33to 500 Mhz	core clock
Mclk	33 to 500	memory clock
P0clk	10 to 450	pixel clock for primary display (5x faster for TMDS//LVDS)
P1clk	10 to 450	pixel clock for secondary display (5x faster for TMDS//LVDS)
Tvclk		pixel clock for tvout

8.4.2 Major interfaces

 Bus
 Description

 Exhibit 2041.DOC
 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15} 12:48 PM

RBBM→CLK control

- 8.4.3 Block diagram
- 8.5 TC test controller
- 8.5.1 Description
- 8.5.2 Major interfaces
- 8.5.3 Block diagram
- 8.6 VIP Video input port
- 8.6.1 Description

8.6.2 Major interfaces

Bus	Description
VIP→MH	DMA transfers
RBBM→VIP	Control

8.6.3 Block diagram

8.7 ROM – boot rom

On powerup the graphics chip reads the straps from the rom. The rom is then used for responding to boot rom read requests from the PCI bus. We will only support serial roms. The list of supported roms is TBD.

8.7.1 Description

8.7.2 Major interfaces

Bus	Description	
ROM→HBIU	Boot rom read interface	
RBBM→ROM	Flash/eeprom boot rom write interface	
ROM→chip	Decoded straps	

8.7.3 Block diagram

8.8 I2C - I2C interface

8.8.1 Description

The I2C bus is a 2 wire bus used to communicate with other multimedia devices (such as tv tuners)

8.8.2 Major interfaces

Bus	Description
RBBM→I2C	Read/write interface

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15} 12:48 PM



PAGE 23 of

8.8.3 Block diagram

8.9 DU – Display

The R400 display drives up to three displays: two monitors and a TVOUT.

The chip can have as much as two analog RGB DAC's, two dual channel TMDS outputs, and one dual channel LVDS output.

All support for the scaling overlay is removed. The display supports a non-scaling overlay on each display.

See display operation section above for more details.

8.9.1 Description

Hopefully we have the resources to move to the time interleaved display design. The following frame buffer formats are supported: Primary surface: 8bpp index 16bpp 4444,565.555 RGB 32bpp 8888 RGB 64bpp 16:16:16:16, either sRGB or the R400 floating point format

Overlay: 32bpp 8888 RGB 4:2:2 YUYV the color conversion for the overlay is controlled by a programmable matrix, so the choice of color space is arbitrary.

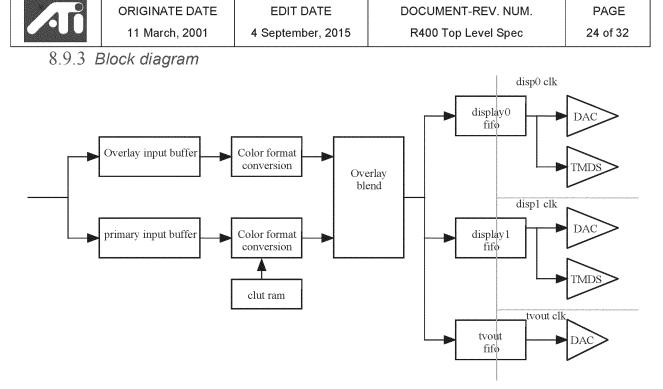
The maximum display pixel clock is greater than 400 MHz.

If we build the time interleaved design, then the maximum number of display pixels will be 2x the core clock speed. This will be divided among the two displays and tvout.

8.9.2 Major interfaces

Bus	Description
TD→DU	Memory read interface
RBBM→DU	Register writes/reads
DU→CP	Synchronization information

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM



8.10 MH – Memory Hub

8.10.1 Description

The memory hub acts as a switch between the many small clients of the memory controllers, and the two or four memory controllers. This allows most blocks to not have any dependencies on the number of memory controllers.

8.10.2 Major interfaces

The memory hub has a 32 bit read and a 32 bit write bus to each of the memory controllers. If we co-locate the MH and the L2 cache in the texture decompression block, then the 128 bit read return bus from the MC to the L2 cache can be used to return read data to the MH instead of a private bus.

The clients of the MH:

```
HDP
CP
VIP
PA- index buffer reading
RE- Vertex/pixel program loads, hierarchical Z
IDCT
?
```

8.10.3 Block diagram

8.11 HDP – Host Data Path

The host data path allow the host cpu to access video memory. It provides eight "surfaces" that provide endian and tiling translation, making the target area of memory look like a linear surface in the processors native endian.

The HDP also implements most of the legacy VGA functionality.

Exhibit 2041.DCC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

AMD1044_0258016

ATI Ex. 2109 IPR2023-00922 Page 306 of 326



8.11.1 Description

8.11.2 *Major interfaces*

Bus	Description
HI→HDP	Memory read/write requests to HDP
HDP→MH	HDP reads/writes to local memory

8.11.3 Block diagram

8.12 IDCT - Mpeg decoder

8.12.1 Description

The R400 uses the same implementation of IDCT/MPEG as the R300. This block decodes the compressed stream, placing the resulting IDCT data in one buffer in memory, and the motion vectors in another. The 3D pipe is then programmed to read the motion vectors and IDCT data and complete the decoding operation

8.12.2 Major interfaces

Bus	Description
IDCT→MH	Read command stream, write IDCT results and motion vectors

8.12.3 Block diagram

8.13 PA – Primitive Assembly

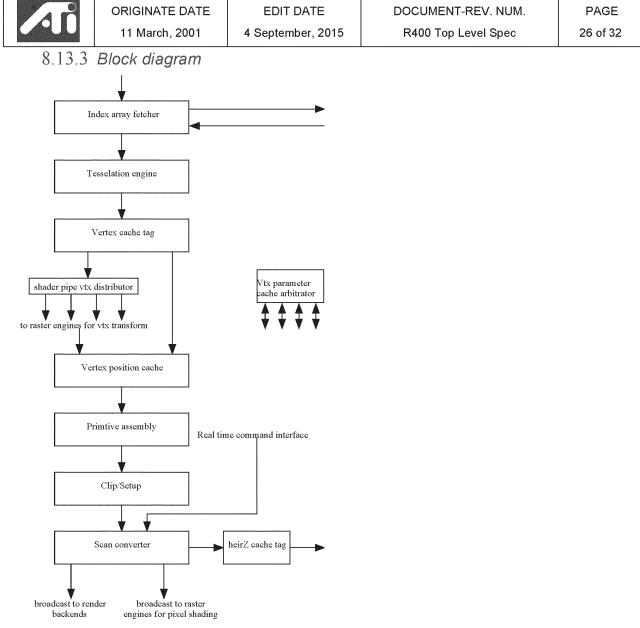
8.13.1 Description

The primitive assembly block fetches or creates the indices to vertices, possibly creating extra vertices with the tesselation engine. It determines which vertices have not been recently seen (and will therefore not be located in the post transform vertex cache), assembles vectors of sixteen vertices than need to be transformed, and submits them to a raster engine/shader pipe set to be transformed. It then receives the transformed vertex position data from the shader pipes. The vertex cache tag also outputs the sequence of cache addresses generated from the incoming indices. The primitive assembly subblock then creates primitives (lines, points, rectangles, triangles) from the vertices. It also implements the line counter for styled lines. The primitives are setup in the setup/clip block, but first clipped to the view frustum and optionally the user clip planes. The scan converted does a course walk of the primitive using an 8x8 grid. The scan converted also determines when the hierarchical Z data needed for culling will not be in the local hierarchical z cache in each rasterizers reads from the post transform vertex parameter cache, which is distributed among the shader pipes.

8.13.2 Major interfaces

Bus	Description
RBBM→PA	State changes, and initiator register writes
РА→МН	Index fetch path
PA→RE	Vertex transform packets
РА→МН	Hierarchical Z read request
PA→RBn	Coverage mask, position, and Z slope
PA→SPn	8x8 tiles to be rasterized

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM



8.14 TD – Texture Decompression

8.14.1 Description

The texture decompression block converts the memory texture formats into the uncompressed texture formats supported by the texture pipes. It consists of the L2 texture cache, the texture decompression logic, a set of output buffers, and the texture addressing logic.

The decompressed formats supported are:

32 bpp (8888) unsigned 32 bpp (8888) signed 32 bpp (16,16) unsigned and signed 32 bpp (32) unsigned and signed

we may also support a 8bpp mono format to improve the performance of shadow buffering.

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM



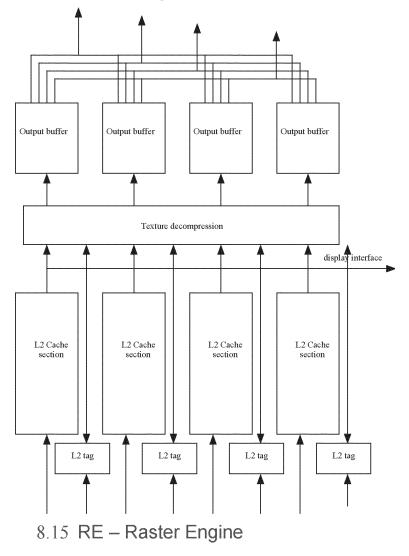
ORIGINATE DATE 11 March, 2001

PAGE 27 of

8.14.2 Major interfaces

Bus	Description
TP→TD	Texture requests and returned data
TD→MCn	Memory read requests
TD→Display	Data path for display which uses the L2 cache as its line buffers
MCn→TD	Invalidate snoop bus for cache coherency

8.14.3 Block diagram



8.15.1 Description

The raster engine performs two duties: it does the detail walk of 8x8 tiles of primitives, and it contains the sequencer for the shader pipe.

The shader pipe has the FIFO to allow for balance between the pipelines in the chip, it appears that this FIFO is 64 8x8 tiles deep. Only tiles that are owned by this pipeline are stored in the FIFO, others are immediately rejected. When an 8x8 tile is read out of the FIFO, it is checked against the heir-Z fail data that has arrived in the local heir-Z cache. If the primitive fails, it is rejected and the RB is informed that the tile has been killed.

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

AMD1044_0258019

ATI Ex. 2109 IPR2023-00922 Page 309 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
11 March, 2001	4 September, 2015	R400 Top Level Spec	28 of 32

We are going to support hierarchical Z object culling within the command stream. To support this we have the ability to draw a bounding object, heir-Z test it, but kill it before we rasterize it. The raster engine will receive tiles that are marked indicating that they are part of an occlusion query, and test them against the heir-z memory. All the tiles are rejected, but if any of them pass the heir-Z test then id then a flag is set. When the marker (which is an id) changes, indicating the end of this occlusion query, the RE will signal back to the primitive assembly if the flag was set or not.

If the tile passes the heir-z test we need to ensure that the parameter data needed to interpolate the triangle is either in the local I0 parameter cache, or on its way there. If not a request needs to be made to the arbitrator in the PA to get the needed data/

A FIFO on the output of the HZ cull and parameter cache tag buffers the passing tiles while waiting for the parameter data to arrive at the cache. It also provides buffering so that the rest of the pipeline can stay busy during a long string of tiles that fail the hierarchical Z test.

The next step is a detail walker that generates the coverage mask for each potentially covered 2x2 quad in the 8x8. We may need a path from this result to the render backend to aid in its determination as to what to fetch. The parametric coordinates are calculated, and used to driver the interpolator. We need to be able to do both perspectively correction interpolation and non-corrected interpolation.

The raster engine breaks the stream of pixels into 4 quad vectors (16 pixels) and will wait until the needed space is available in the shader pipe, and then start the sequencer running the pixel shader program.

The second part of the raster engine is the sequencer.

The sequencer first arbitrates between vectors of 16 vertices that arrive directly from primitive assembly and vectors of 4 quads (16 pixels) that are generated in the raster engine.

The vertex or pixel program specifies how many GPR's it needs to execute. The sequencer will not start the next vector until the needed space is available.

The sequencer is based on the R300 design. It chooses an ALU clause and a texture clause to execute, and execute all of the instructions in a clause before looking for a new clause of the same type. Each vector will have eight texture and eight alu clauses, but clauses do not need to contain instructions. A vector of pixels or vertices ping-pongs along the sequencer FIFO, bouncing from texture reservation station to alu reservation station. A FIFO exists between each reservation station can be chosen to execute. The sequencer looks at all eight alu reservation stations to chose a alu clause to execute and all eight texture stations to chose a texture clause to execute. The arbitrator will give priority to clauses/reservation stations closer to the top of the pipeline. It will not execute an alu clause until the texture fetches initiated by the previous texture clause have completed.

To support the shader pipe the raster engine also contains the shader instruction cache and constant store.

Bus	Description
PA(sc)→RE	Broadcast bus for 8x8 slices of primitives. I,J,K plane equations, front most Z for heir-Z culling, pointer for location of parameter data in vertex parameter cache
MH→RE	Returned hierarchical Z data for local cache.
PA→RE	Parameter request port
SC→RE	Returned parameter data
PA→RE	Requests to transform packets of vertices
RBBM→RE	State register reads/writes
RE→SC	Interpolated parameter data
RE→SC	Instructions, constants, register file addresses
RE→RB	Heir-Z pass/fail information
RE→RB	Sequencing information for availability of pixels
RE→PA	Sequencing interface for returning transformed vertices.
RE→PA	Occlusion query results
MH→RE	Shader I Cache fills

8.15.2 Major interfaces

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

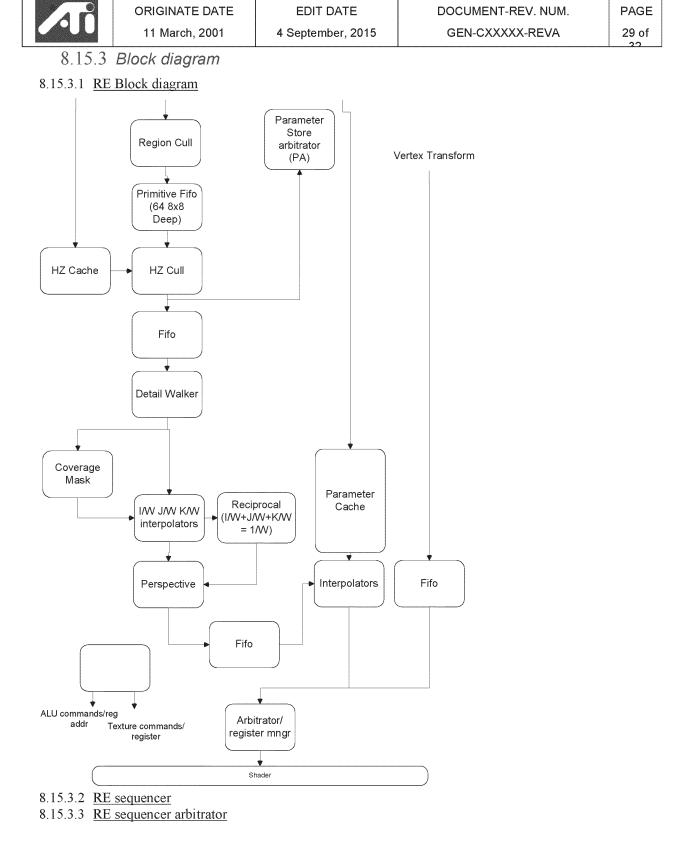
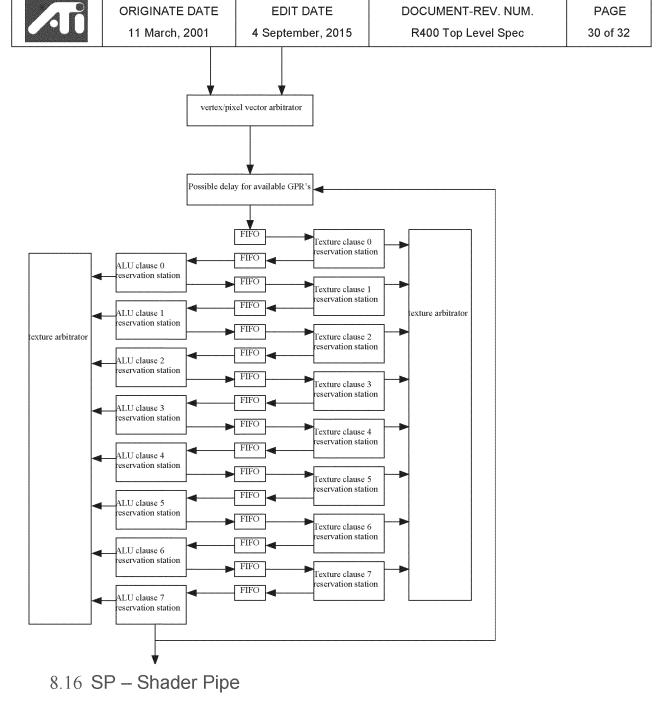


Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

AMD1044_0258021

ATI Ex. 2109 IPR2023-00922 Page 311 of 326



8.16.1 Description

The shader pipe implements the math pipeline of the R400. It has no sequencing/control logic; the control is located in the raster engine.

The shader pipe contains four floating point MAC's, and an

8.16.2 Major interfaces

Bus	Description
RE→SP	Interpolated data
RE→SP	Control

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

		ORIGINATE DATE	EDIT DATE DOCUMENT-REV. NUM.		PAGE
		11 March, 2001	4 September, 2015	GEN-CXXXXX-REVA	31 of
SF	P→TX	Texture requests + vertex parameters + pixels to render backend			
T)	≺→SP	Returned texture data			
R	E→SP	Constants			
SF	P→SP	Local w bus for derivative opcode			

8.16.3 Block diagram

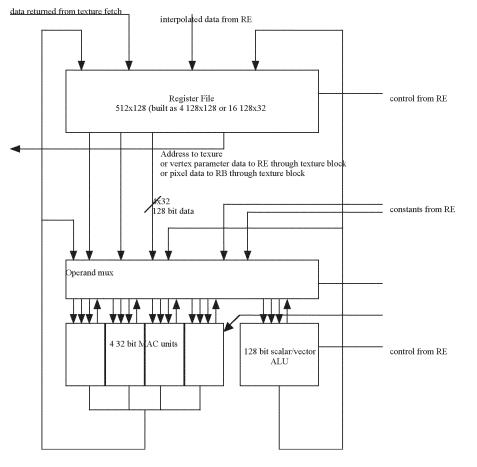


Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15} 12:48 PM

AMD1044_0258023

ATI Ex. 2109 IPR2023-00922 Page 313 of 326

- 8.17 TP Texture Pipe
- 8.17.1 Description
- 8.17.2 Major interfaces
- 8.17.3 Block diagram
- 8.18 RB Render Backend
- 8.18.1 Description
- 8.18.2 Major interfaces
- 8.18.3 Block diagram
- 8.19 MC Memory Controller
- 8.19.1 Description
- 8.19.2 Major interfaces
- 8.19.3 Block diagram

9. Common Foundations

- 9.1 Logic Design
- 9.1.1 Data formats

As much as possible, data should be stored and processed identically to x86 (or sparc) conventions. This will, for example, allow the emulator to use normal 32 bit floats and the processors native multiply, add and other operations. This will have a significant effect on the achieved simulation performance compared to being "almost" identical which requires the emulator take several operations to match the hardware bit-exact.

9.1.2 Register Bus

Issues:

32 vs. 64 bit Do rendering state updates happen on this bus or over a dedicated path from memory? Flow control

9.1.3 Block Communication protocol

We want to specify a limited number (one Is probably not possible) number of different ways that blocks are interconnected to simplify verification and emulation.

9.2 Software

Exhibit 2041.DOC 48154 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 12:48 PM

AMD1044_0258024

ATI Ex. 2109 IPR2023-00922 Page 314 of 326

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	23 January, 2001	4 September, 2015	GEN-CXXXXX-REVA	1 of 12
Author:	Andrew Gruber, Andi Sker	nde		£
ssue To:		Сору №	:	
		Shadar Draaa		
		Shader Proce	SSOF	
		ver 0.1		
	is document describes the ov s well as the timing of the sha		ders, interfaces, partitioning into func r use by hadware designers.	tional blocks
	CALLY UPDATED FIELDS:			
Document l Current Intr	Location : HD PC ranet Search Title: Shader P	Processor		
		APPROVALS		
	Name/Dept		Signature/Date	
Remarks:				
			NFORMATION THAT COL	
SUBST/			EREST OF ATI TECHNOL	JGIES
	INC. THROUGH	JNAUTHORIZED U	SE OR DISCLOSURE.	
			· · · · · · · · · · · · · · · · · · ·	
			erial in this document constitutes an provide notice that ATI owns a copy	
unpublished v	work. The copyright notice	is not an admission that	publication has occurred. This wo	ork contains
			t of this document may be used, rep ssion of ATI Technologies Inc."	produced, or
	-	- -		
Exhibit 2042.c	doc 16774 Bytes*** © ATI Confi	dential. Reference Copyrig	ht Notice on Cover Page © *** _{09/04/15}	^{5 04:44 PM} ATI 2
				LG v
				IPR2
				11 1(2
			A	AMD1044_02

ATI Ex. 2109 IPR2023-00922 Page 315 of 326



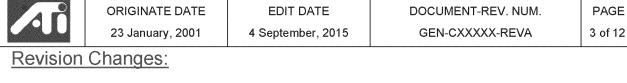
Table Of Contents

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	
4. TEXTURE/MEMORY	10
4.1 Instruction Format	
4.2 Opcodes	11

Exhibit 2042.doc 16774 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:44 PM

AMD1044_0258026

ATI Ex. 2109 IPR2023-00922 Page 316 of 326



Rev 0.0 (Andi Skende) Date: May 09, 2001 Initial revision.

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

Rev 0.2 (Andi Skende) Date: May 10, 2001 Initial revision Document started

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

A more detailed description of the SP ->RB interface as well as RE/Sequencer ->SP interface.

Exhibit 2042.doc 16774 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15 04:44 PM}

AMD1044_0258027

ATI Ex. 2109 IPR2023-00922 Page 317 of 326

0	0

PAGE

4 of 12

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.

Exhibit 2042.doc 16774 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:44 PM

AMD1044_0258028

ATI Ex. 2109 IPR2023-00922 Page 318 of 326



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 unavailable 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.

Exhibit 2042.doc 16774 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:44 PM

AMD1044_0258029

ATI Ex. 2109 IPR2023-00922 Page 319 of 326

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
23 January, 2001	4 September, 2015	R400 Shader Processor Model	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 simultainous 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 clase 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	В	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 accumution 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 neeed, but with mipcntl set to "lower mip level") R = Tt0 + texture(address, and rest of state neeed, 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.

Exhibit 2042.doc 16774 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:44 PM



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- texure is in range 1 to 0 after clamp mirror, must be multiplied by txwidth/txheight (power2) 1- texture has been multipled 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 12	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_typ	3	
е		
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.

Exhibit 2042.doc 16774 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:44 PM

AMD1044_0258031

ATI Ex. 2109 IPR2023-00922 Page 321 of 326



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 expxy, 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 apha 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

Exhibit 2042.doc 16774 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:44 PM

AMD1044_0258032

ATI Ex. 2109 IPR2023-00922 Page 322 of 326

	ORIG	INATE DATE		EDIT DATE	DOCUMENT-REV. NUM. PAGE	-
	23 Ja	inuary, 2001		4 September, 2015	GEN-CXXXXX-REVA 9 of 1	2
SRC C swizzle	Э	12	3 bits for each component			
Scalar/Alpha Mod	ha Output 3		Output modifier for the result of the Scalar or Alpha channel operation			
Vector Output	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 C	Scalar/Alpha Clamp 1					
Vector Clamp	imp 1					
Scalar/Alpha Mask	•		Defines which out of 32 bits words (four of them) in the result is written back In the register file			
Vector Write M	Write Mask		Defines which out of 32 bits words (four of them) in the result is written back			
Scalar/Alpha pointer	a result		Sp	ecifies the address into the	register files for result of scalar/alpha operation	
Vector result p	ointer					
Scalar Export	flag		ТΒ	D		
Vector Export	flag		ΤВ	D		

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
СМхх	D = A if C xx 0.0, else B	Xx can be: gt,gte,eq (lt,lte,ne can be generated by swaping 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 - floor(A)	
MINV	D = min(A,B)	
MAXV	D = max(A,B)	
Area	D = 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.
Ехр	D = Pow(a)	Possible opcode, otherwise texture lookup
RSQR	D = 1/sqrt(a)	Possible opcode, otherwise texture lookup
Log2	D = Log(A, base = 2)	Possible opcode, otherwise table lookup
Log	D = Log(A, base = B)	Possible opcode, otherwise table lookup
Pow	D = A to the B'th power	
SCLP	D = ABC	Concatinate 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.

Exhibit 2042.doc 16774 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:44 PM

	0	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
		23 January, 2001	4 September, 2015	R400 Shader Processor Model	10 of 12
	The	export opcodes must b			
EXPP	Exp	ort position: position = A,	clipcodes = B	Export position and clip codes o vertex shader	f vertex, end
EXP Export vertex component, {D} = A			Export vertex component to verte possible destinations	ex cache, 16	
EXPXY	Plac	ceholder, {0} = 1,0,0,0		Placeholder for the rasterizer to put and Z into pixel shader.	t pixel position
EXPC	Export color			Export color of pixel, end pixel shade	er
EXPZ	PZ 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	D 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 betweeen volume texture levels			

Exhibit 2042.doc 16774 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:44 PM

	ORIO	GINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	23 .	January, 2001	4 September, 2015	GEN-CXXXXX-REVA	11 of
		0- Nearest 1- Linear			
MIP_enable	1	0- no mipmapin	-		
		1- mip-mapping			
Filter_mode	2		four 8bit values		
			two 16bit values		
			one 32 bit value		
		3- input is not fi			
Scale	4			alue, used for vertex fetching	
Offset	4		o the x component, after the	e scale, used for vertex fetching.	
Combine	2	0: R = R			
mode		1: R = R + Tt0			
		2: Use Tt1 values	s as blend factors		
		3:			
Output mode	2		sult of fetch (NOP)		
			1: store result in GPR		
		2: store result in			
		3: store result in	Tt1		
Mip mode	2	0: normal			
		2: lower mip filter			
		3: upper mip filte			
Tex3D mode	2	0: normal (non vo	olume)		
		2: lower z fetch			
		3: upper z fetch			
Sample_bias_	4		offset for bi-linear sample.		
x				negative number will fetch the 2x2 sam	ples that
				cubic and arbitrary sized filters	
Sample_bias_	4		offset for bi-linear sample.		
У				negative number will fetch the 2x2 sam	ples 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			

Exhibit 2042.doc 16774 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***09/04/15 04:44 PM

	ORIG	INATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE			
	23 January, 2001		4 September, 2015	R400 Shader Processor Model	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 betweeen volume texture levels						
_		2- Nearest						
MIP_enable	1	2- no mipmaping						
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 multipled 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 typ	3	?:1D array						
e	-	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.						

Exhibit 2042.doc 16774 Bytes*** © ATI Confidential. Reference Copyright Notice on Cover Page © ***_{09/04/15 04:44 PM}