	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 2015~~14~~ ~~October, 200211~~	GEN-CXXXXX-REVA	1 of 51

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SQ

Version 2.07

Overview: 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 sub-blocks, and provides internal state diagrams.

AUTOMATICALLY UPDATED FIELDS:
Document Location: C:\perforce\r400\doc_lib\design\blocks\sq\R400_Sequencer.doc
Current Intranet Search Title: R400 Sequencer Specification

APPROVALS

Name/Dept	Signature/Date

Remarks:

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

Table Of Contents

Revision Changes:

Rev 0.1 (Laurent Lefebvre) Date: May 7, 2001	First draft.
Rev 0.2 (Laurent Lefebvre) Date : July 9, 2001	Changed the interfaces to reflect the changes in the SP. Added some details in the arbitration section.
Rev 0.3 (Laurent Lefebvre) Date : August 6, 2001	Reviewed the Sequencer spec after the meeting on August 3, 2001.
Rev 0.4 (Laurent Lefebvre) Date : August 24, 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.
Rev 0.5 (Laurent Lefebvre) Date : September 7, 2001	Added timing diagrams (Vic)
Rev 0.6 (Laurent Lefebvre) Date : September 24, 2001	Changed the spec to reflect the new R400 architecture. Added interfaces.
Rev 0.7 (Laurent Lefebvre) Date : October 5, 2001	Added constant store management, instruction store management, control flow management and data dependant predication.
Rev 0.8 (Laurent Lefebvre) Date : October 8, 2001	Changed the control flow method to be more flexible. Also updated the external interfaces.
Rev 0.9 (Laurent Lefebvre) Date : October 17, 2001	Incorporated changes made in the 10/18/01 control flow meeting. Added a NOP instruction, removed the conditional_execute_or_jump. Added debug registers.
Rev 1.0 (Laurent Lefebvre) Date : October 19, 2001	Refined interfaces to RB. Added state registers.
Rev 1.1 (Laurent Lefebvre) Date : October 26, 2001	Added SEQ→SP0 interfaces. Changed delta precision. Changed VGT→SP0 interface. Debug Methods added.
Rev 1.2 (Laurent Lefebvre) Date : November 16, 2001	Interfaces greatly refined. Cleaned up the spec.
Rev 1.3 (Laurent Lefebvre) Date : November 26, 2001	Added the different interpolation modes.
Rev 1.4 (Laurent Lefebvre) Date : December 6, 2001	Added the auto incrementing counters. Changed the VGT→SQ interface. Added content on constant management. Updated GPRs.
Rev 1.5 (Laurent Lefebvre) Date : December 11, 2001	Removed from the spec all interfaces that weren't directly tied to the SQ. Added explanations on constant management. Added PA→SQ synchronization fields and explanation.
Rev 1.6 (Laurent Lefebvre) Date : January 7, 2002	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.
Rev 1.7 (Laurent Lefebvre) Date : February 4, 2002	Added Real Time parameter control in the SX interface. Updated the control flow section.
Rev 1.8 (Laurent Lefebvre) Date : March 4, 2002	New interfaces to the SX block. Added the end of clause modifier, removed the end of clause instructions.
Rev 1.9 (Laurent Lefebvre) Date : March 18, 2002	Rearangement of the CF instruction bits in order to ensure byte alignement.
Rev 1.10 (Laurent Lefebvre) Date : March 25, 2002	Updated the interfaces and added a section on exporting rules.
Rev 1.11 (Laurent Lefebvre) Date : April 19, 2002	Added CP state report interface. Last version of the spec with the old control flow scheme
Rev 2.0 (Laurent Lefebvre) Date : April 19, 2002	New control flow scheme

Rev 2.01 (Laurent Lefebvre) Date : May 2, 2002	Changed slightly the control flow instructions to allow force jumps and calls.
Rev 2.02 (Laurent Lefebvre) Date : May 13, 2002	Updated the Opcodes. Added type field to the constant/pred interface. Added Last field to the SQ→SP instruction load interface.
Rev 2.03 (Laurent Lefebvre) Date : July 15, 2002	SP interface updated to include predication optimizations. Added the predicate no stall instructions,
Rev 2.04 (Laurent Lefebvre) Date :August 2, 2002	Documented the new parameter generation scheme for XY coordinates points and lines STs.
Rev 2.05 (Laurent Lefebvre) Date : September 10, 2002	Some interface changes and an architectural change to the auto-counter scheme.
Rev 2.06 (Laurent Lefebvre) Date : October 11, 2002	Widened the event interface to 5 bits. Some other little typos corrected.
Rev 2.07 (Laurent Lefebvre) Date : October 14, 2002	Loops, jumps and calls are now using a 13 bit address which allows to jump and call and loop around any control flow addresses (does not requires to be even anymore).

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.

PROTECTIVE ORDER MATERIAL

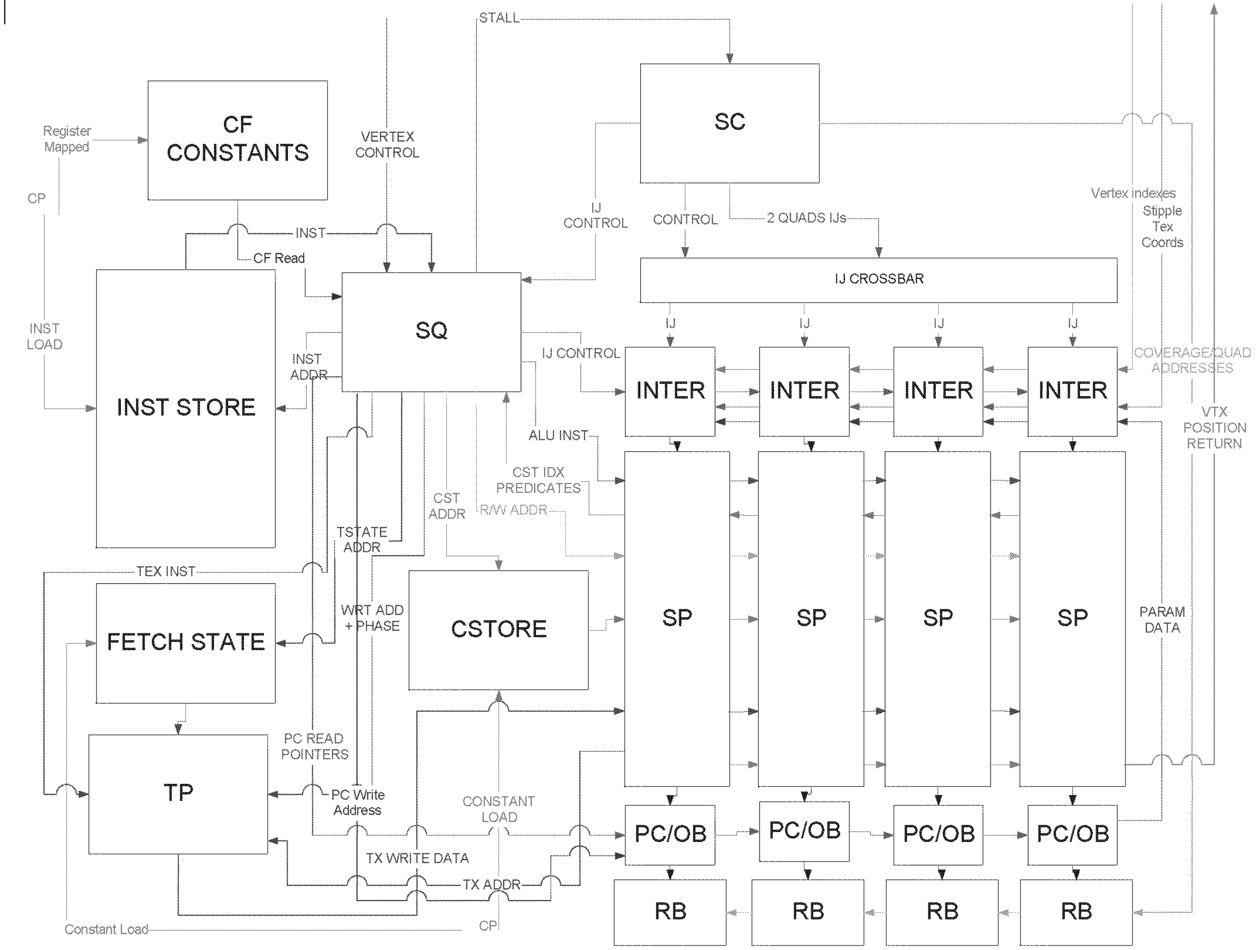


Figure 1: General Sequencer overview

1.1 Top Level Block Diagram

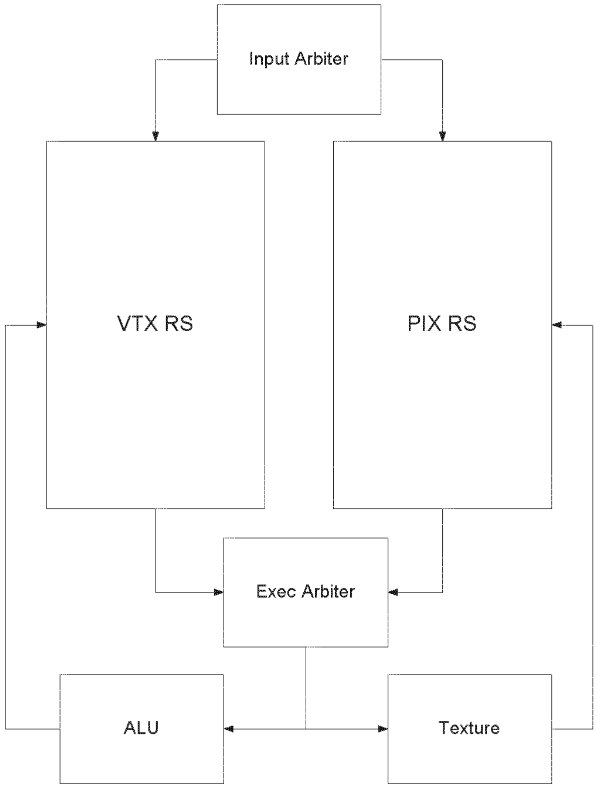


Figure 2: Reservation stations and arbiters

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).

1.2 Data Flow graph (SP)

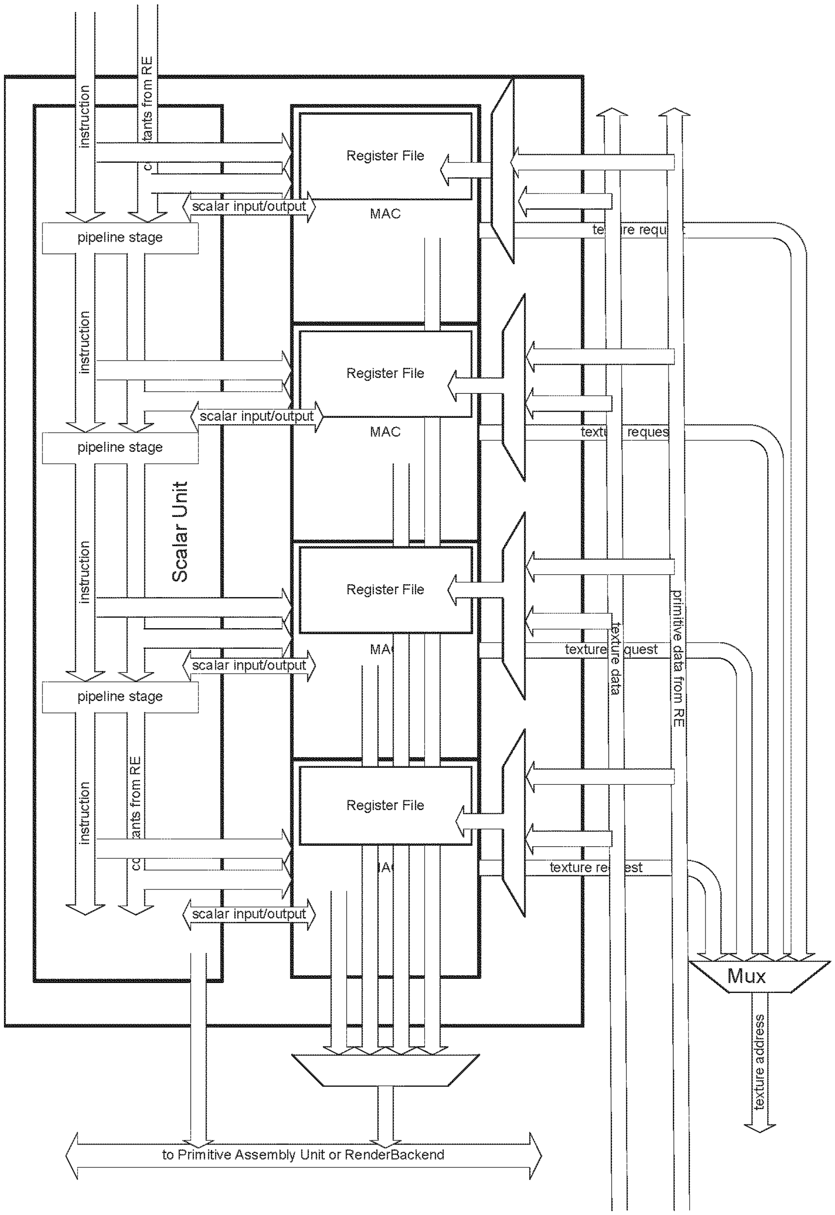


Figure 3: The shader Pipe

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

1.3 Control Graph

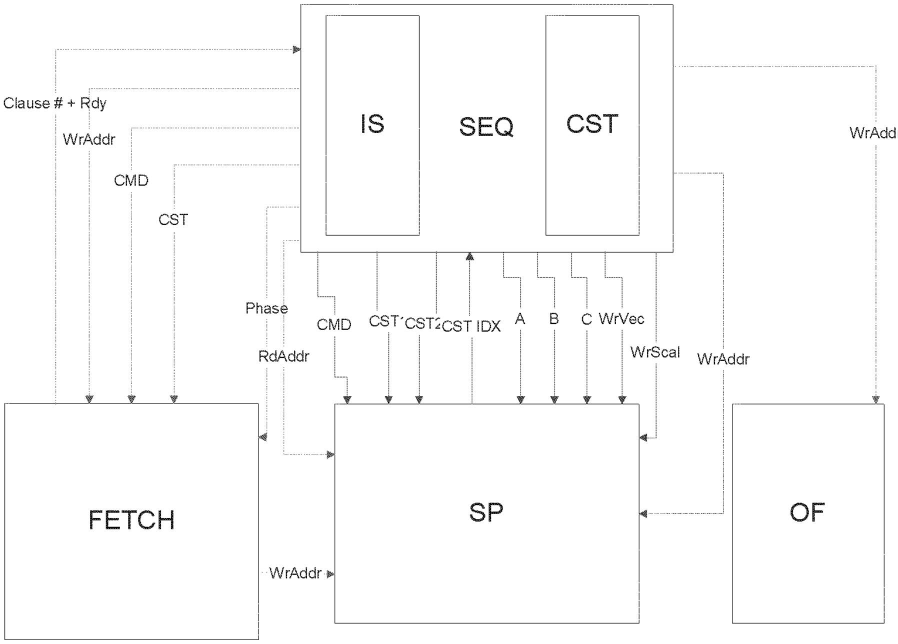


Figure 4: Sequencer Control interfaces

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.

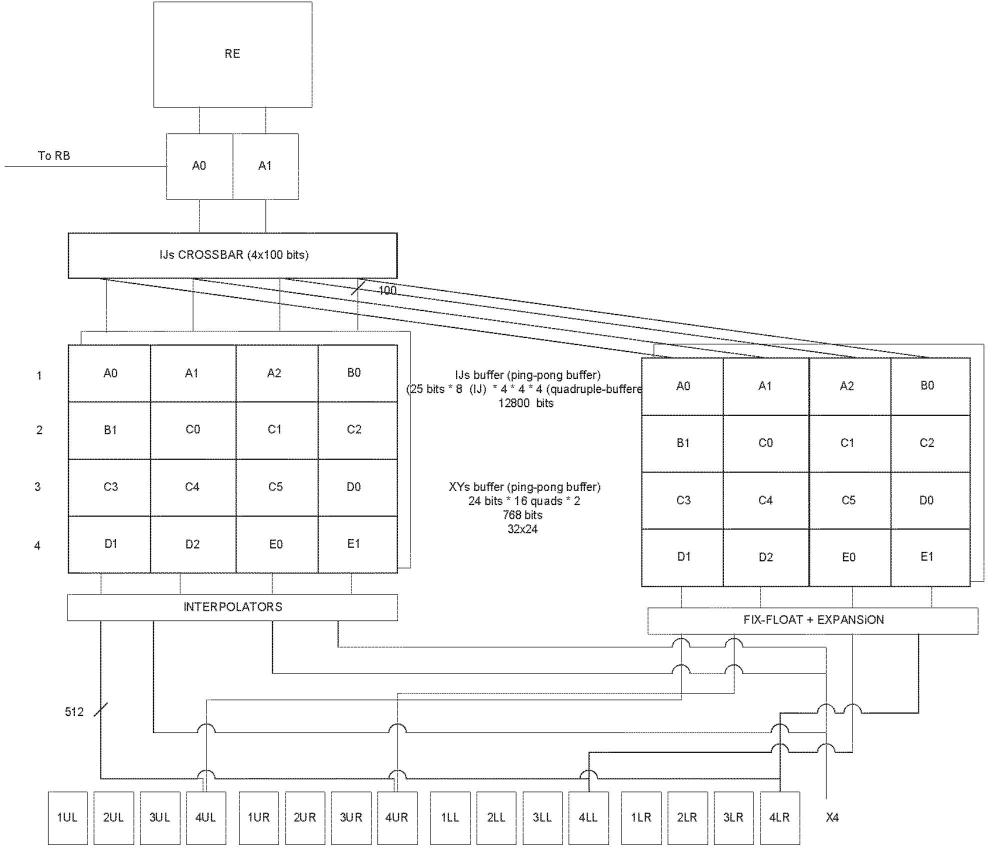


Figure 5: Interpolation buffers

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19	T20	T21	T22	T23
WRITES																								
SP 0	A0	A0	XY A0	B1	B1	XY B1	C3	C3	XY C3				D1	D1	XY D1									
SP 1	A1	A1	XY A1				C0	C0	XY C0	C4	C4	XY C4	D2	D2	XY D2									
SP 2	A2	A2	XY A2				C1	C1	XY C1	C5	C5	XY C5				E0	E0	XY E0						
SP 3				B0	B0	XY B0	C2	C2	XY C2				D0	D0	XY D0	E1	E1	XY E1						
READS																								
SP 0	XY 0-3	XY 16-19	XY 32-35	XY 48-51	A0	B1	C3	D1					A0	B1	C3	D1					V 0-3	V 16-19	V 32-35	V 48-51
SP 1	XY 4-7	XY 20-23	XY 36-39	XY 52-55	A1		C4	D2		C0			A1		C4	D2		C0			V 4-7	V 20-23	V 36-39	V 52-55
SP 2	XY 8-11	XY 24-27	XY 40-43	XY 56-59	A2		C5			C1		E0	A2		C5			C1		E0	V 8-11	V 24-27	V 40-43	V 56-59
SP 3	XY 12-15	XY 28-31	XY 44-47	XY 60-63					B0	C2	D0	E1					B0	C2	D0	E1	V 12-15	V 28-31	V 44-47	V 60-63
		XY						P1								P2						VTX		

Figure 6: Interpolation timing diagram

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.

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.

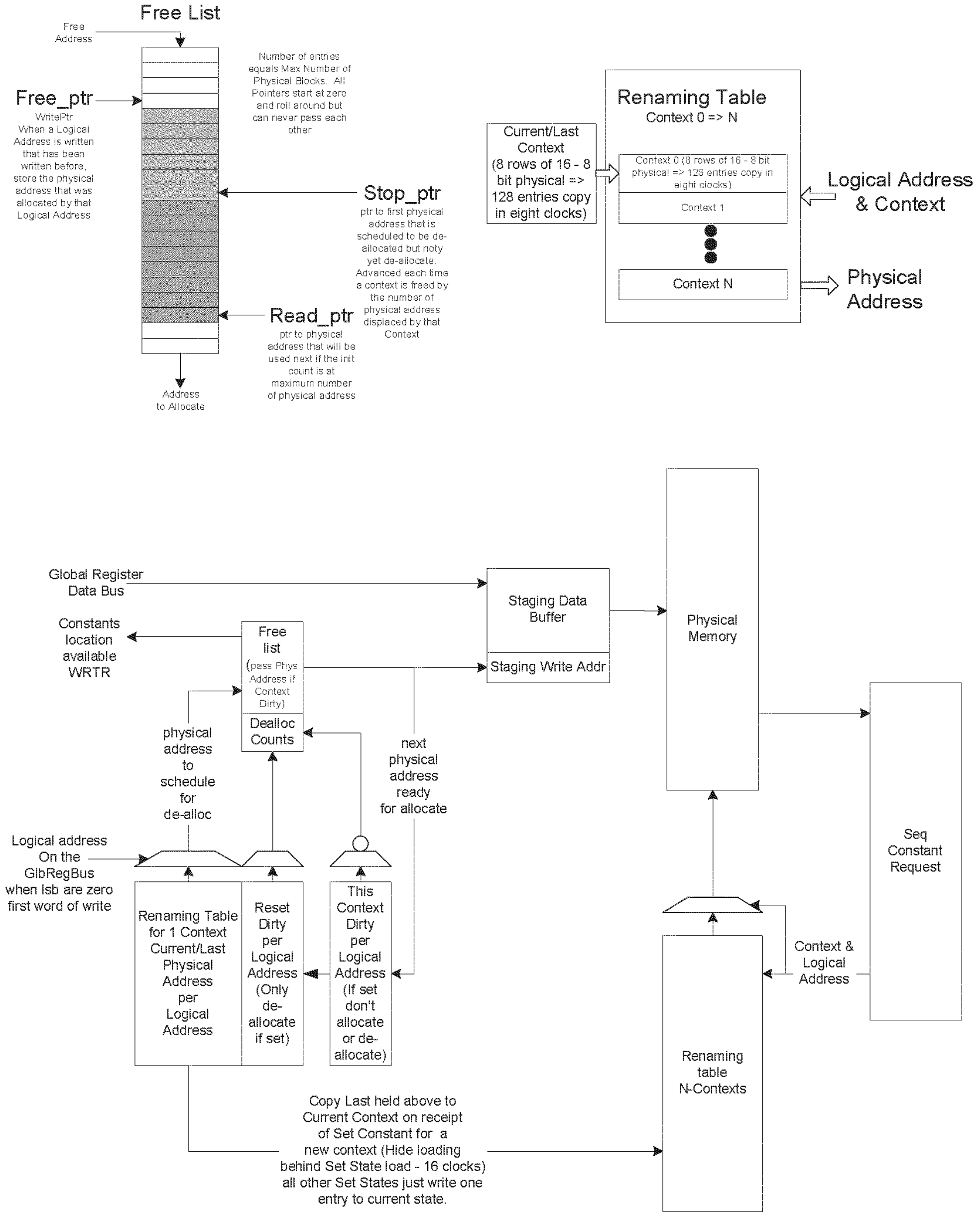
The constant memory update will be incremental, the driver only need to update the constants that actually changed between the two state changes.

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 Figure 8: De-allocation mechanism~~Figure 8: De-allocation mechanism~~~~Figure 8: De-allocation mechanism~~). Note that in the case a state is cleared a value 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.

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).



Free List
Free Address
Free_ptr
WritePtr
When a Logical Address is written that has been written before, store the physical address that was allocated by that Logical Address
Number of entries equals Max Number of Physical Blocks. All Pointers start at zero and roll around but can never pass each other
Stop_ptr
ptr to first physical address that is scheduled to be de-allocated but noty yet de-allocate. Advanced each time a context is freed by the number of physical address displaced by that Context
Read_ptr
ptr to physical address that will be used next if the init count is at maximum number of physical address
Address to Allocate
Renaming Table
Context 0 => N
Current/Last Context (8 rows of 16 - 8 bit physical => 128 entries copy in eight clocks)
Context 0 (8 rows of 16 - 8 bit physical => 128 entries copy in eight clocks)
Context 1
Context N
Logical Address & Context
Physical Address
Global Register Data Bus
Staging Data Buffer
Staging Write Addr
Physical Memory
Constants location available WRTR
Free list (pass Phys Address if Context Dirty)
Dealloc Counts
physical address to schedule for de-alloc
next physical address ready for allocate
Logical address On the GlbRegBus when lsb are zero first word of write
Renaming Table for 1 Context Current/Last Physical Address per Logical Address
Reset Dirty per Logical Address (Only de-allocate if set)
This Context Dirty per Logical Address (If set don't allocate or de-allocate)
Seq Constant Request
Context & Logical Address
Renaming table N-Contexts
Copy Last held above to Current Context on receipt of Set Constant for a new context (Hide loading behind Set State load - 16 clocks) all other Set States just write one entry to current state.

Figure 7: Constant management

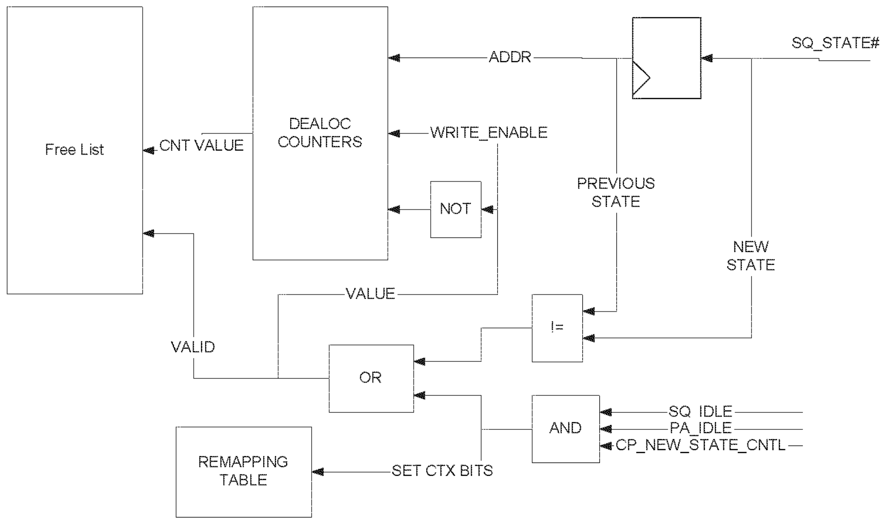


Figure 8: De-allocation mechanism for R400LE

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 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.

5.3.4 *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.

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 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 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). 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.

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.

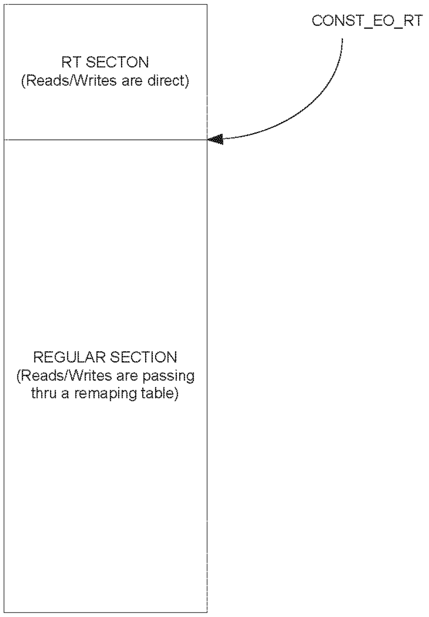


Figure 9: The Constant store

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[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 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 …16	15…12	11 … 0
0001	Addressing	RESERVED	Instructions type + serialize (9 instructions)	Count	Exec Address

Execute_End					
47 … 44	43	40 … 34	33 …16	15…12	11 … 0
0010	Addressing	RESERVED	Instructions type + serialize (9 instructions)	Count	Exec Address

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	33…16	15 …12	11 … 0
0011	Addressing	Condition	Boolean address	Instructions type + serialize (9 instructions)	Count	Exec Address

Conditional_Execute_End						
47 … 44	43	42	41 … 34	33…16	15 …12	11 … 0
0100	Addressing	Condition	Boolean address	Instructions type + serialize (9 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	33…16	15…12	11 … 0
0101	Addressing	Condition	RESERVED	Predicate vector	Instructions type + serialize (9 instructions)	Count	Exec Address

Conditional_Execute_Predicates_End							
47 … 44	43	42	41 … 36	35 … 34	33…16	15…12	11 … 0
0110	Addressing	Condition	RESERVED	Predicate vector	Instructions type + serialize (9 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.

Conditional_Execute_Predicates_No_Stall							
47 … 44	43	42	41 … 36	35 … 34	33…16	15…12	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	33…16	15…12	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	15…1~~2~~3	12~~1~~ … 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	~~11~~ 12 … 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.

Conditionnal_Call						
47 … 44	43	42	41 … 34	33 … ~~13~~14	~~12~~13	~~11~~ 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 … ~~13~~14	~~12~~13	~~11~~ 12 … 0
1011	Addressing	Condition	Boolean address	FW only	RESERVED	Force Jump	Jump 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	42…41	40 … 3	2…0
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 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)
~~12.~~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.

Formatted: Bullets and Numbering

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)
- 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:

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 non-masked 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.

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

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 (position, color, z, ect) 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:

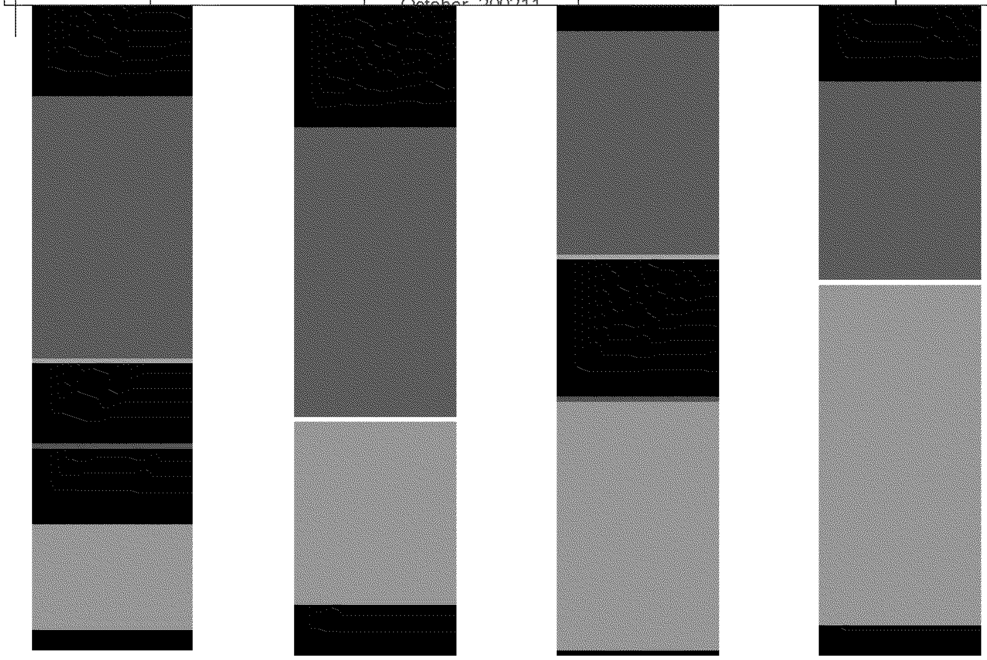
MASK_SETE
MASK_SETNE
MASK_SETGT
MASK_SETGTE

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.



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 arbiters, 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.

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.

$$P0 = A + I(0) * (B - A) + J(0) * (C - A)$$
$$P1 = A + I(1) * (B - A) + J(1) * (C - A)$$
$$P2 = A + I(2) * (B - A) + J(2) * (C - A)$$
$$P3 = A + I(3) * (B - A) + J(3) * (C - A)$$

P0	P1
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 11~~Figure 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 10~~Figure 10Figure 10~~.

Basis for 8-deep Latch Memory (from 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 μ^2 per converter		
Method 1	Block	Quantity	Area
	F2F	3	14136
	8x96 Latch	16	744384
			758520 μ^2

Figure 10:Area Estimate for VGT to Shader Interface

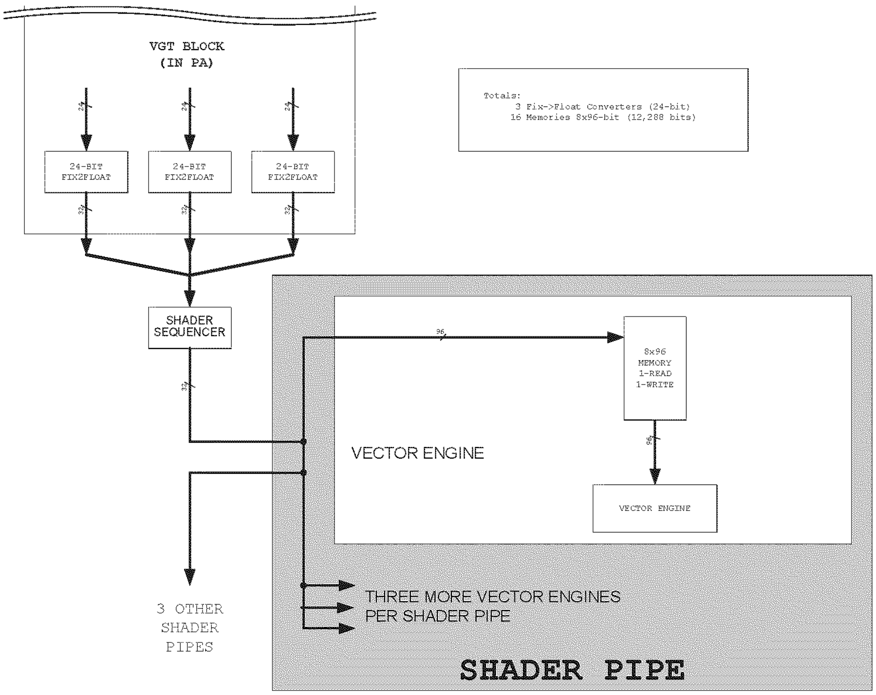


Figure 11:VGT to Shader Interface

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 00100000000 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 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 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 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

63 - sprite size export that goes with position export
(X= point size, Y= edge flag is bit 0, Z= VtxKill is bitwise OR of bits 30:0. Any bit other than sign means VtxKill.)

18.2 Pixel Shading

0	- Color for buffer 0 (primary)
1	- Color for buffer 1
2	- Color for buffer 2
3	- Color for buffer 3
4:15	- Empty
16	- Buffer 0 Color/Fog (primary)
17	- Buffer 1 Color/Fog
18	- Buffer 2 Color/Fog
19	- Buffer 3 Color/Fog
20:31	- Empty
32	- Export Address
33:37	- 5 exports for multipass pixel shaders.
38:47	- 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_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

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 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:

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:

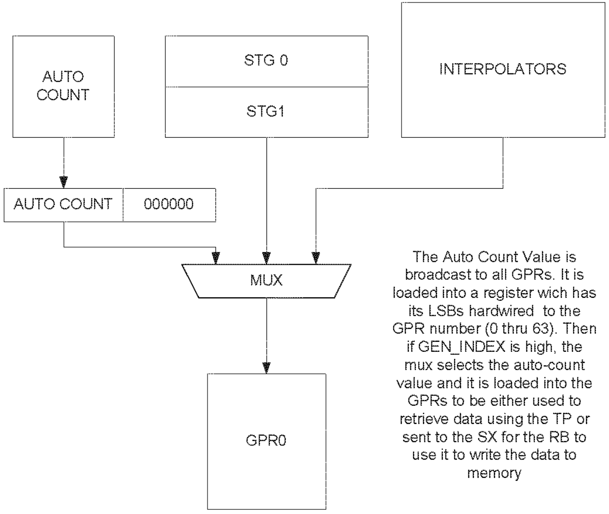
Counter (21 bits)	Hardwired (4 bits)

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.



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. 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.

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→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
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

		interpolation 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
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, 2nd clk could be all zeroes

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

Name	BitField	Bits	Description
1st 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

			=> TBD
SC_SQ_state_id	[8:6]	3	State/constant pointer (6*3+3)
SC_SQ_pc_dealloc	[11:9]	3	Deallocation token for the Parameter Cache
SC_SQ_new_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_mask	[16:13]	4	Quad Write mask left to right SP0 => SP3
SC_SQ_end_of_prim	17	1	End Of the primitive
SC_SQ_pix_mask	[33:18]	16	Valid bits for all pixels SP0=>SP3 (UL,UR,LL,LR)
SC_SQ_provok_vtx	[35:34]	2	Provoking vertex for flat shading
SC_SQ_lod_correct_0	[44:36]	9	LOD correction for quad 0 (SP0) (9 bits per quad)
SC_SQ_lod_correct_1	[53:45]	9	LOD correction for quad 1 (SP1) (9 bits per quad)
2nd Clock Transfer			
SC_SQ_lod_correct_2	[8:0]	9	LOD correction for quad 2 (SP2) (9 bits per quad)
SC_SQ_lod_correct_3	[17:9]	9	LOD correction for quad 3 (SP3) (9 bits per quad)
SC_SQ_pc_ptr0	[28:18]	11	Parameter Cache pointer for vertex 0
SC_SQ_pc_ptr1	[39:29]	11	Parameter Cache pointer for vertex 1
SC_SQ_pc_ptr2	[50:40]	11	Parameter Cache pointer for vertex 2
SC_SQ_prim_type	[53:51]	3	Stippled line and Real time command need to load tex cords from alternate buffer 000: Sprite (point) 001: Line 010: Tri_rect 100: Realtime Sprite (point) 101: Realtime Line 110: Realtime Tri_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.)

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

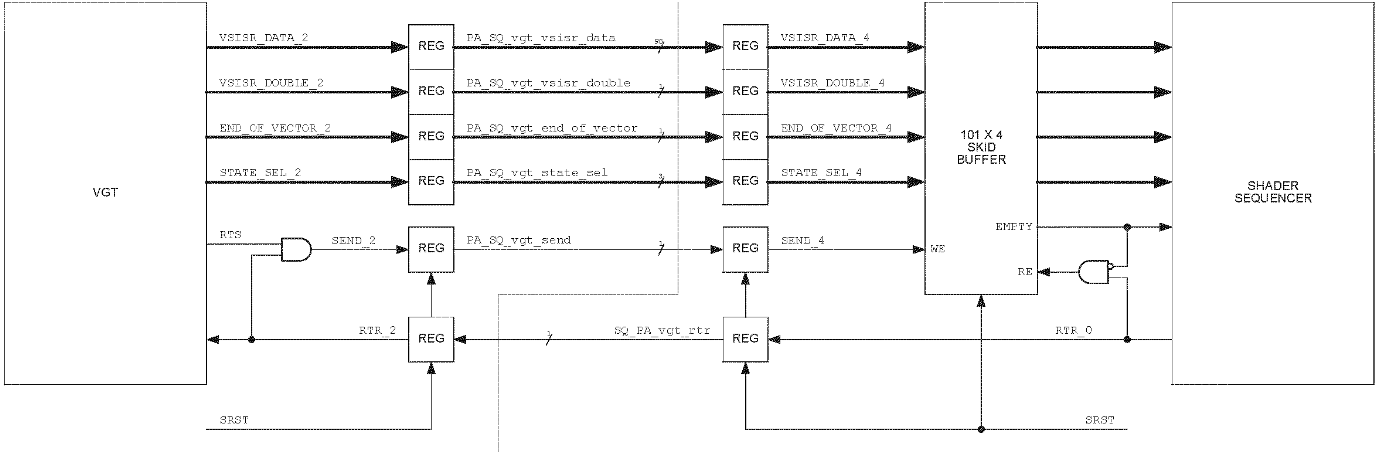
23.2.5 *VGT to SQ : Vertex interface*

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. **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.** 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


VGT
VSISR_DATA_2
VSISR_DOUBLE_2
END_OF_VECTOR_2
STATE_SEL_2
RTS
SEND_2
RTR_2
SRST
REG
PA_SQ_vgt_vsisr_data
96
PA_SQ_vgt_vsisr_double
1
PA_SQ_vgt_end_of_vector
1
PA_SQ_vgt_state_sel
3
PA_SQ_vgt_send
1
SQ_PA_vgt_rtr
VSISR_DATA_4
VSISR_DOUBLE_4
END_OF_VECTOR_4
STATE_SEL_4
SEND_4
101 X 4
SKID
BUFFER
WE
EMPTY
RE
RTR_0
SRST
SHADER
SEQUENCER

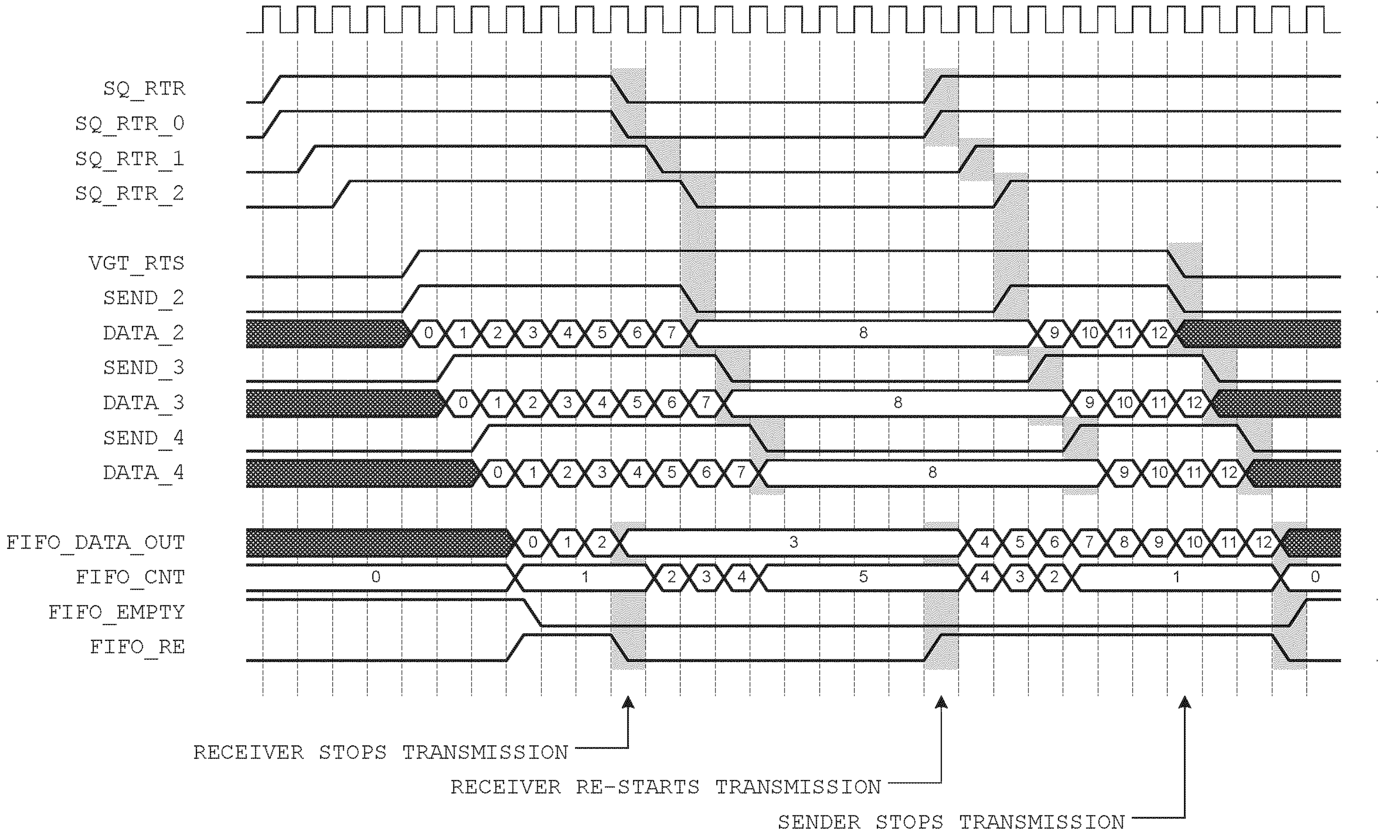


Figure 1. Detailed Logical Diagram for PA_SQ_vgt Interface.

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)
  - 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

			pixels in a clause) ... 64: 128K-bits available (16 128-bit entries for each of 64 pixels) 65-127: RESERVED

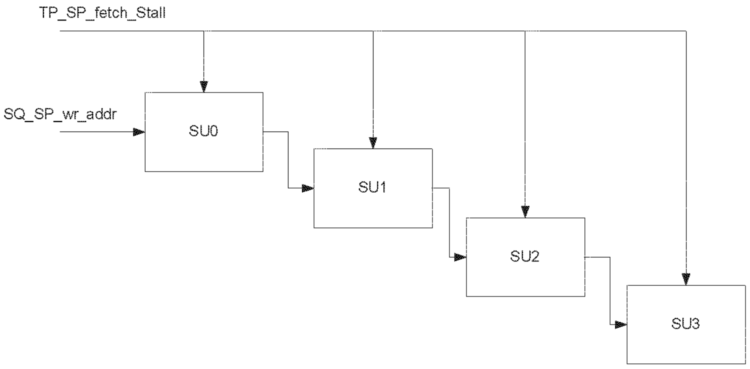
23.2.8 *SQ to TP: Control bus*

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→ SQ	1	Data ready
TPx_SQ_rs_line_num	TPx→ 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.



Name	Direction	Bits	Description
TP_SQ_fetch_stall	TP→ SQ	1	Do not send more texture request if asserted

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

23.2.12 *SQ to SPx: Instructions*

Name	Direction	Bits	Description
SQ_SPx_instr_start	SQ→SPx	1	Instruction start
SQ_SP_instr	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 all 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 all 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 all 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_pred_override	SQ→SP0	4	0: Use per channel RGBA field (enables the per channel logic, if not set only pay attention to the 11 seting). 1: Use GPR
SQ_SP1_pred_override	SQ→SP1	4	0: Use per channel RGBA field (enables the per channel logic, if not set only pay attention to the 11 seting). 1: Use GPR
SQ_SP2_pred_override	SQ→SP2	4	0: Use per channel RGBA field (enables the per channel logic, if not set only pay attention to the 11 seting). 1: Use GPR
SQ_SP3_pred_override	SQ→SP3	4	0: Use per channel RGBA field (enables the per channel logic, if not set only pay attention to the 11 seting). 1: Use GPR
SQ_SPx_exp_id	SQ→SPx	1	GPR ID

SQ_SPx_exporting	SQ→SPx	1	0: Not Exporting 1: Exporting
SQ_SPx_stall	SQ→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

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

23.2.18 *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)

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	TYPE	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:

```
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.

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:

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 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 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 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?

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 2015~~16~~ October, 200214	GEN-CXXXXX-REVA	1 of 51

Author: Laurent Lefebvre

Issue To: **Copy No:**

R400 Sequencer Specification

SQ

Version 2.08~~7~~

Overview: 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 sub-blocks, and provides internal state diagrams.

AUTOMATICALLY UPDATED FIELDS:
Document Location: C:\perforce\r400\doc_lib\design\blocks\sq\R400_Sequencer.doc
Current Intranet Search Title: R400 Sequencer Specification

APPROVALS	
Name/Dept	Signature/Date

Remarks:

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

Table Of Contents

Revision Changes:

Rev 0.1 (Laurent Lefebvre) Date: May 7, 2001	First draft.
Rev 0.2 (Laurent Lefebvre) Date : July 9, 2001	Changed the interfaces to reflect the changes in the SP. Added some details in the arbitration section.
Rev 0.3 (Laurent Lefebvre) Date : August 6, 2001	Reviewed the Sequencer spec after the meeting on August 3, 2001.
Rev 0.4 (Laurent Lefebvre) Date : August 24, 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.
Rev 0.5 (Laurent Lefebvre) Date : September 7, 2001	Added timing diagrams (Vic)
Rev 0.6 (Laurent Lefebvre) Date : September 24, 2001	Changed the spec to reflect the new R400 architecture. Added interfaces.
Rev 0.7 (Laurent Lefebvre) Date : October 5, 2001	Added constant store management, instruction store management, control flow management and data dependant predication.
Rev 0.8 (Laurent Lefebvre) Date : October 8, 2001	Changed the control flow method to be more flexible. Also updated the external interfaces.
Rev 0.9 (Laurent Lefebvre) Date : October 17, 2001	Incorporated changes made in the 10/18/01 control flow meeting. Added a NOP instruction, removed the conditional_execute_or_jump. Added debug registers.
Rev 1.0 (Laurent Lefebvre) Date : October 19, 2001	Refined interfaces to RB. Added state registers.
Rev 1.1 (Laurent Lefebvre) Date : October 26, 2001	Added SEQ→SP0 interfaces. Changed delta precision. Changed VGT→SP0 interface. Debug Methods added.
Rev 1.2 (Laurent Lefebvre) Date : November 16, 2001	Interfaces greatly refined. Cleaned up the spec.
Rev 1.3 (Laurent Lefebvre) Date : November 26, 2001	Added the different interpolation modes.
Rev 1.4 (Laurent Lefebvre) Date : December 6, 2001	Added the auto incrementing counters. Changed the VGT→SQ interface. Added content on constant management. Updated GPRs.
Rev 1.5 (Laurent Lefebvre) Date : December 11, 2001	Removed from the spec all interfaces that weren't directly tied to the SQ. Added explanations on constant management. Added PA→SQ synchronization fields and explanation.
Rev 1.6 (Laurent Lefebvre) Date : January 7, 2002	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.
Rev 1.7 (Laurent Lefebvre) Date : February 4, 2002	Added Real Time parameter control in the SX interface. Updated the control flow section.
Rev 1.8 (Laurent Lefebvre) Date : March 4, 2002	New interfaces to the SX block. Added the end of clause modifier, removed the end of clause instructions.
Rev 1.9 (Laurent Lefebvre) Date : March 18, 2002	Rearangement of the CF instruction bits in order to ensure byte alignement.
Rev 1.10 (Laurent Lefebvre) Date : March 25, 2002	Updated the interfaces and added a section on exporting rules.
Rev 1.11 (Laurent Lefebvre) Date : April 19, 2002	Added CP state report interface. Last version of the spec with the old control flow scheme
Rev 2.0 (Laurent Lefebvre) Date : April 19, 2002	New control flow scheme

Rev 2.01 (Laurent Lefebvre) Date : May 2, 2002	Changed slightly the control flow instructions to allow force jumps and calls.
Rev 2.02 (Laurent Lefebvre) Date : May 13, 2002	Updated the Opcodes. Added type field to the constant/pred interface. Added Last field to the SQ→SP instruction load interface.
Rev 2.03 (Laurent Lefebvre) Date : July 15, 2002	SP interface updated to include predication optimizations. Added the predicate no stall instructions,
Rev 2.04 (Laurent Lefebvre) Date :August 2, 2002	Documented the new parameter generation scheme for XY coordinates points and lines STs.
Rev 2.05 (Laurent Lefebvre) Date : September 10, 2002	Some interface changes and an architectural change to the auto-counter scheme.
Rev 2.06 (Laurent Lefebvre) Date : October 11, 2002	Widened the event interface to 5 bits. Some other little typos corrected.
Rev 2.07 (Laurent Lefebvre) Date : October 14, 2002	Loops, jumps and calls are now using a 13 bit address which allows to jump and call and loop around any control flow addresses (does not requires to be even anymore).
Rev 2.08 (Laurent Lefebvre) Date : October 16, 2002	Clarification updates after discussion with Clay.

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.

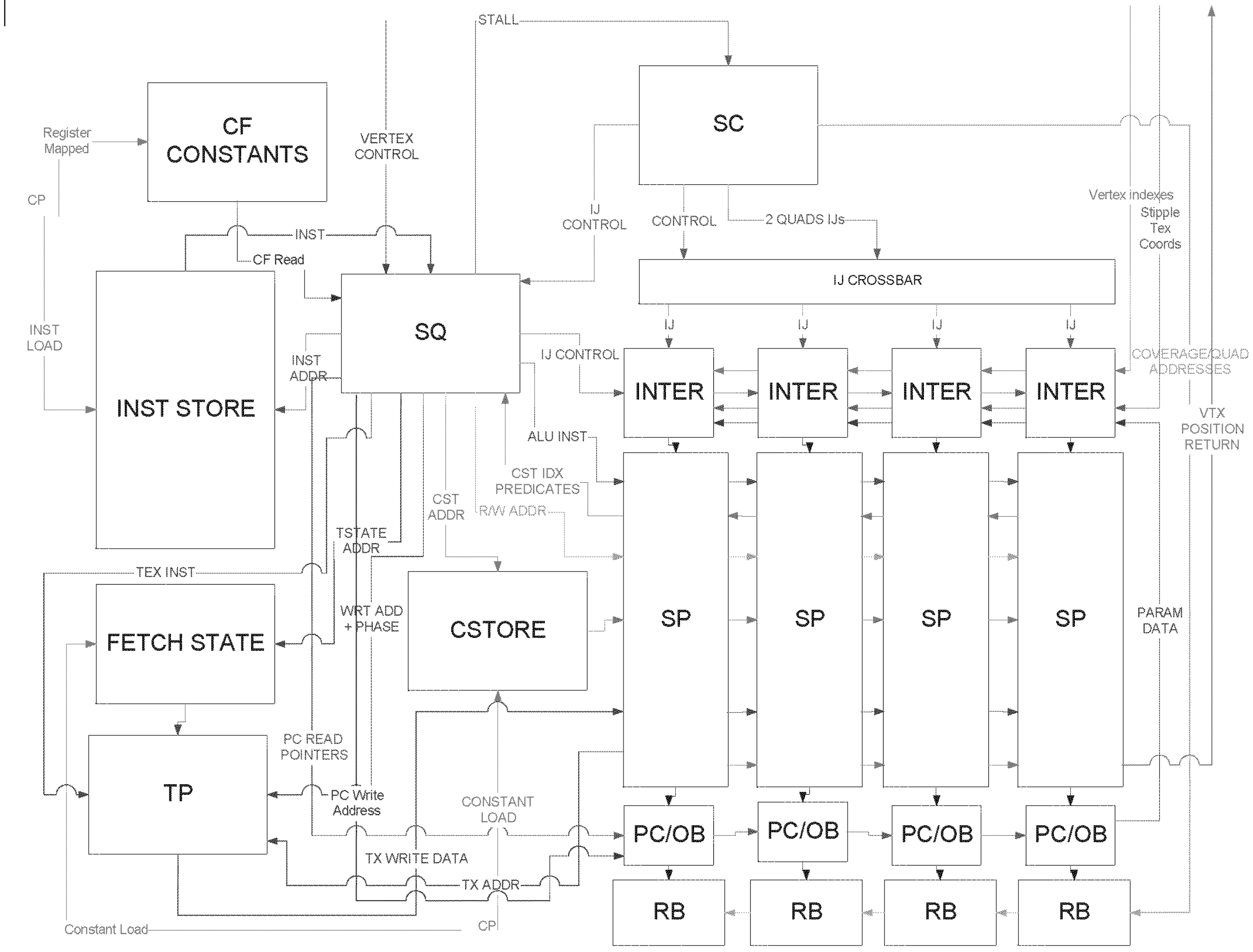


Figure 1: General Sequencer overview

1.1 Top Level Block Diagram

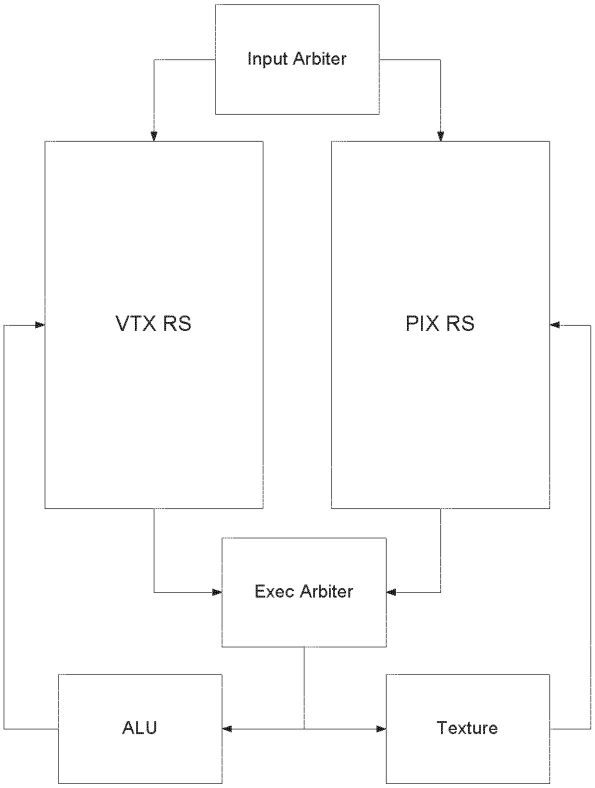


Figure 2: Reservation stations and arbiters

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).

1.2 Data Flow graph (SP)

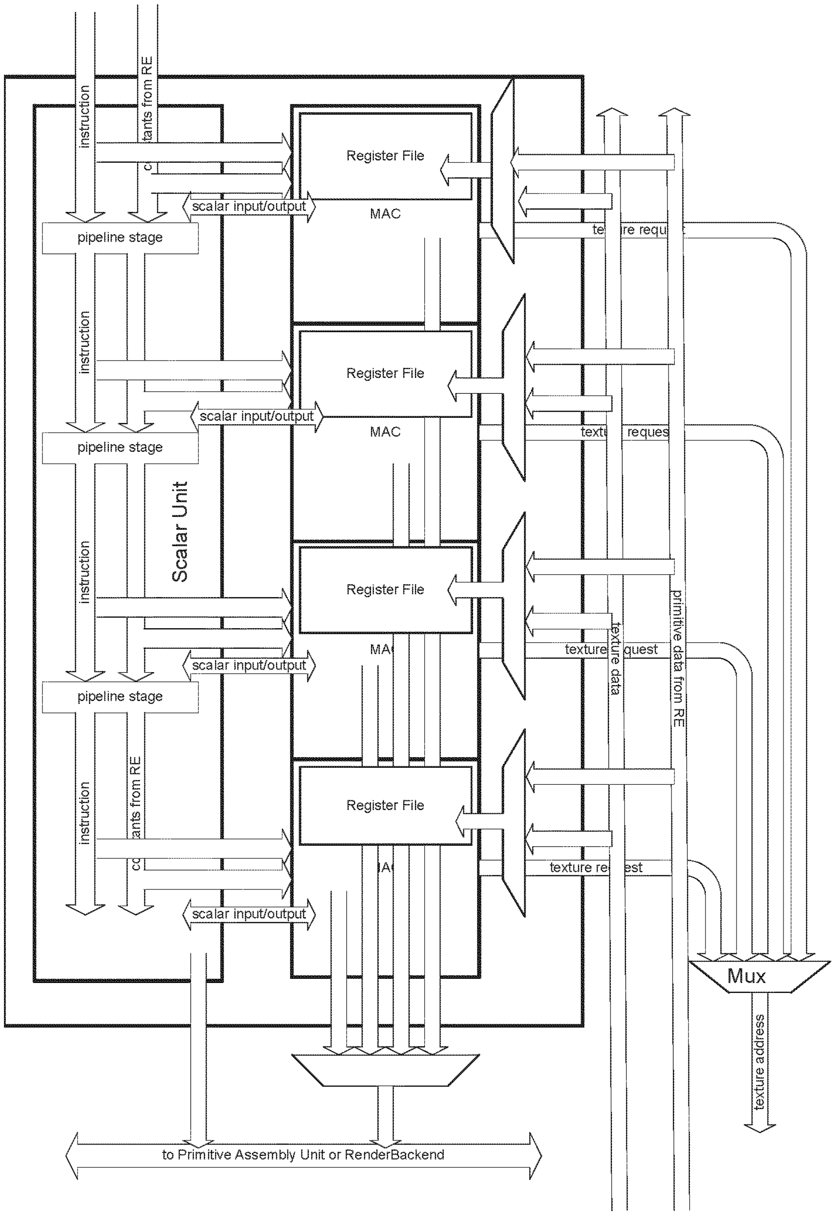


Figure 3: The shader Pipe

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

1.3 Control Graph

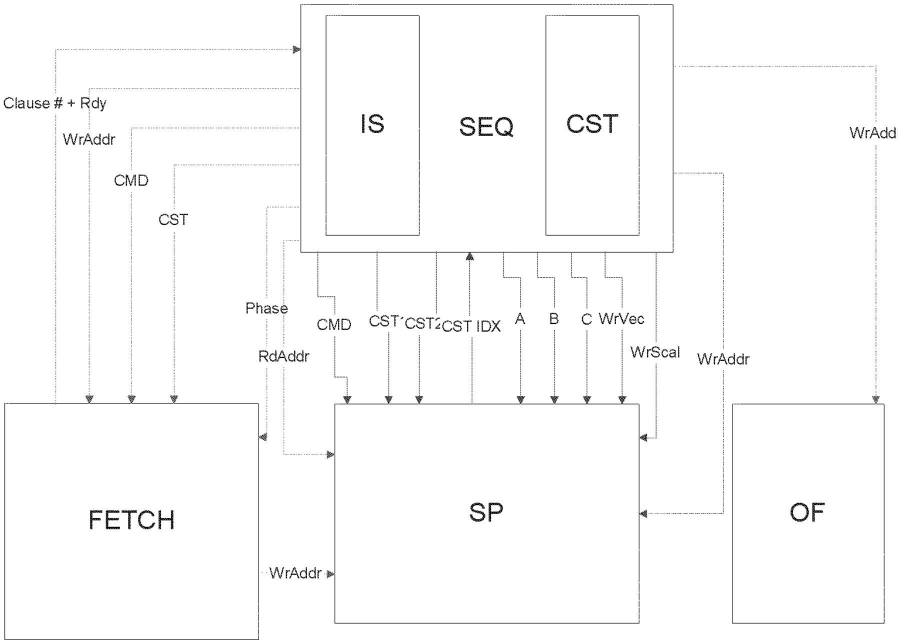


Figure 4: Sequencer Control interfaces

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.

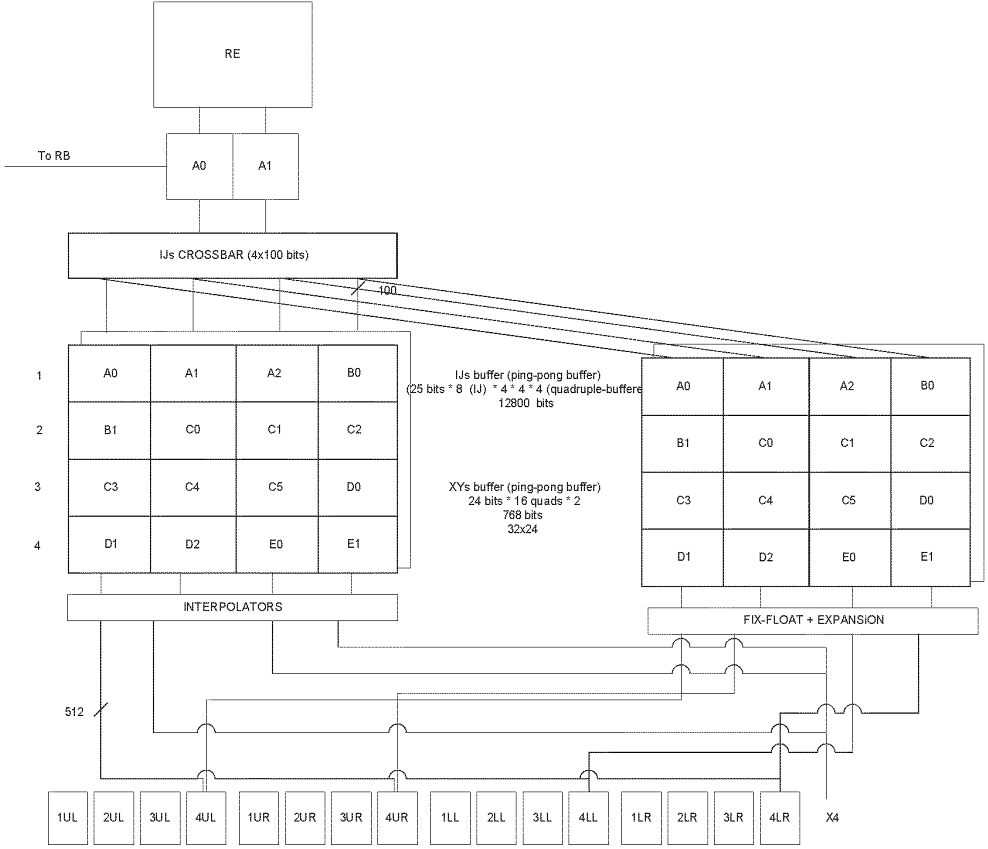


Figure 5: Interpolation buffers

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19	T20	T21	T22	T23
											WRITES													
SP 0	A0	A0	XY A0	B1	B1	XY B1	C3	C3	XY C3				D1	D1	XY D1									
SP 1	A1	A1	XY A1				C0	C0	XY C0	C4	C4	XY C4	D2	D2	XY D2									
SP 2	A2	A2	XY A2				C1	C1	XY C1	C5	C5	XY C5				E0	E0	XY E0						
SP 3				B0	B0	XY B0	C2	C2	XY C2				D0	D0	XY D0	E1	E1	XY E1						
											READS													
SP 0	XY 0-3	XY 16-19	XY 32-35	XY 48-51	A0	B1	C3	D1					A0	B1	C3	D1					V 0-3	V 16-19	V 32-35	V 48-51
SP 1	XY 4-7	XY 20-23	XY 36-39	XY 52-55	A1		C4	D2		C0			A1		C4	D2		C0			V 4-7	V 20-23	V 36-39	V 52-55
SP 2	XY 8-11	XY 24-27	XY 40-43	XY 56-59	A2		C5			C1		E0	A2		C5			C1		E0	V 8-11	V 24-27	V 40-43	V 56-59
SP 3	XY 12-15	XY 28-31	XY 44-47	XY 60-63					B0	C2	D0	E1					B0	C2	D0	E1	V 12-15	V 28-31	V 44-47	V 60-63
		XY						P1								P2						VTX		

Figure 6: Interpolation timing diagram

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.

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.

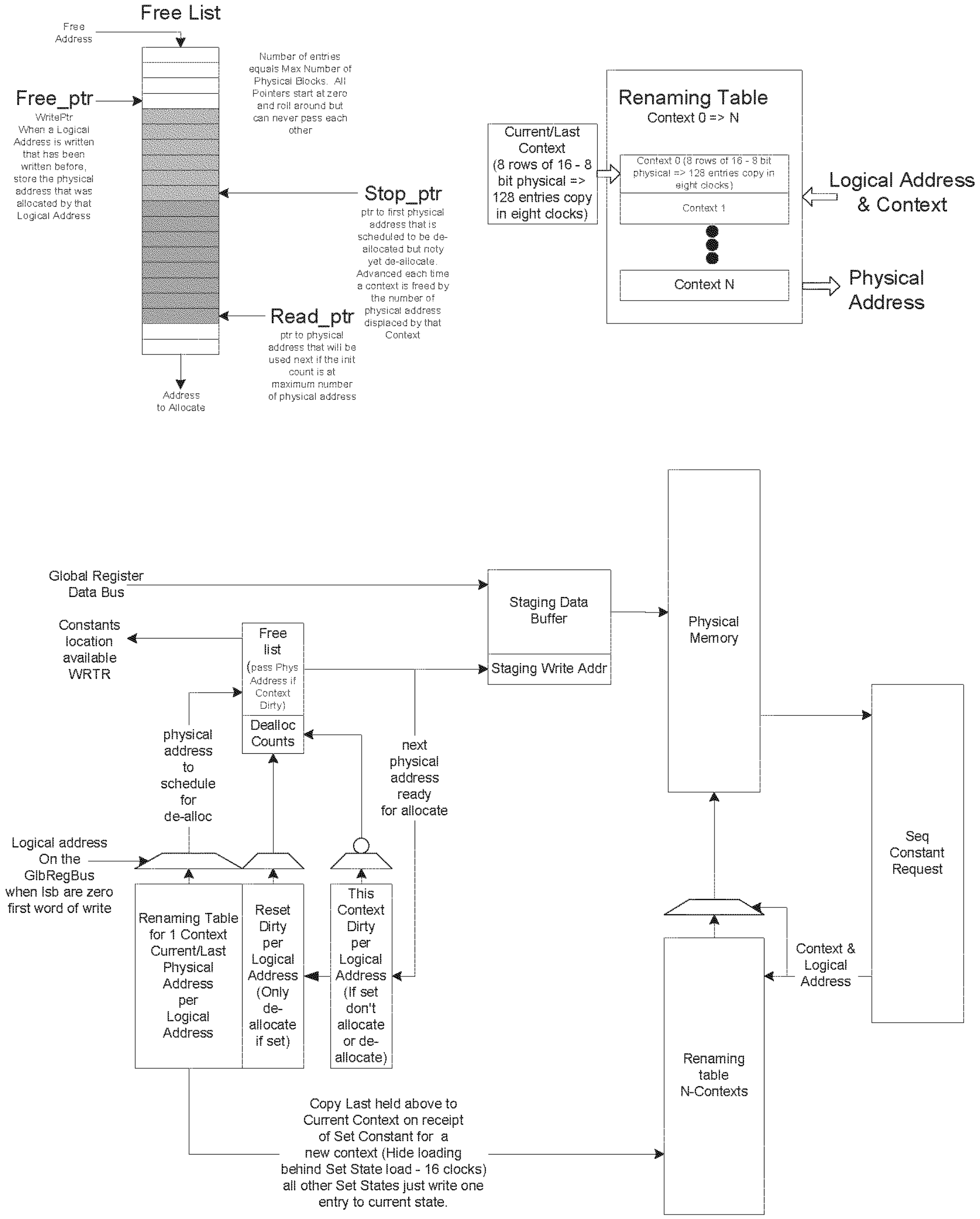
The constant memory update will be incremental, the driver only need to update the constants that actually changed between the two state changes.

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 Figure 8: De-allocation mechanism~~Figure 8: De-allocation mechanism~~Figure 8: De-allocation mechanism). Note that in the case a state is cleared a value 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.

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).



Free List
Free Address
Number of entries equals Max Number of Physical Blocks. All Pointers start at zero and roll around but can never pass each other
Free_ptr
WritePtr
When a Logical Address is written that has been written before, store the physical address that was allocated by that Logical Address
Stop_ptr
ptr to first physical address that is scheduled to be de-allocated but noty yet de-allocate. Advanced each time a context is freed by the number of physical address displaced by that Context
Read_ptr
ptr to physical address that will be used next if the init count is at maximum number of physical address
Address to Allocate
Renaming Table
Context 0 => N
Current/Last Context (8 rows of 16 - 8 bit physical => 128 entries copy in eight clocks)
Context 0 (8 rows of 16 - 8 bit physical => 128 entries copy in eight clocks)
Context 1
Context N
Logical Address & Context
Physical Address
Global Register Data Bus
Staging Data Buffer
Staging Write Addr
Physical Memory
Constants location available WRTR
Free list (pass Phys Address if Context Dirty)
Dealloc Counts
physical address to schedule for de-alloc
next physical address ready for allocate
Logical address On the GlbRegBus when lsb are zero first word of write
Renaming Table for 1 Context Current/Last Physical Address per Logical Address
Reset Dirty per Logical Address (Only de-allocate if set)
This Context Dirty per Logical Address (If set don't allocate or de-allocate)
Seq Constant Request
Context & Logical Address
Renaming table N-Contexts
Copy Last held above to Current Context on receipt of Set Constant for a new context (Hide loading behind Set State load - 16 clocks) all other Set States just write one entry to current state.

Figure 7: Constant management

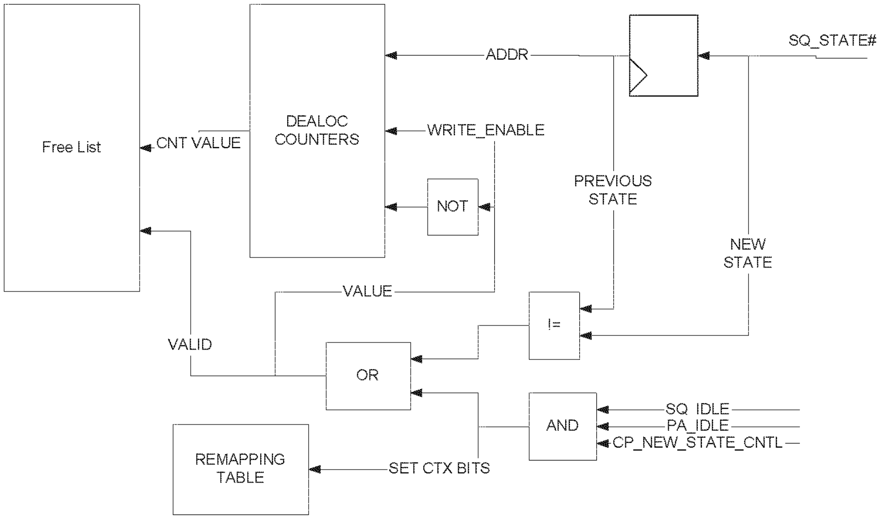


Figure 8: De-allocation mechanism for R400LE

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 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.

5.3.4 *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.

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 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 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).

~~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.

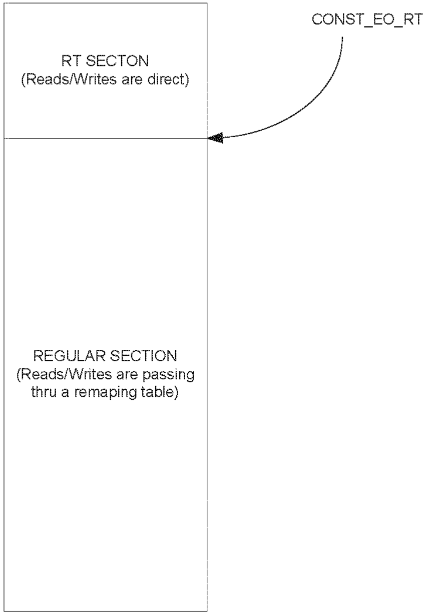

CONST_EO_RT
RT SECTON
(Reads/Writes are direct)
REGULAR SECTION
(Reads/Writes are passing thru a remaping table)

Figure 9: 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~~6~~: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 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 …16	15…12	11 … 0
0001	Addressing	RESERVED	Instructions type + serialize (9 instructions)	Count	Exec Address

Execute_End					
47 … 44	43	40 … 34	33 …16	15…12	11 … 0
0010	Addressing	RESERVED	Instructions type + serialize (9 instructions)	Count	Exec Address

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	33…16	15 …12	11 … 0
0011	Addressing	Condition	Boolean address	Instructions type + serialize (9 instructions)	Count	Exec Address

Conditional_Execute_End						
47 … 44	43	42	41 … 34	33…16	15 …12	11 … 0
0100	Addressing	Condition	Boolean address	Instructions type + serialize (9 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	33…16	15…12	11 … 0
0101	Addressing	Condition	RESERVED	Predicate vector	Instructions type + serialize (9 instructions)	Count	Exec Address

Conditional_Execute_Predicates_End							
47 … 44	43	42	41 … 36	35 … 34	33…16	15…12	11 … 0
0110	Addressing	Condition	RESERVED	Predicate vector	Instructions type + serialize (9 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.

Conditional_Execute_Predicates_No_Stall							
47 … 44	43	42	41 … 36	35 … 34	33…16	15…12	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	33…16	15…12	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	15…13	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 ~~… 24~~	41… 36	35…34	33… 22~~23… 21~~	21	20 … 16	15…13	12 … 0
1000	Addressing	~~RESERVED~~Cond	RESERVED	Predicate Vector	RESERVED ~~Predicate break~~	Pred 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~~Vector) to condition. If all bits ~~cleared then break the loop~~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

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 address	FW only	RESERVED	Force Jump	Jump 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	42…41	40 … 3	2…0
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 ~~1~~ 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)
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)

- 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:

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_SETE0~~_# – SETE0~~

PRED_SET~~E1_#~~NE ~~– SETE~~
PRED_SETGT ~~1~~
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 ~~4~~ 1 set~~s~~ of 64 bits predicate vectors (in fact 2~~8~~ 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 non-masked 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.~~detect wich channels to read from the GPRs and which ones 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

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 (but for position~~, color, z, ect~~) will be~~en~~ 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:

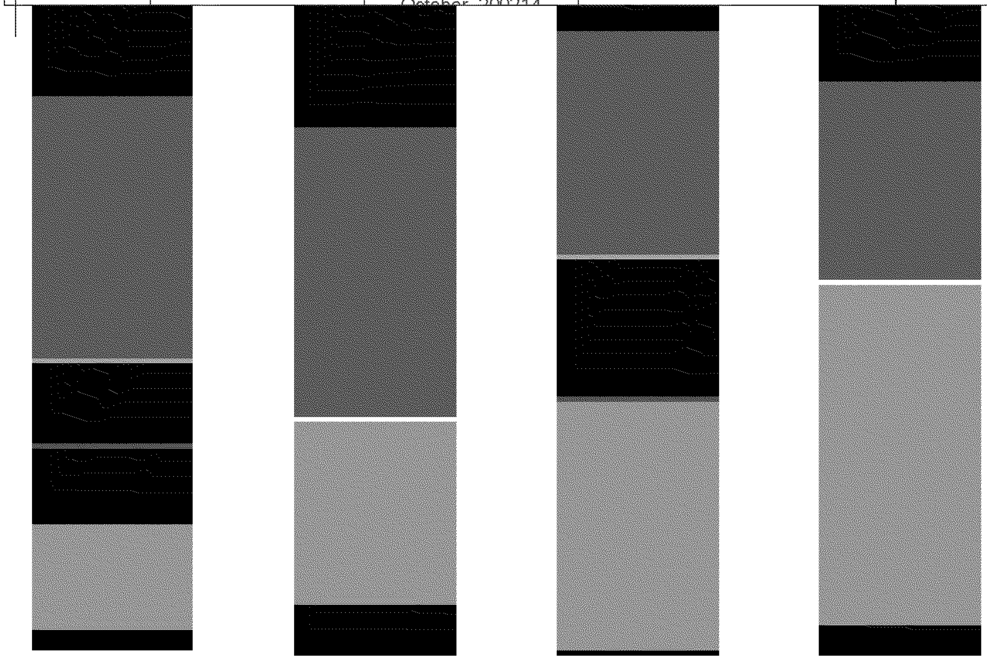
MASK_SETE
MASK_SETNE
MASK_SETGT
MASK_SETGTE

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.



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 arbiters, 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.

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.

$$P0 = A + I(0)*(B - A) + J(0)*(C - A)$$
$$P1 = A + I(1)*(B - A) + J(1)*(C - A)$$
$$P2 = A + I(2)*(B - A) + J(2)*(C - A)$$
$$P3 = A + I(3)*(B - A) + J(3)*(C - A)$$

P0	P1
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:

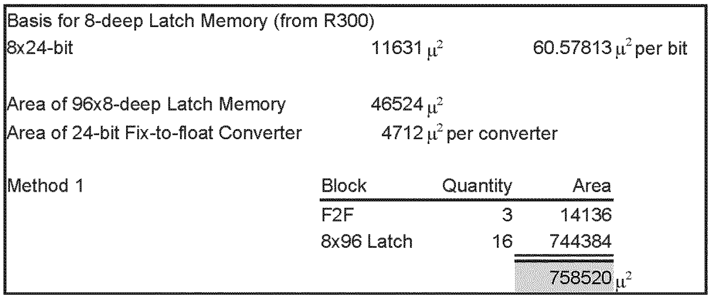
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 || 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 || 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 || 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

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 11~~Figure 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 10~~Figure 10Figure 10~~.



Basis for 8-deep Latch Memory (from 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 μ^2 per converter			
Method 1	Block	Quantity	Area	
	F2F	3	14136	
	8x96 Latch	16	744384	
			758520	μ^2

Figure 10:Area Estimate for VGT to Shader Interface

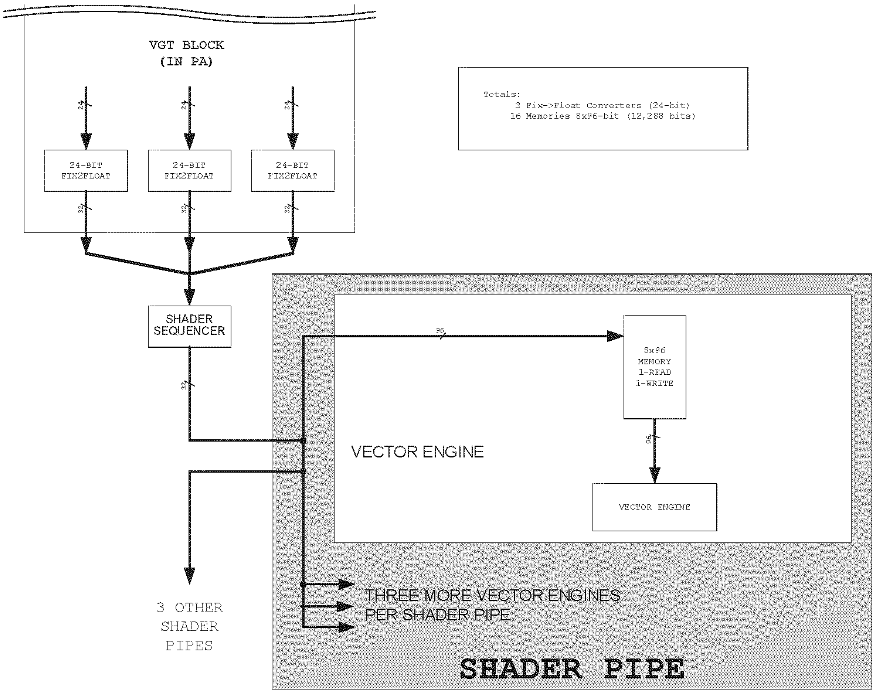


Figure 11:VGT to Shader Interface

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 00100000000 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 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 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 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 (or 8~~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.

Formatted

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)~~4)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.

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

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
63	- sprite size export that goes with position export

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

18.2 Pixel Shading

0	- Color for buffer 0 (primary)
1	- Color for buffer 1
2	- Color for buffer 2
3	- Color for buffer 3
4:15	- Empty
16	- Buffer 0 Color/Fog (primary)
17	- Buffer 1 Color/Fog
18	- Buffer 2 Color/Fog
19	- Buffer 3 Color/Fog
20:31	- Empty
32	- Export Address
33:37	- 5 exports for multipass pixel shaders.
38:47	- 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_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

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 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:

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:

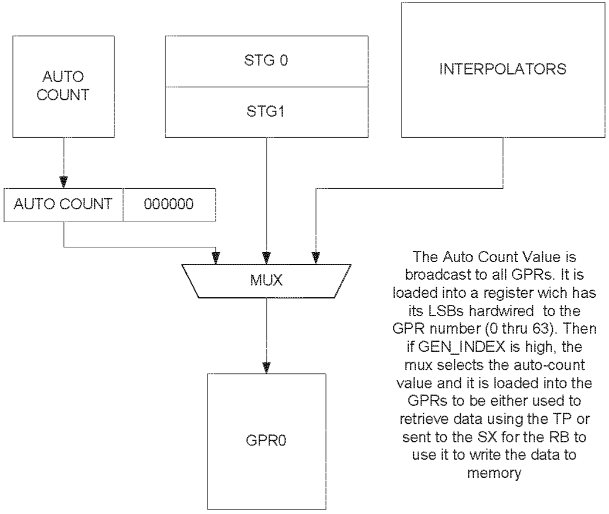
Counter (21 bits)	Hardwired (4 bits)

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.



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. 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.

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→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
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

		interpolation 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
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, 2nd clk could be all zeroes

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

Name	BitField	Bits	Description
1st 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

			=> TBD
SC_SQ_state_id	[8:6]	3	State/constant pointer (6*3+3)
SC_SQ_pc_dealloc	[11:9]	3	Deallocation token for the Parameter Cache
SC_SQ_new_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_mask	[16:13]	4	Quad Write mask left to right SP0 => SP3
SC_SQ_end_of_prim	17	1	End Of the primitive
SC_SQ_pix_mask	[33:18]	16	Valid bits for all pixels SP0=>SP3 (UL,UR,LL,LR)
SC_SQ_provok_vtx	[35:34]	2	Provoking vertex for flat shading
SC_SQ_lod_correct_0	[44:36]	9	LOD correction for quad 0 (SP0) (9 bits per quad)
SC_SQ_lod_correct_1	[53:45]	9	LOD correction for quad 1 (SP1) (9 bits per quad)
2nd Clock Transfer			
SC_SQ_lod_correct_2	[8:0]	9	LOD correction for quad 2 (SP2) (9 bits per quad)
SC_SQ_lod_correct_3	[17:9]	9	LOD correction for quad 3 (SP3) (9 bits per quad)
SC_SQ_pc_ptr0	[28:18]	11	Parameter Cache pointer for vertex 0
SC_SQ_pc_ptr1	[39:29]	11	Parameter Cache pointer for vertex 1
SC_SQ_pc_ptr2	[50:40]	11	Parameter Cache pointer for vertex 2
SC_SQ_prim_type	[53:51]	3	Stippled line and Real time command need to load tex cords from alternate buffer 000: Sprite (point) 001: Line 010: Tri_rect 100: Realtime Sprite (point) 101: Realtime Line 110: Realtime Tri_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.)

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

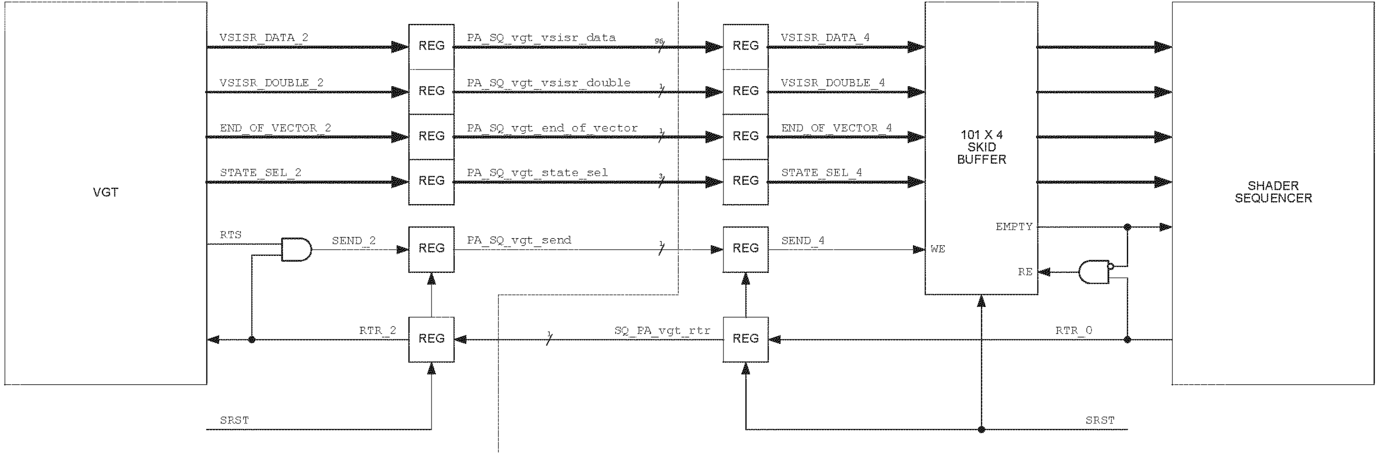
23.2.5 *VGT to SQ : Vertex interface*

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. **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.** 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


VGT
VSISR_DATA_2
VSISR_DOUBLE_2
END_OF_VECTOR_2
STATE_SEL_2
RTS
SEND_2
RTR_2
SRST
REG
PA_SQ_vgt_vsisr_data
PA_SQ_vgt_vsisr_double
PA_SQ_vgt_end_of_vector
PA_SQ_vgt_state_sel
PA_SQ_vgt_send
SQ_PA_vgt_rtr
96
1
1
3
1
VSISR_DATA_4
VSISR_DOUBLE_4
END_OF_VECTOR_4
STATE_SEL_4
SEND_4
101 X 4
SKID
BUFFER
WE
EMPTY
RE
RTR_0
SRST
SHADER
SEQUENCER

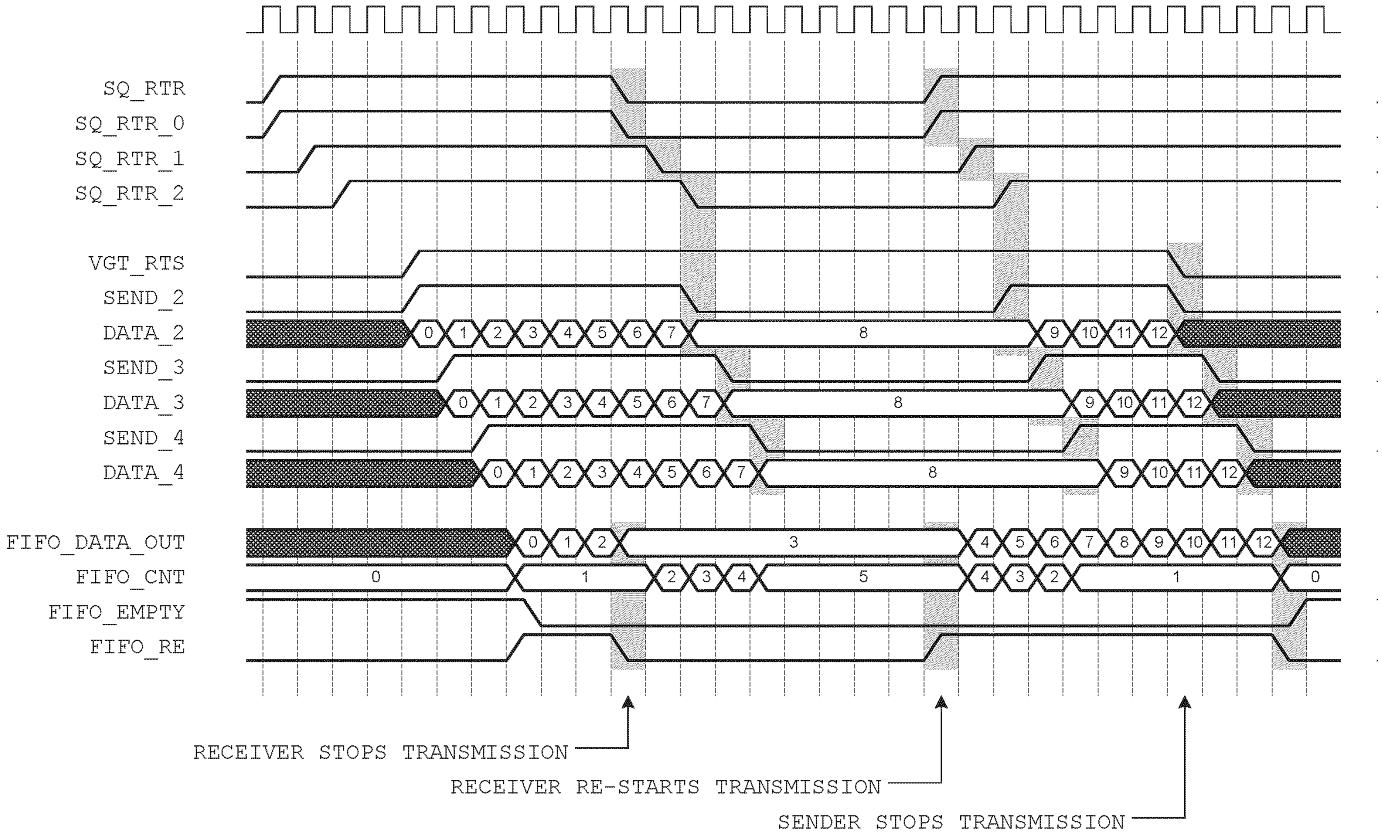


Figure 1. Detailed Logical Diagram for PA_SQ_vgt Interface.

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)
  - 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

			pixels in a clause) ... 64: 128K-bits available (16 128-bit entries for each of 64 pixels) 65-127: RESERVED

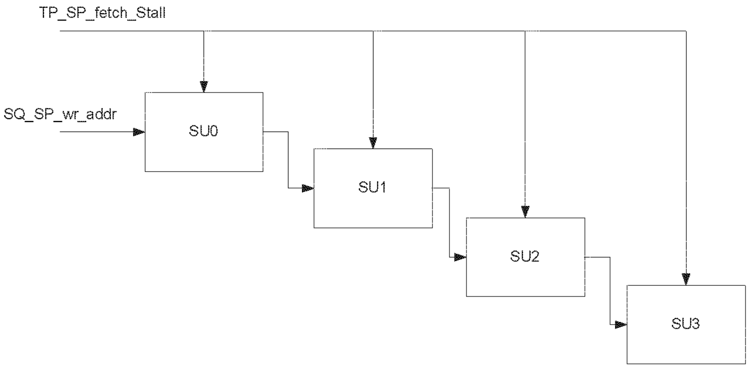
23.2.8 SQ to TP: Control bus

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→ SQ	1	Data ready
TPx_SQ_rs_line_num	TPx→ 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.



Name	Direction	Bits	Description
TP_SQ_fetch_stall	TP→ SQ	1	Do not send more texture request if asserted

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

23.2.12 *SQ to SPx: Instructions*

Name	Direction	Bits	Description
SQ_SPx_instr_start	SQ→SPx	1	Instruction start
SQ_SP_instr	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 all 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 all 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 all 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_pred_override	SQ→SP0	4	0: Use per channel RGBA field (enables the per channel logic, if not set only pay attention to the 11 seting). 1: Use GPR
SQ_SP1_pred_override	SQ→SP1	4	0: Use per channel RGBA field (enables the per channel logic, if not set only pay attention to the 11 seting). 1: Use GPR
SQ_SP2_pred_override	SQ→SP2	4	0: Use per channel RGBA field (enables the per channel logic, if not set only pay attention to the 11 seting). 1: Use GPR
SQ_SP3_pred_override	SQ→SP3	4	0: Use per channel RGBA field (enables the per channel logic, if not set only pay attention to the 11 seting). 1: Use GPR
SQ_SPx_exp_id	SQ→SPx	1	GPR ID

SQ_SPx_exporting	SQ→SPx	1	0: Not Exporting 1: Exporting
SQ_SPx_stall	SQ→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

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

23.2.18 *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)

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	TYPE	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:

```
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.

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:

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 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 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 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?

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 2015~~7~~ January, 200316	GEN-CXXXXX-REVA	1 of 54

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SQ

Version 2.09~~8~~

Overview: 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 sub-blocks, and provides internal state diagrams.

AUTOMATICALLY UPDATED FIELDS:
Document Location: C:\perforce\r400\doc_lib\design\blocks\sq\R400_Sequencer.doc
Current Intranet Search Title: R400 Sequencer Specification

APPROVALS	
Name/Dept	Signature/Date

Remarks:

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

ATI 2037
LG v. ATI
IPR2015-00325

Table Of Contents

Revision Changes:

Rev 0.1 (Laurent Lefebvre) Date: May 7, 2001	First draft.
Rev 0.2 (Laurent Lefebvre) Date : July 9, 2001	Changed the interfaces to reflect the changes in the SP. Added some details in the arbitration section.
Rev 0.3 (Laurent Lefebvre) Date : August 6, 2001	Reviewed the Sequencer spec after the meeting on August 3, 2001.
Rev 0.4 (Laurent Lefebvre) Date : August 24, 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.
Rev 0.5 (Laurent Lefebvre) Date : September 7, 2001	Added timing diagrams (Vic)
Rev 0.6 (Laurent Lefebvre) Date : September 24, 2001	Changed the spec to reflect the new R400 architecture. Added interfaces.
Rev 0.7 (Laurent Lefebvre) Date : October 5, 2001	Added constant store management, instruction store management, control flow management and data dependant predication.
Rev 0.8 (Laurent Lefebvre) Date : October 8, 2001	Changed the control flow method to be more flexible. Also updated the external interfaces.
Rev 0.9 (Laurent Lefebvre) Date : October 17, 2001	Incorporated changes made in the 10/18/01 control flow meeting. Added a NOP instruction, removed the conditional_execute_or_jump. Added debug registers.
Rev 1.0 (Laurent Lefebvre) Date : October 19, 2001	Refined interfaces to RB. Added state registers.
Rev 1.1 (Laurent Lefebvre) Date : October 26, 2001	Added SEQ→SP0 interfaces. Changed delta precision. Changed VGT→SP0 interface. Debug Methods added.
Rev 1.2 (Laurent Lefebvre) Date : November 16, 2001	Interfaces greatly refined. Cleaned up the spec.
Rev 1.3 (Laurent Lefebvre) Date : November 26, 2001	Added the different interpolation modes.
Rev 1.4 (Laurent Lefebvre) Date : December 6, 2001	Added the auto incrementing counters. Changed the VGT→SQ interface. Added content on constant management. Updated GPRs.
Rev 1.5 (Laurent Lefebvre) Date : December 11, 2001	Removed from the spec all interfaces that weren't directly tied to the SQ. Added explanations on constant management. Added PA→SQ synchronization fields and explanation.
Rev 1.6 (Laurent Lefebvre) Date : January 7, 2002	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.
Rev 1.7 (Laurent Lefebvre) Date : February 4, 2002	Added Real Time parameter control in the SX interface. Updated the control flow section.
Rev 1.8 (Laurent Lefebvre) Date : March 4, 2002	New interfaces to the SX block. Added the end of clause modifier, removed the end of clause instructions.
Rev 1.9 (Laurent Lefebvre) Date : March 18, 2002	Rearangement of the CF instruction bits in order to ensure byte alignement.
Rev 1.10 (Laurent Lefebvre) Date : March 25, 2002	Updated the interfaces and added a section on exporting rules.
Rev 1.11 (Laurent Lefebvre) Date : April 19, 2002	Added CP state report interface. Last version of the spec with the old control flow scheme
Rev 2.0 (Laurent Lefebvre) Date : April 19, 2002	New control flow scheme

Rev 2.01 (Laurent Lefebvre) Date : May 2, 2002	Changed slightly the control flow instructions to allow force jumps and calls.
Rev 2.02 (Laurent Lefebvre) Date : May 13, 2002	Updated the Opcodes. Added type field to the constant/pred interface. Added Last field to the SQ→SP instruction load interface.
Rev 2.03 (Laurent Lefebvre) Date : July 15, 2002	SP interface updated to include predication optimizations. Added the predicate no stall instructions,
Rev 2.04 (Laurent Lefebvre) Date :August 2, 2002	Documented the new parameter generation scheme for XY coordinates points and lines STs.
Rev 2.05 (Laurent Lefebvre) Date : September 10, 2002	Some interface changes and an architectural change to the auto-counter scheme.
Rev 2.06 (Laurent Lefebvre) Date : October 11, 2002	Widened the event interface to 5 bits. Some other little typos corrected.
Rev 2.07 (Laurent Lefebvre) Date : October 14, 2002	Loops, jumps and calls are now using a 13 bit address which allows to jump and call and loop around any control flow addresses (does not requires to be even anymore).
Rev 2.08 (Laurent Lefebvre) Date : October 16, 2002	Clarification updates after discussion with Clay.
Rev 2.09 (Laurent Lefebvre) Date : January 7, 2003	Corrected the SQ→SP staging register interface.

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.

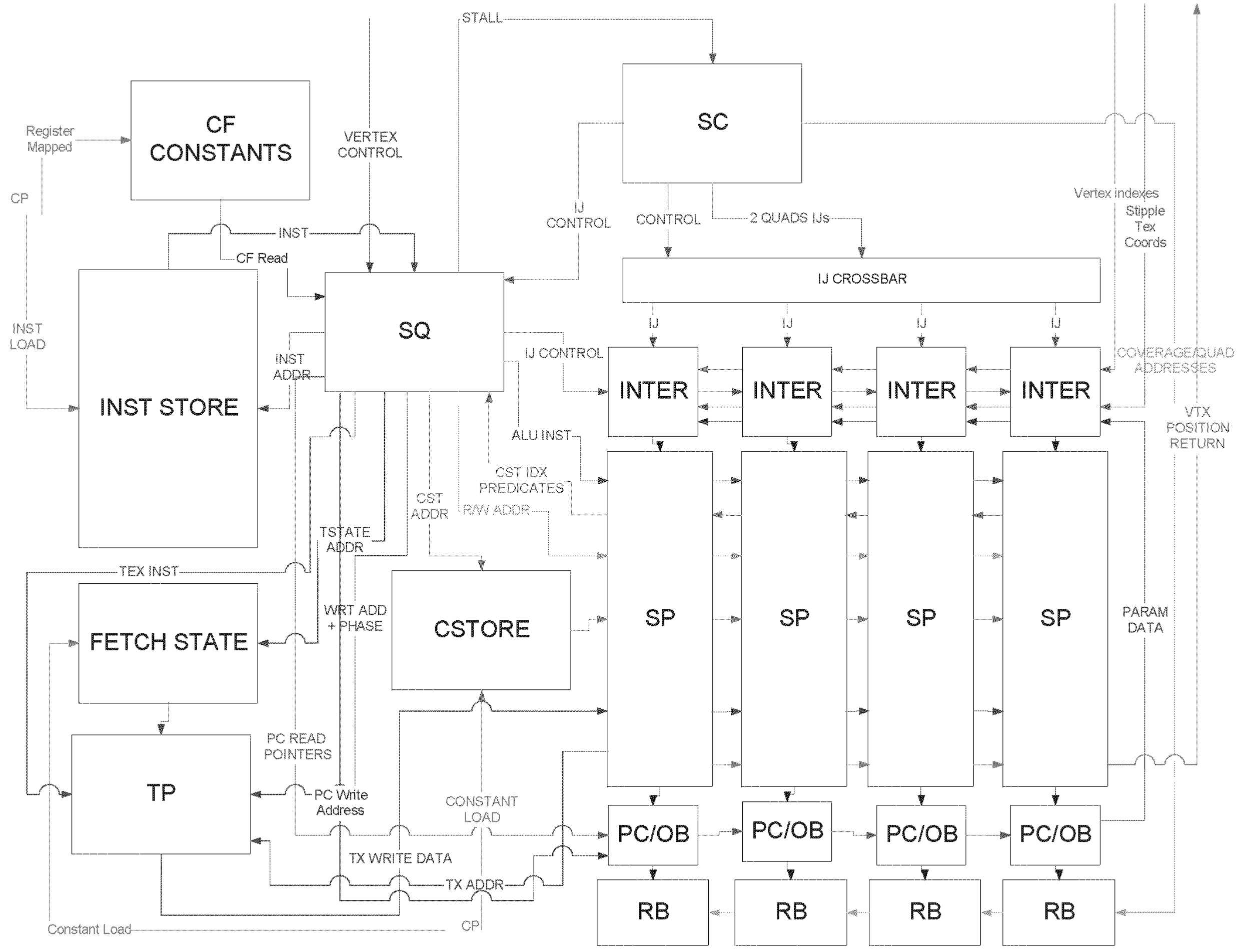


Figure 1: General Sequencer overview

1.1 Top Level Block Diagram

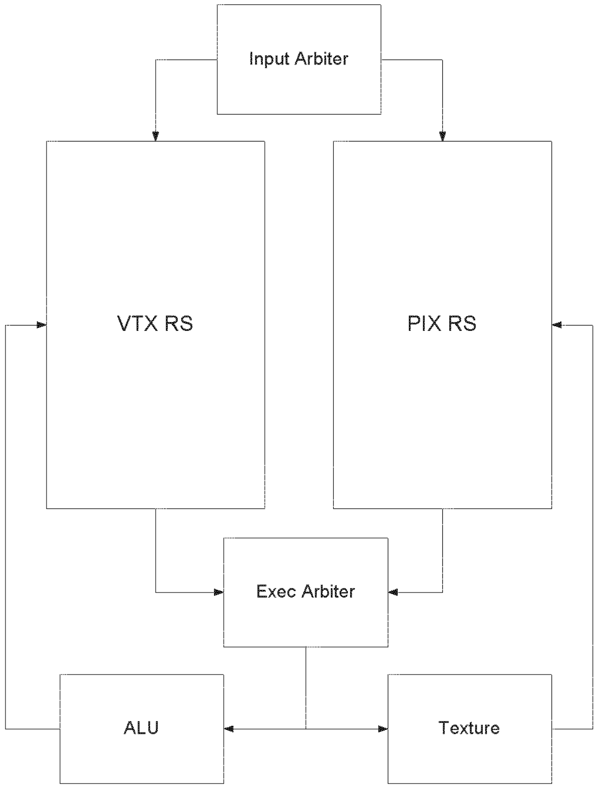


Figure 2: Reservation stations and arbiters

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).

1.2 Data Flow graph (SP)

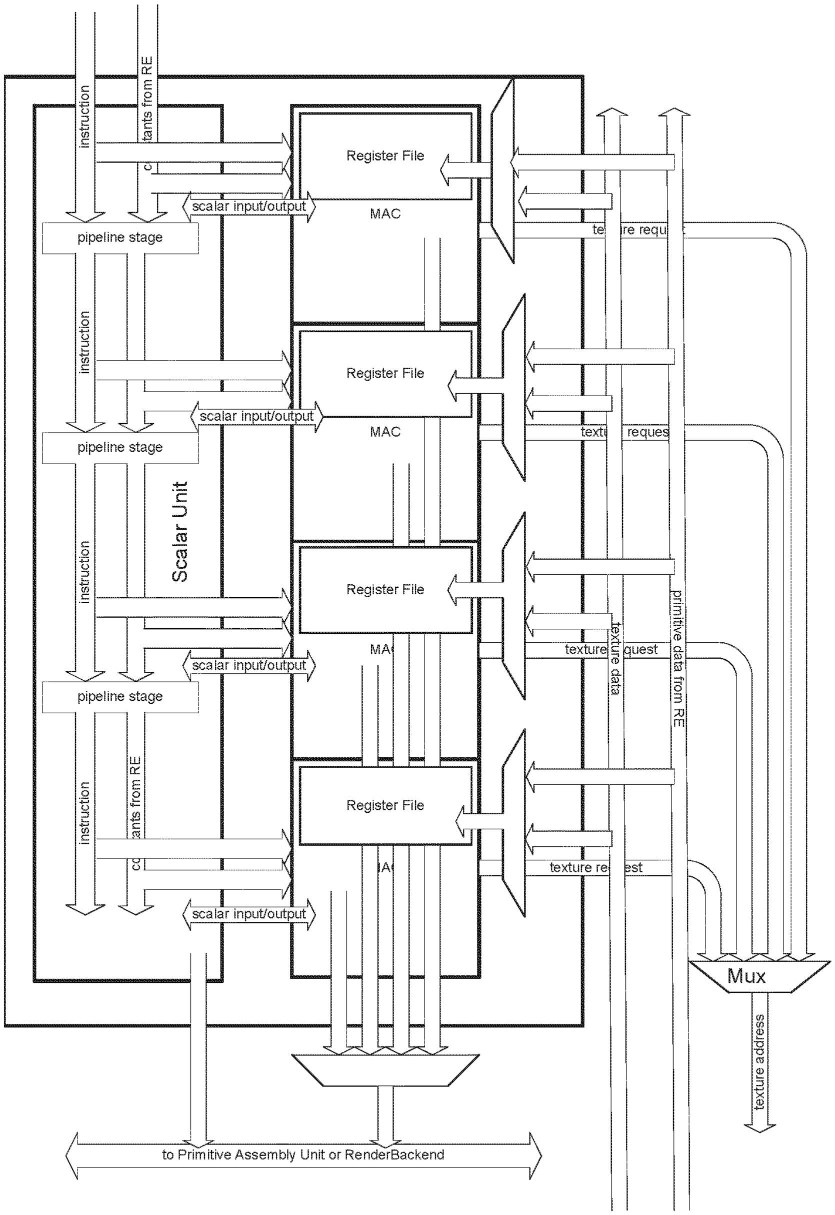


Figure 3: The shader Pipe

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

1.3 Control Graph

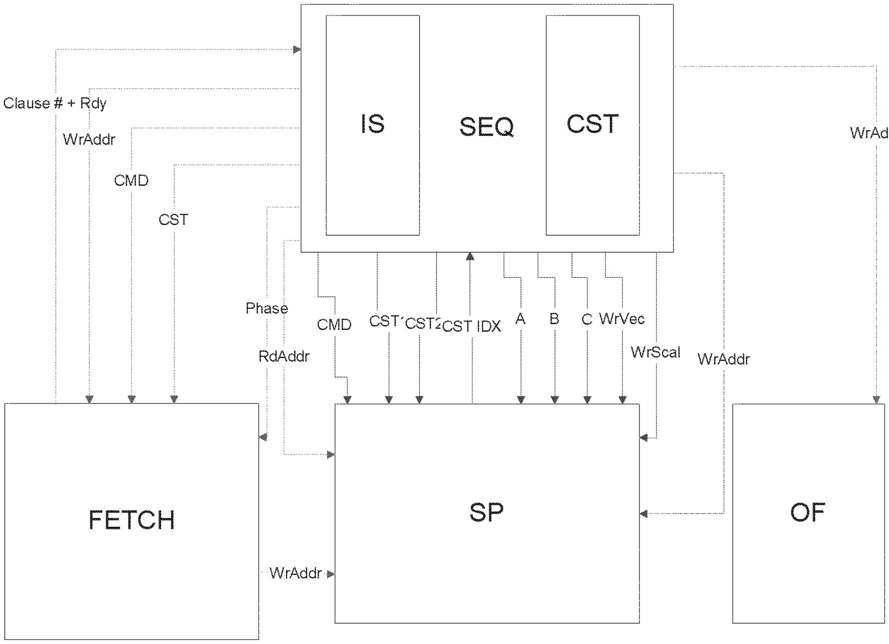


Figure 4: Sequencer Control interfaces

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.

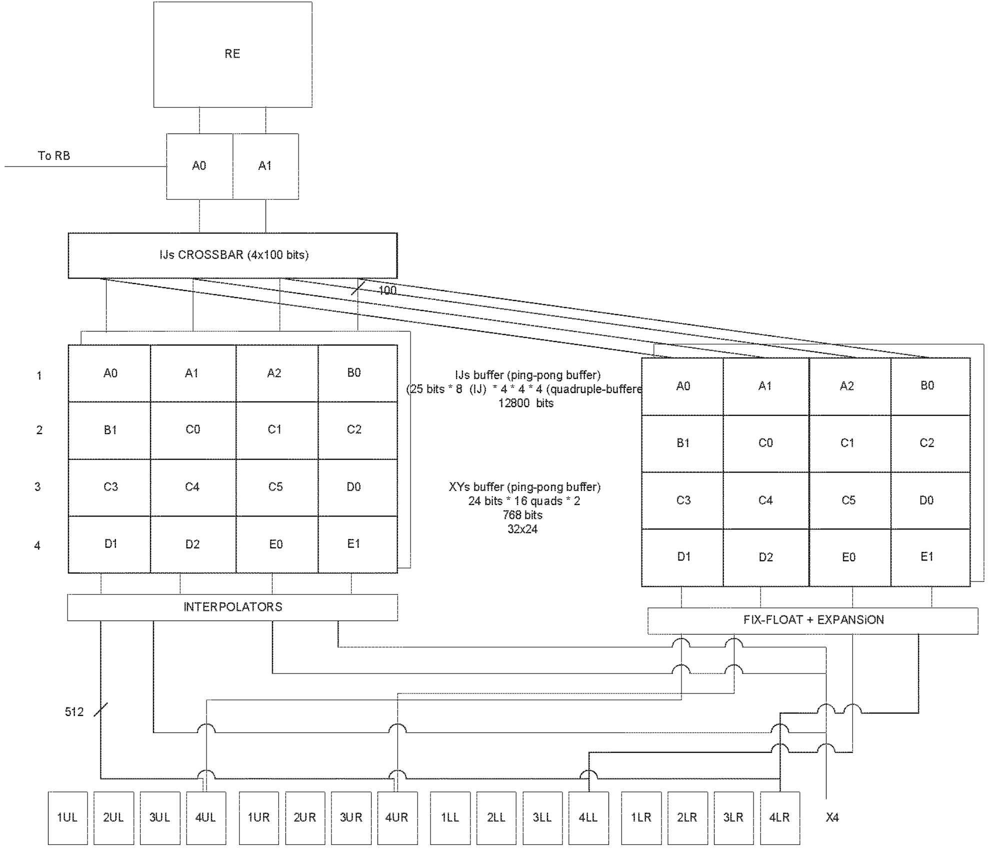


Figure 5: Interpolation buffers

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19	T20	T21	T22	T23
WRITES																								
SP 0	A0	A0	XY A0	B1	B1	XY B1	C3	C3	XY C3				D1	D1	XY D1									
SP 1	A1	A1	XY A1				C0	C0	XY C0	C4	C4	XY C4	D2	D2	XY D2									
SP 2	A2	A2	XY A2				C1	C1	XY C1	C5	C5	XY C5				E0	E0	XY E0						
SP 3				B0	B0	XY B0	C2	C2	XY C2				D0	D0	XY D0	E1	E1	XY E1						
READS																								
SP 0	XY 0-3	XY 16-19	XY 32-35	XY 48-51	A0	B1	C3	D1					A0	B1	C3	D1					V 0-3	V 16-19	V 32-35	V 48-51
SP 1	XY 4-7	XY 20-23	XY 36-39	XY 52-55	A1		C4	D2		C0			A1		C4	D2		C0			V 4-7	V 20-23	V 36-39	V 52-55
SP 2	XY 8-11	XY 24-27	XY 40-43	XY 56-59	A2		C5			C1		E0	A2		C5			C1		E0	V 8-11	V 24-27	V 40-43	V 56-59
SP 3	XY 12-15	XY 28-31	XY 44-47	XY 60-63					B0	C2	D0	E1					B0	C2	D0	E1	V 12-15	V 28-31	V 44-47	V 60-63
	XY				P1								P2								VTX			

Figure 6: Interpolation timing diagram

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.

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.

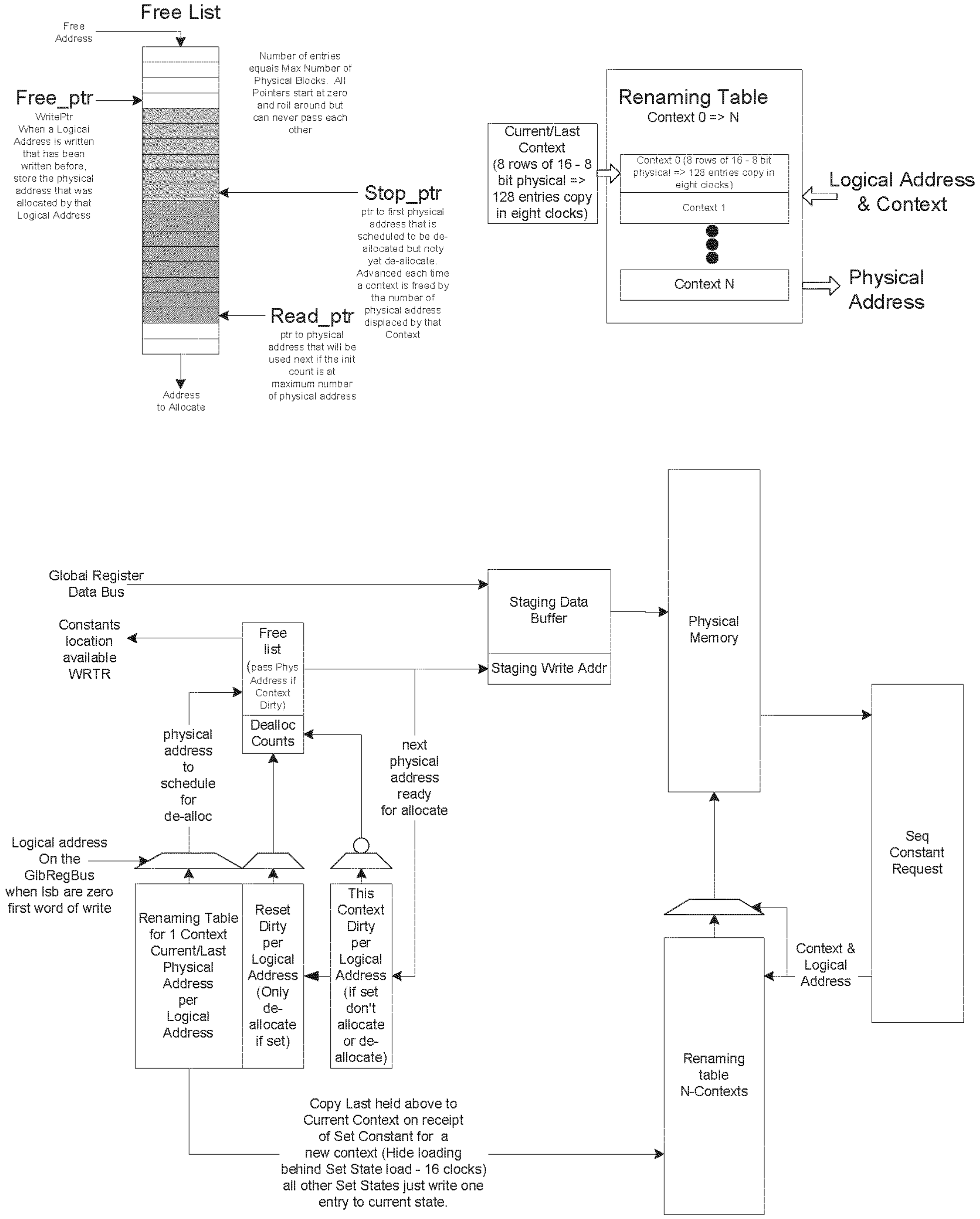
The constant memory update will be incremental, the driver only need to update the constants that actually changed between the two state changes.

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 Figure 8: De-allocation mechanism~~Figure 8: De-allocation mechanism~~Figure 8: De-~~allocation mechanism~~). Note that in the case a state is cleared a value 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.

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).



Free List
Free Address
Free_ptr
WritePtr
When a Logical Address is written that has been written before, store the physical address that was allocated by that Logical Address
Number of entries equals Max Number of Physical Blocks. All Pointers start at zero and roll around but can never pass each other
Stop_ptr
ptr to first physical address that is scheduled to be de-allocated but noty yet de-allocate. Advanced each time a context is freed by the number of physical address displaced by that Context
Read_ptr
ptr to physical address that will be used next if the init count is at maximum number of physical address
Address to Allocate
Renaming Table
Context 0 => N
Current/Last Context (8 rows of 16 - 8 bit physical => 128 entries copy in eight clocks)
Context 0 (8 rows of 16 - 8 bit physical => 128 entries copy in eight clocks)
Context 1
Context N
Logical Address & Context
Physical Address
Global Register Data Bus
Staging Data Buffer
Staging Write Addr
Physical Memory
Constants location available WRTR
Free list
(pass Phys Address if Context Dirty)
Dealloc Counts
physical address to schedule for de-alloc
next physical address ready for allocate
Logical address On the GlbRegBus when lsb are zero first word of write
Renaming Table for 1 Context Current/Last Physical Address per Logical Address
Reset Dirty per Logical Address (Only de-allocate if set)
This Context Dirty per Logical Address (If set don't allocate or de-allocate)
Seq Constant Request
Context & Logical Address
Renaming table N-Contexts
Copy Last held above to Current Context on receipt of Set Constant for a new context (Hide loading behind Set State load - 16 clocks) all other Set States just write one entry to current state.

Figure 7: Constant management

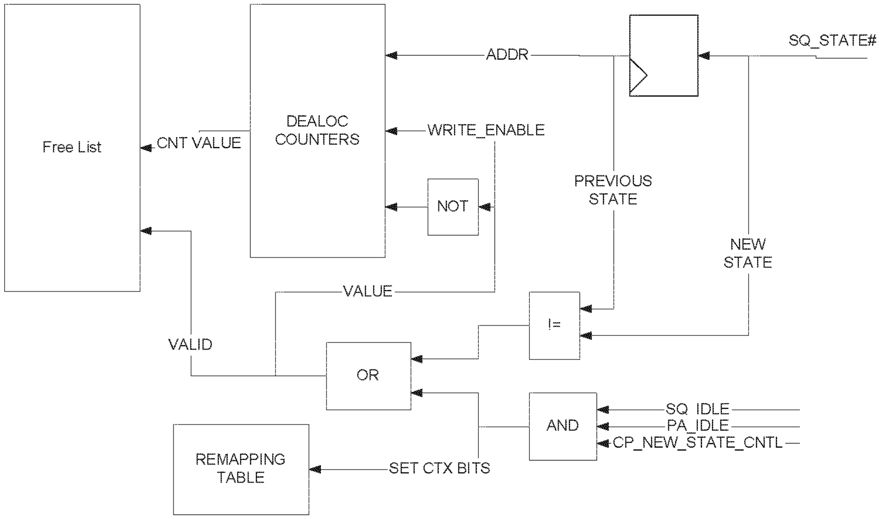


Figure 8: De-allocation mechanism for R400LE

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 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.

5.3.4 *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.

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 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 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).

```
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.

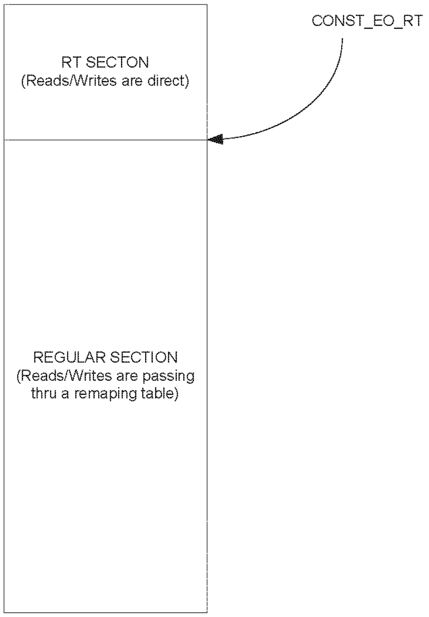

CONST_EO_RT
RT SECTON
(Reads/Writes are direct)
REGULAR SECTION
(Reads/Writes are passing thru a remaping table)

Figure 9: 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
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 …16	15…12	11 … 0
0001	Addressing	RESERVED	Instructions type + serialize (9 instructions)	Count	Exec Address

Execute_End					
47 … 44	43	40 … 34	33 …16	15…12	11 … 0
0010	Addressing	RESERVED	Instructions type + serialize (9 instructions)	Count	Exec Address

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	33…16	15 …12	11 … 0
0011	Addressing	Condition	Boolean address	Instructions type + serialize (9 instructions)	Count	Exec Address

Conditional_Execute_End						
47 … 44	43	42	41 … 34	33…16	15 …12	11 … 0
0100	Addressing	Condition	Boolean address	Instructions type + serialize (9 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	33…16	15…12	11 … 0
0101	Addressing	Condition	RESERVED	Predicate vector	Instructions type + serialize (9 instructions)	Count	Exec Address

Conditional_Execute_Predicates_End							
47 … 44	43	42	41 … 36	35 … 34	33…16	15…12	11 … 0
0110	Addressing	Condition	RESERVED	Predicate vector	Instructions type + serialize (9 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.

Conditional_Execute_Predicates_No_Stall							
47 … 44	43	42	41 … 36	35 … 34	33…16	15…12	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	33…16	15…12	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	15…13	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	35…34	33… 22	21	20 … 16	15…13	12 … 0
1000	Addressing	Cond	RESERVED	Predicate Vector	RESERVED	Pred 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 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

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 address	FW only	RESERVED	Force Jump	Jump 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	42…41	40 … 3	2…0
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)
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)
- 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:

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 non-masked 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.

{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

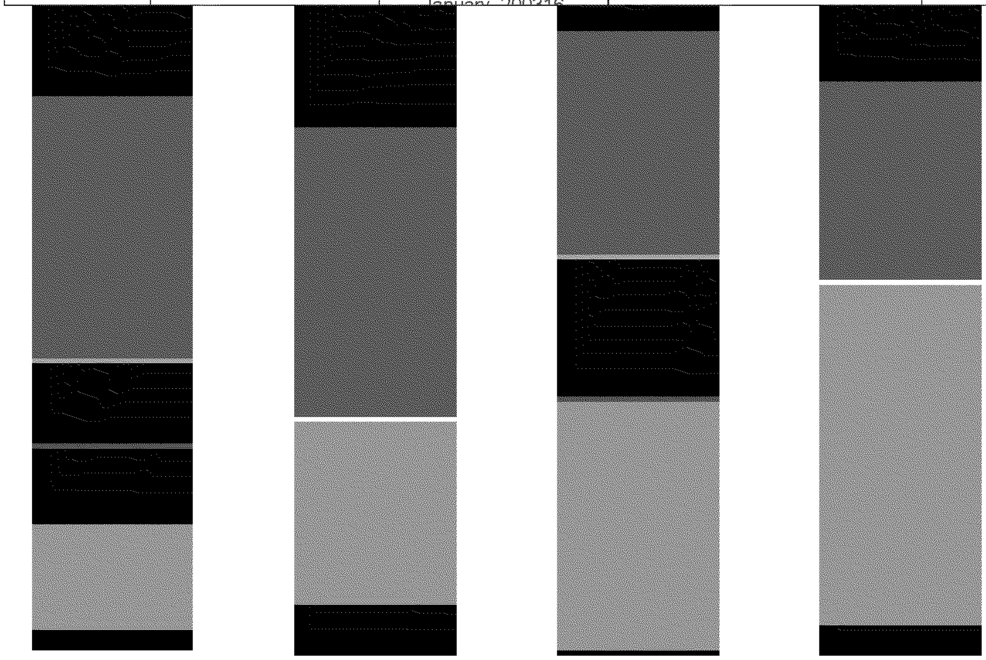
~~Multipass vertex shaders (HOS)~~

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

Formatted: Bullets and Numbering

~~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.



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.

Formatted: Bullets and Numbering

~~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.

Formatted: Bullets and Numbering

~~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 arbiters, 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.

Formatted: Bullets and Numbering

~~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.

Formatted: Bullets and Numbering

~~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.

Formatted: Bullets and Numbering

~~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).

Formatted: Bullets and Numbering

~~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.

$$P0 = A + I(0) * (B - A) + J(0) * (C - A)$$
$$P1 = A + I(1) * (B - A) + J(1) * (C - A)$$
$$P2 = A + I(2) * (B - A) + J(2) * (C - A)$$
$$P3 = A + I(3) * (B - A) + J(3) * (C - A)$$

P0	P1
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.

Formatted: Bullets and Numbering

~~15.1~~14.1 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.~~15. Staging Registers

Formatted: Bullets and Numbering

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:

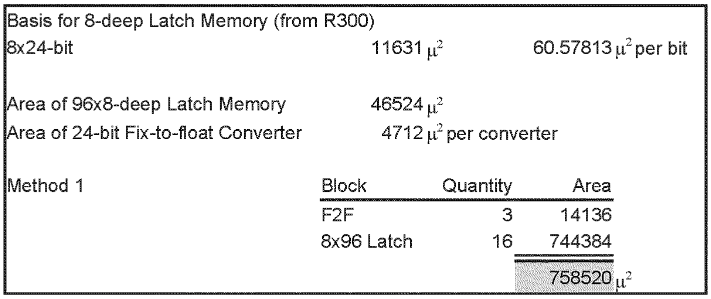
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 || 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 || 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 || 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

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 11~~Figure 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 10~~Figure 10Figure 10~~.



Basis for 8-deep Latch Memory (from 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 μ^2 per converter		
Method 1	Block	Quantity	Area
	F2F	3	14136
	8x96 Latch	16	744384
			758520 μ^2

Figure 10:Area Estimate for VGT to Shader Interface

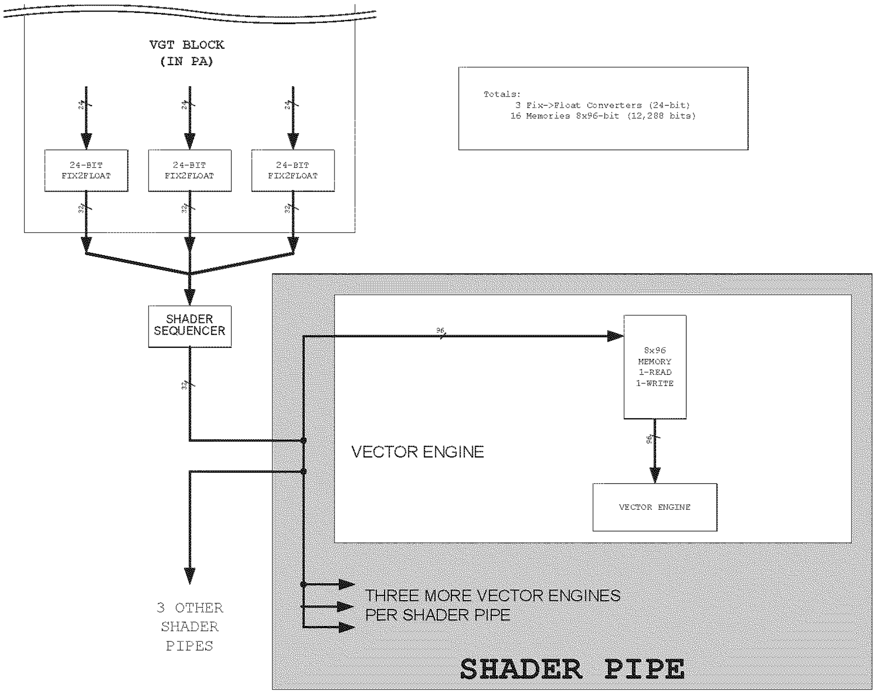


Figure 11:VGT to Shader Interface

Formatted: Bullets and Numbering

~~17.~~16. 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 00100000000 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~~16.1 Export restrictions

Formatted: Bullets and Numbering

~~17.1.1~~16.1.1 *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~~16.1.2 *Vertex exports:*

Formatted: Bullets and Numbering

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~~16.1.3 *Pass thru exports:*

Formatted: Bullets and Numbering

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

```
Alloc ~~4~~1 thru 5 (max export offset + 1, for example if using EM4 alloc size 5)~~(or 8)~~
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.

Formatted

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:

.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 EXaddress:

.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 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:

Formatted

1) Index value >= Max Index value
2) bit 31 != 0 (negative index)

Formatted: Bullets and Numbering

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, ~~**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~~ 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~~16.2 Arbitration restrictions

Formatted: Bullets and Numbering

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.
4) 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
5) Cannot execute a texture clause if texture reads are pending
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.~~17. Export Types

Formatted: Bullets and Numbering

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~~17.1 Vertex Shading

Formatted: Bullets and Numbering

0:15	- 16 parameter cache
16:31	- Empty (Reserved?)
32	- Export Address
33:37	- 5 vertex exports to the frame buffer and index
38:4~~7~~6	- Empty
47	- Debug Address
48:52	- 5 debug export (interpret as normal memory export)
53:59	- Empty
60	- export addressing mode
61	- Empty
62	- position
63	- sprite size export that goes with position export

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

~~18.2~~17.2 Pixel Shading

Formatted: Bullets and Numbering

0	- Color for buffer 0 (primary)
1	- Color for buffer 1
2	- Color for buffer 2
3	- Color for buffer 3
4:15	- Empty
16	- Buffer 0 Color/Fog (primary)
17	- Buffer 1 Color/Fog

18	- Buffer 2 Color/Fog
19	- Buffer 3 Color/Fog
20:31	- Empty
32	- ~~-~~Export Address
33:37	- 5 exports for multipass pixel shaders.
38:4~~7~~6	- 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

Formatted: Bullets and Numbering

~~19.~~18. Special Interpolation modes

~~19.1~~18.1 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~~ 4x128 memories (one for each of three vertices x ~~16~~ 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~~. 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.

Formatted: Bullets and Numbering

~~19.2~~18.2 Sprites/ XY screen coordinates/ FB information

XY screen coordinates may be needed in the shader program. This functionality is controlled by the param_gen~~n_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

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~~18.3 Auto generated counters

Formatted: Bullets and Numbering

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:

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~~18.3.1 *Vertex shaders*

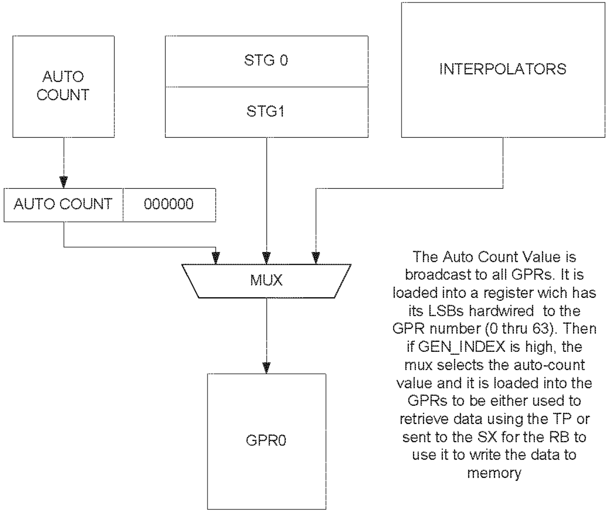
Formatted: Bullets and Numbering

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~~18.3.2 *Pixel shaders*

Formatted: Bullets and Numbering

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.



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

Formatted: Bullets and Numbering

~~20.~~19. 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.

Formatted: Bullets and Numbering

~~20.1~~19.1 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.

Formatted: Bullets and Numbering

~~21.~~20. 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 18.2~~18.219.2~~ for details on how to control the interpolation in this mode.

Formatted: Bullets and Numbering

~~21.1~~20.1 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.

Formatted: Bullets and Numbering

~~22.~~21. Registers

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

~~23.~~22. Interfaces

Formatted: Bullets and Numbering

~~23.1~~22.1 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→SPx it means that SQ is going to broadcast the same information to all SP instances.

~~23.2~~22.2 SC to SP Interfaces

Formatted: Bullets and Numbering

~~23.2.1~~22.2.1 *SC_SP#*

Formatted: Bullets and Numbering

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
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

		interpolation 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~~22.2.2 SC_SQ

Formatted: Bullets and Numbering

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, 2nd clk could be all zeroes

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

Name	BitField	Bits	Description
1st 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

			=> TBD
SC_SQ_state_id	[8:6]	3	State/constant pointer (6*3+3)
SC_SQ_pc_dealloc	[11:9]	3	Deallocation token for the Parameter Cache
SC_SQ_new_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_mask	[16:13]	4	Quad Write mask left to right SP0 => SP3
SC_SQ_end_of_prim	17	1	End Of the primitive
SC_SQ_pix_mask	[33:18]	16	Valid bits for all pixels SP0=>SP3 (UL,UR,LL,LR)
SC_SQ_provok_vtx	[35:34]	2	Provoking vertex for flat shading
SC_SQ_lod_correct_0	[44:36]	9	LOD correction for quad 0 (SP0) (9 bits per quad)
SC_SQ_lod_correct_1	[53:45]	9	LOD correction for quad 1 (SP1) (9 bits per quad)
2nd Clock Transfer			
SC_SQ_lod_correct_2	[8:0]	9	LOD correction for quad 2 (SP2) (9 bits per quad)
SC_SQ_lod_correct_3	[17:9]	9	LOD correction for quad 3 (SP3) (9 bits per quad)
SC_SQ_pc_ptr0	[28:18]	11	Parameter Cache pointer for vertex 0
SC_SQ_pc_ptr1	[39:29]	11	Parameter Cache pointer for vertex 1
SC_SQ_pc_ptr2	[50:40]	11	Parameter Cache pointer for vertex 2
SC_SQ_prim_type	[53:51]	3	Stippled line and Real time command need to load tex cords from alternate buffer 000: Sprite (point) 001: Line 010: Tri_rect 100: Realtime Sprite (point) 101: Realtime Line 110: Realtime Tri_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.)

~~23.2.3~~22.2.3 *SQ to SX(SP): Interpolator bus*

Formatted: Bullets and Numbering

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~~22.2.4 *SQ to SP: Staging Register Data*

Formatted: Bullets and Numbering

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_vsr_double~~	SQ→SPx	3~~1~~	Staging register read address~~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~~22.2.5 *VGT to SQ : Vertex interface*

Formatted: Bullets and Numbering

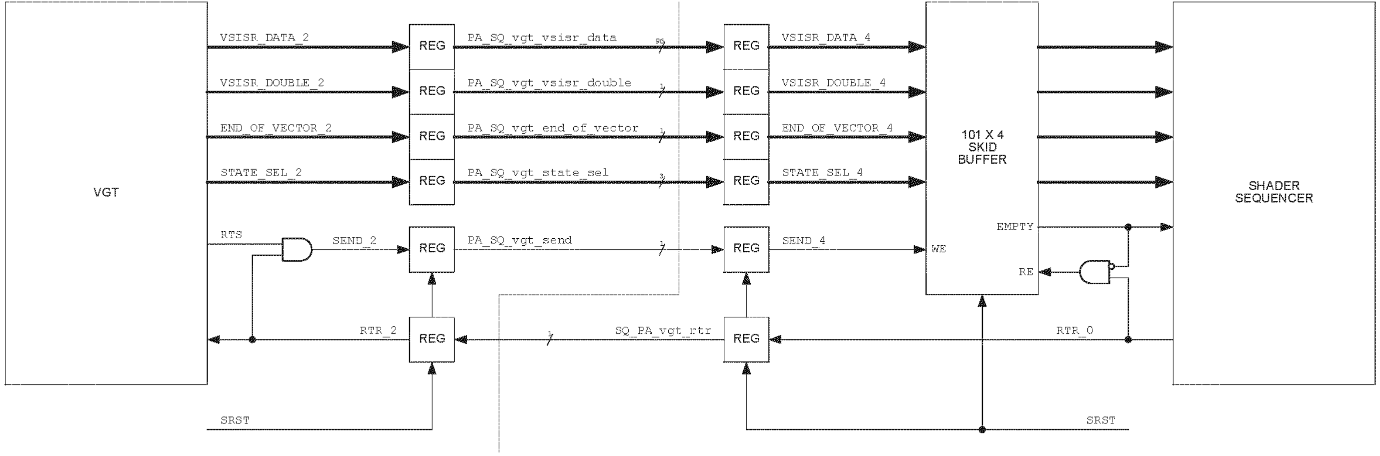
~~23.2.5.1~~22.2.5.1 Interface Signal Table

The area difference between the two methods is not sufficient to warrant complicating the interface or the state requirements of the VSISRs. **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.** 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~~22.2.5.2 Interface Diagrams

Formatted: Bullets and Numbering


VGT
VSISR_DATA_2
VSISR_DOUBLE_2
END_OF_VECTOR_2
STATE_SEL_2
RTS
SEND_2
RTR_2
SRST
REG
REG
REG
REG
REG
REG
PA_SQ_vgt_vsisr_data
96
PA_SQ_vgt_vsisr_double
1
PA_SQ_vgt_end_of_vector
1
PA_SQ_vgt_state_sel
3
PA_SQ_vgt_send
1
SQ_PA_vgt_rtr
1
REG
REG
REG
REG
REG
REG
VSISR_DATA_4
VSISR_DOUBLE_4
END_OF_VECTOR_4
STATE_SEL_4
SEND_4
101 X 4
SKID
BUFFER
WE
EMPTY
RE
RTR_0
SRST
SHADER
SEQUENCER

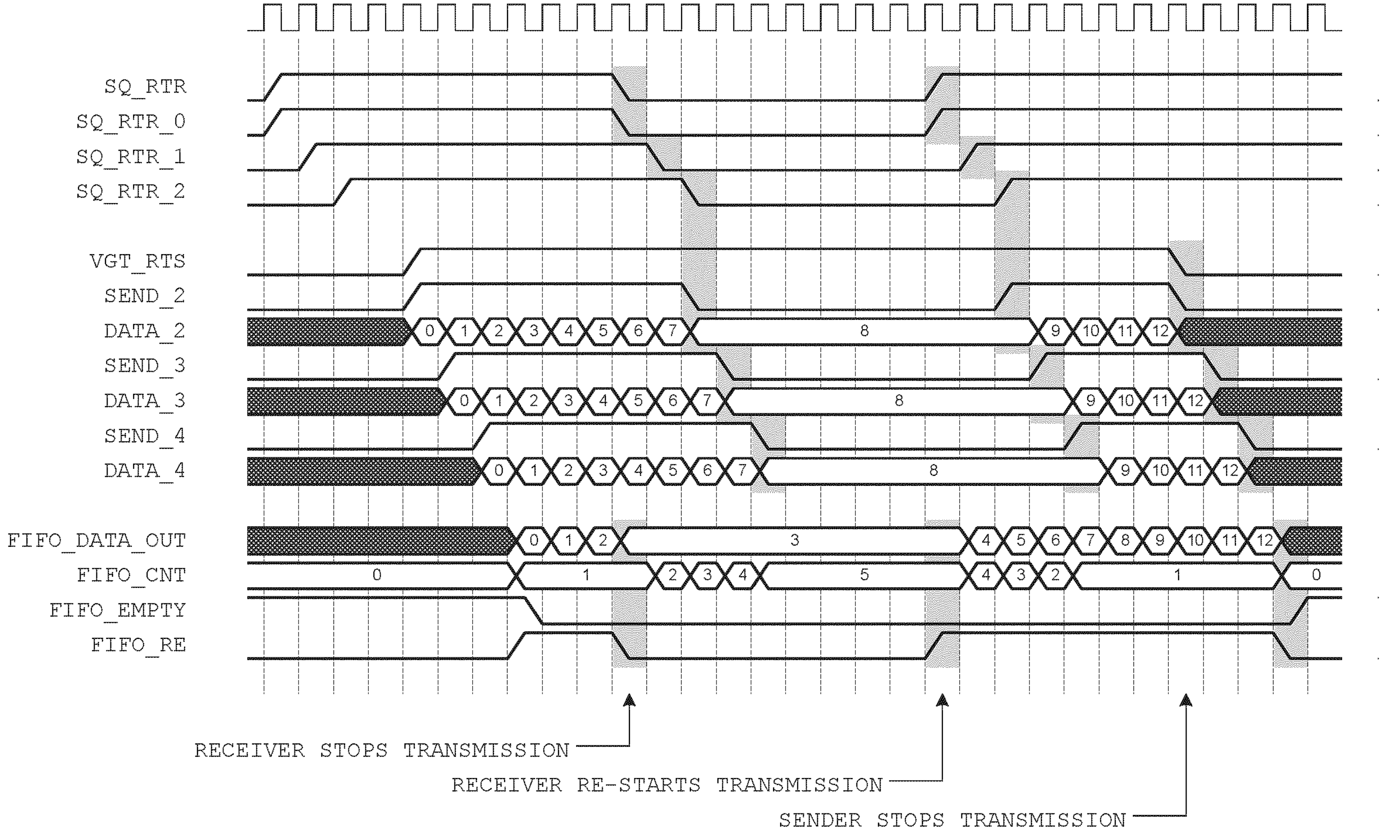


Figure 1. Detailed Logical Diagram for PA_SQ_vgt Interface.

~~23.2.6~~22.2.6 SQ to SX: Control bus

Formatted: Bullets and Numbering

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)
  - 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~~22.2.7 SX to SQ : Output file control

Formatted: Bullets and Numbering

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

			pixels in a clause) ... 64: 128K-bits available (16 128-bit entries for each of 64 pixels) 65-127: RESERVED

~~23.2.8~~22.2.8 SQ to TP: Control bus

Formatted: Bullets and Numbering

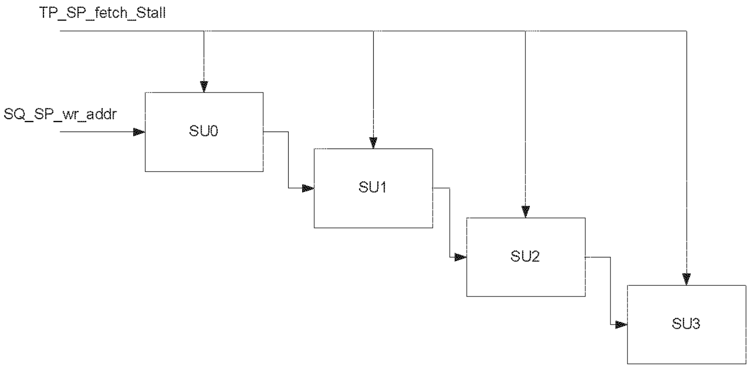
Once every clock, the fetch unit sends to the sequencer on which RS line it is now working and if the data in the GPRs is ready or not. This way the sequencer can update the fetch valid bits flags for the reservation station. The sequencer also provides the instruction and constants for the fetch to 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→ 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~~22.2.9 TP to SQ: Texture stall

Formatted: Bullets and Numbering

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



Name	Direction	Bits	Description
TP_SQ_fetch_stall	TP→ SQ	1	Do not send more texture request if asserted

~~23.2.10 SQ to SP: Texture stall~~

Formatted: Bullets and Numbering

~~23.2.11~~22.2.10 SQ to SP: GPR and auto counter

Formatted: Bullets and Numbering

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	SQ→SPx	4	Write Enable for the GPRs of SP3 for PS and PV
SQ_SP0_gpr_int_wr_en	SQ→SPx	1	Write Enable for the GPRs of SP0 for Inputs (interp/vtx)
SQ_SP1_gpr_int_wr_en	SQ→SPx	1	Write Enable for the GPRs of SP1 for Inputs (interp/vtx)
SQ_SP2_gpr_int_wr_en	SQ→SPx	1	Write Enable for the GPRs of SP2 for Inputs (interp/vtx)
SQ_SP3_gpr_int_wr_en~~SQ_SP3_gpr_wr_en~~	SQ→SPx~~SQ→SPx~~	1~~4~~	Write Enable for the GPRs of SP3 for Inputs (interp/vtx)~~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

~~23.2.12~~22.2.11 SQ to SPx: Instructions

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_instr_start	SQ→SPx	1	Instruction start
SQ_SP_instr	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 all 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 all 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 all 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_pred_override	SQ→SP0	4	0: Use per channel RGBA field (enables the per channel logic, if not set only pay attention to the 11 seting). 1: Use GPR
SQ_SP1_pred_override	SQ→SP1	4	0: Use per channel RGBA field (enables the per channel logic, if not set only pay attention to the 11 seting). 1: Use GPR
SQ_SP2_pred_override	SQ→SP2	4	0: Use per channel RGBA field (enables the per channel logic, if not set only pay attention to the 11 seting). 1: Use GPR
SQ_SP3_pred_override	SQ→SP3	4	0: Use per channel RGBA field (enables the per channel logic, if not set only pay attention to the 11

			seting). 1: Use GPR
SQ_SPx_exp_id	SQ→SPx	1	GPR ID
SQ_SPx_exporting	SQ→SPx	1	0: Not Exporting 1: Exporting
SQ_SPx_stall	SQ→SPx	1	Stall signal
SQ_SPx_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

~~23.2.13~~22.2.12 *SQ to SX: write mask interface (must be aligned with the SP data)*

Formatted: Bullets and Numbering

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~~22.2.13 *SP to SQ: Constant address load/ Predicate Set/Kill set*

Formatted: Bullets and Numbering

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~~22.2.14 *SQ to SPx: constant broadcast*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_const	SQ→SPx	128	Constant broadcast

~~23.2.16~~22.2.15 *SQ to CP: RBBM bus*

Formatted: Bullets and Numbering

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~~22.2.16 *CP to SQ: RBBM bus*

Formatted: Bullets and Numbering

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

~~23.2.18~~22.2.17 *SQ to CP: State report*

Formatted: Bullets and Numbering

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~~22.3 Example of control flow program execution

Formatted: Bullets and Numbering

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)
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	TYPE	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:

```
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.

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:

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 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 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 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.

Formatted: Bullets and Numbering

~~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?

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SQ

Version 2.10

Overview: 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 sub-blocks, and provides internal state diagrams.

AUTOMATICALLY UPDATED FIELDS:
Document Location: C:\perforce\r400\doc_lib\design\blocks\sq\R400_Sequencer.doc
Current Intranet Search Title: R400 Sequencer Specification

APPROVALS

Name/Dept	Signature/Date

Remarks:

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

Table Of Contents

Revision Changes:

Rev 0.1 (Laurent Lefebvre) Date: May 7, 2001	First draft.
Rev 0.2 (Laurent Lefebvre) Date : July 9, 2001	Changed the interfaces to reflect the changes in the SP. Added some details in the arbitration section.
Rev 0.3 (Laurent Lefebvre) Date : August 6, 2001	Reviewed the Sequencer spec after the meeting on August 3, 2001.
Rev 0.4 (Laurent Lefebvre) Date : August 24, 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.
Rev 0.5 (Laurent Lefebvre) Date : September 7, 2001	Added timing diagrams (Vic)
Rev 0.6 (Laurent Lefebvre) Date : September 24, 2001	Changed the spec to reflect the new R400 architecture. Added interfaces.
Rev 0.7 (Laurent Lefebvre) Date : October 5, 2001	Added constant store management, instruction store management, control flow management and data dependant predication.
Rev 0.8 (Laurent Lefebvre) Date : October 8, 2001	Changed the control flow method to be more flexible. Also updated the external interfaces.
Rev 0.9 (Laurent Lefebvre) Date : October 17, 2001	Incorporated changes made in the 10/18/01 control flow meeting. Added a NOP instruction, removed the conditional_execute_or_jump. Added debug registers.
Rev 1.0 (Laurent Lefebvre) Date : October 19, 2001	Refined interfaces to RB. Added state registers.
Rev 1.1 (Laurent Lefebvre) Date : October 26, 2001	Added SEQ→SP0 interfaces. Changed delta precision. Changed VGT→SP0 interface. Debug Methods added.
Rev 1.2 (Laurent Lefebvre) Date : November 16, 2001	Interfaces greatly refined. Cleaned up the spec.
Rev 1.3 (Laurent Lefebvre) Date : November 26, 2001	Added the different interpolation modes.
Rev 1.4 (Laurent Lefebvre) Date : December 6, 2001	Added the auto incrementing counters. Changed the VGT→SQ interface. Added content on constant management. Updated GPRs.
Rev 1.5 (Laurent Lefebvre) Date : December 11, 2001	Removed from the spec all interfaces that weren't directly tied to the SQ. Added explanations on constant management. Added PA→SQ synchronization fields and explanation.
Rev 1.6 (Laurent Lefebvre) Date : January 7, 2002	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.
Rev 1.7 (Laurent Lefebvre) Date : February 4, 2002	Added Real Time parameter control in the SX interface. Updated the control flow section.
Rev 1.8 (Laurent Lefebvre) Date : March 4, 2002	New interfaces to the SX block. Added the end of clause modifier, removed the end of clause instructions.
Rev 1.9 (Laurent Lefebvre) Date : March 18, 2002	Rearangement of the CF instruction bits in order to ensure byte alignement.
Rev 1.10 (Laurent Lefebvre) Date : March 25, 2002	Updated the interfaces and added a section on exporting rules.
Rev 1.11 (Laurent Lefebvre) Date : April 19, 2002	Added CP state report interface. Last version of the spec with the old control flow scheme
Rev 2.0 (Laurent Lefebvre) Date : April 19, 2002	New control flow scheme

Rev 2.01 (Laurent Lefebvre) Date : May 2, 2002	Changed slightly the control flow instructions to allow force jumps and calls.
Rev 2.02 (Laurent Lefebvre) Date : May 13, 2002	Updated the Opcodes. Added type field to the constant/pred interface. Added Last field to the SQ→SP instruction load interface.
Rev 2.03 (Laurent Lefebvre) Date : July 15, 2002	SP interface updated to include predication optimizations. Added the predicate no stall instructions,
Rev 2.04 (Laurent Lefebvre) Date :August 2, 2002	Documented the new parameter generation scheme for XY coordinates points and lines STs.
Rev 2.05 (Laurent Lefebvre) Date : September 10, 2002	Some interface changes and an architectural change to the auto-counter scheme.
Rev 2.06 (Laurent Lefebvre) Date : October 11, 2002	Widened the event interface to 5 bits. Some other little typos corrected.
Rev 2.07 (Laurent Lefebvre) Date : October 14, 2002	Loops, jumps and calls are now using a 13 bit address which allows to jump and call and loop around any control flow addresses (does not requires to be even anymore).
Rev 2.08 (Laurent Lefebvre) Date : October 16, 2002	Clarification updates after discussion with Clay.
Rev 2.09 (Laurent Lefebvre) Date : January 7, 2003	Corrected the SQ→SP staging register interface.
Rev 2.10 (Laurent Lefebvre) Date : April 8, 2003	Adding R500 modifications

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.

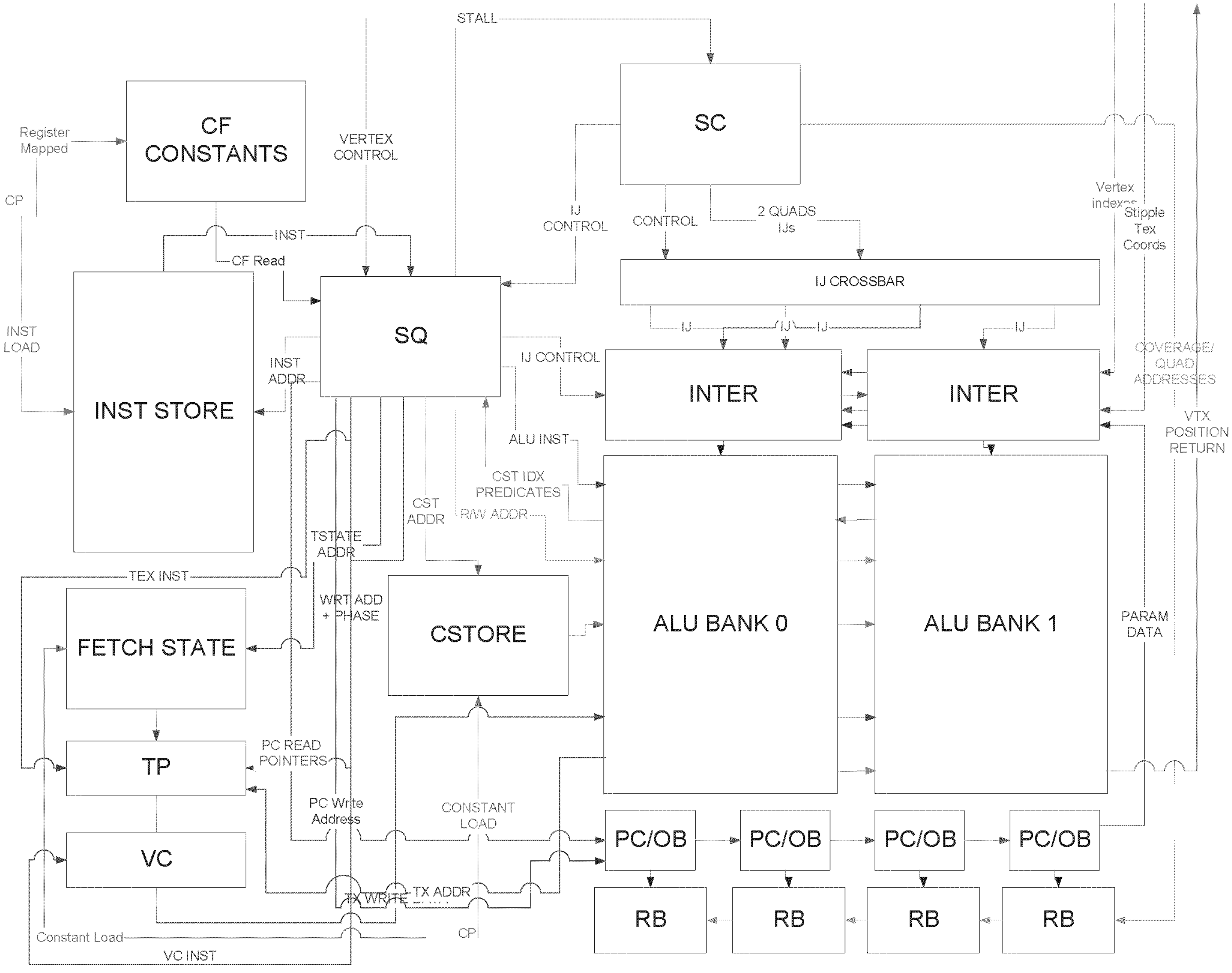


Figure 1: General Sequencer overview

1.1 Top Level Block Diagram

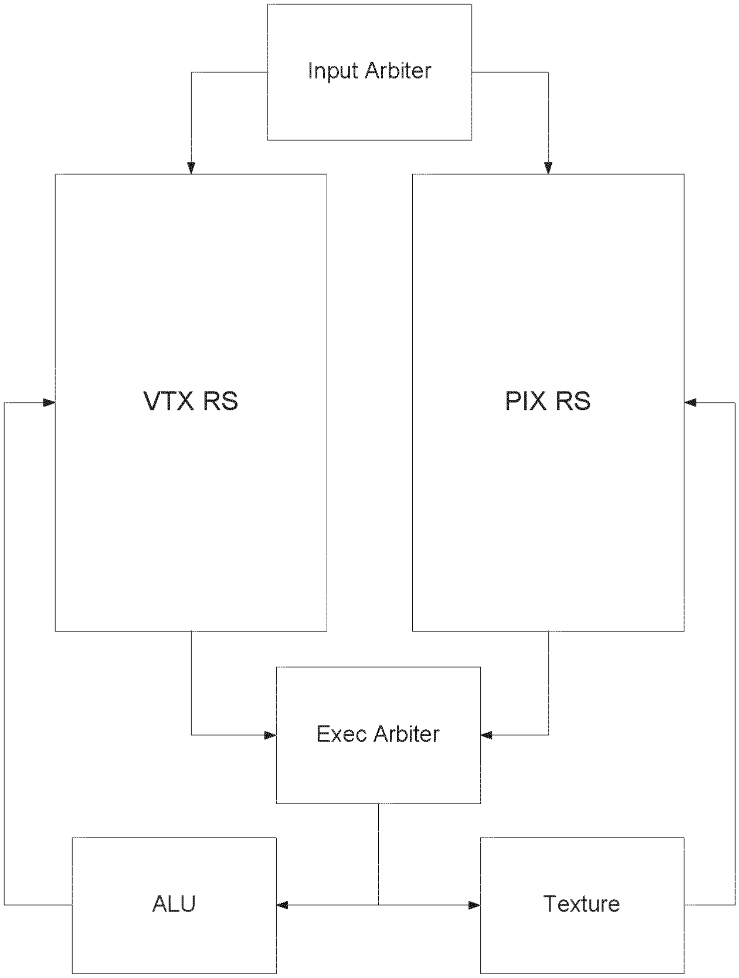


Figure 2: Reservation stations and arbiters

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).

1.2 Data Flow graph (SP)

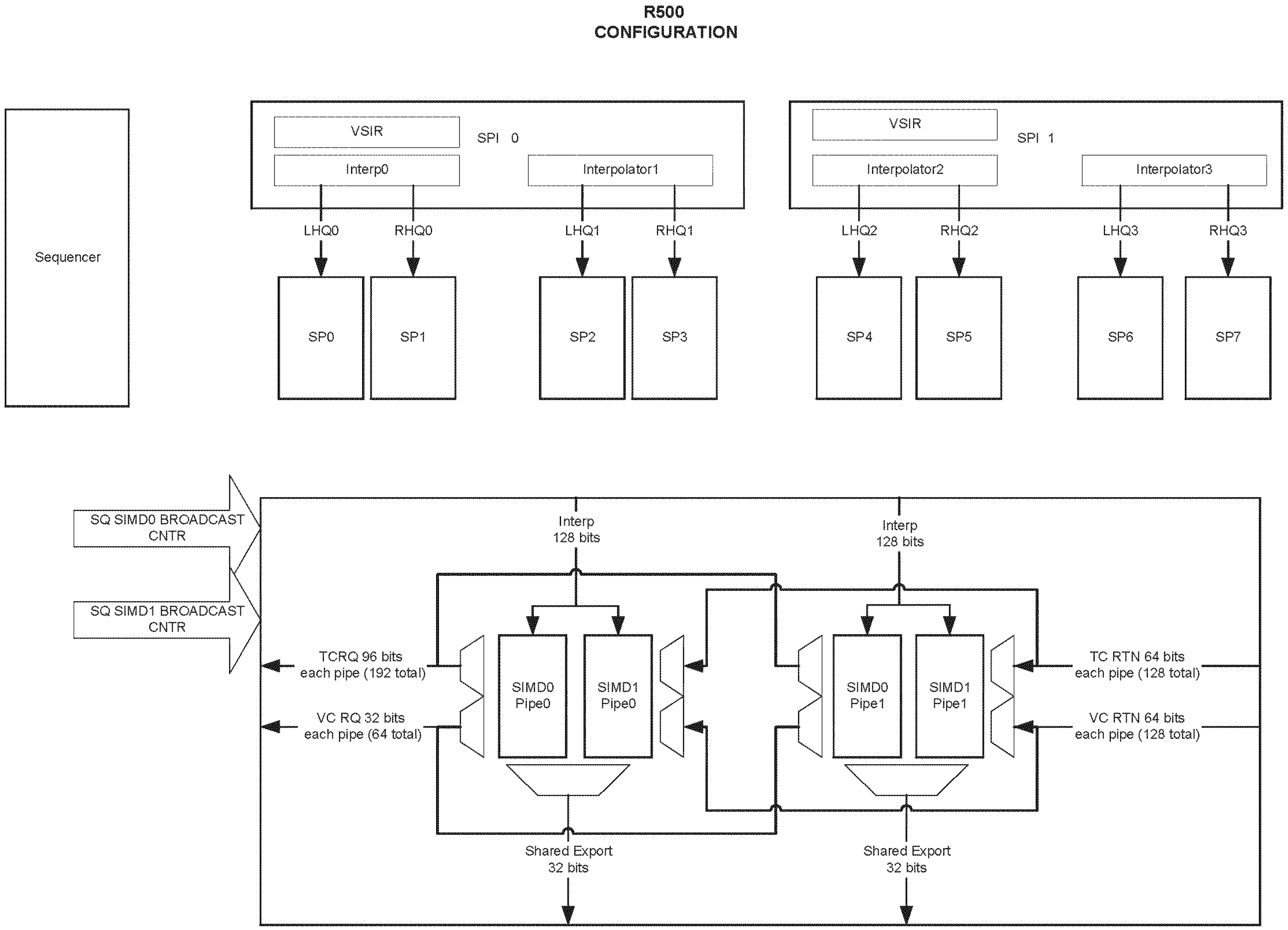


Figure 3: The shader Pipe

1.3 Control Graph

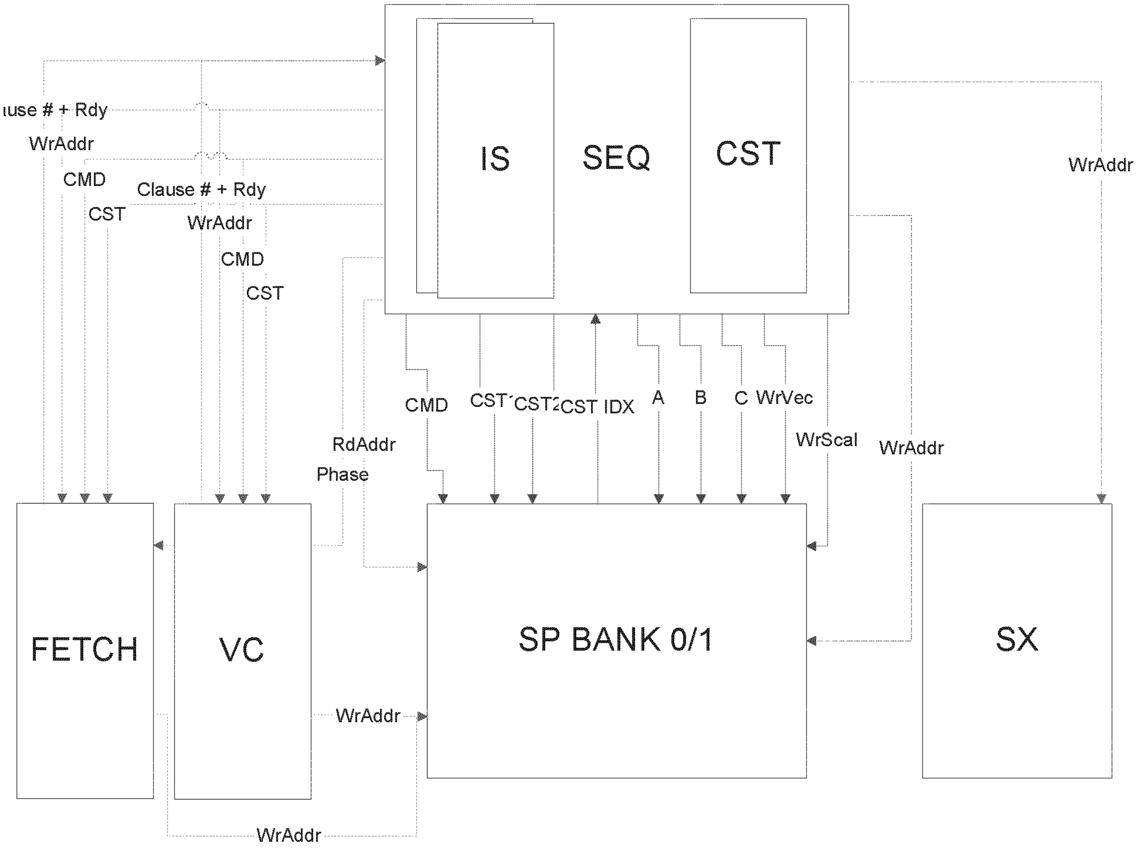


Figure 4: Sequencer Control interfaces

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.

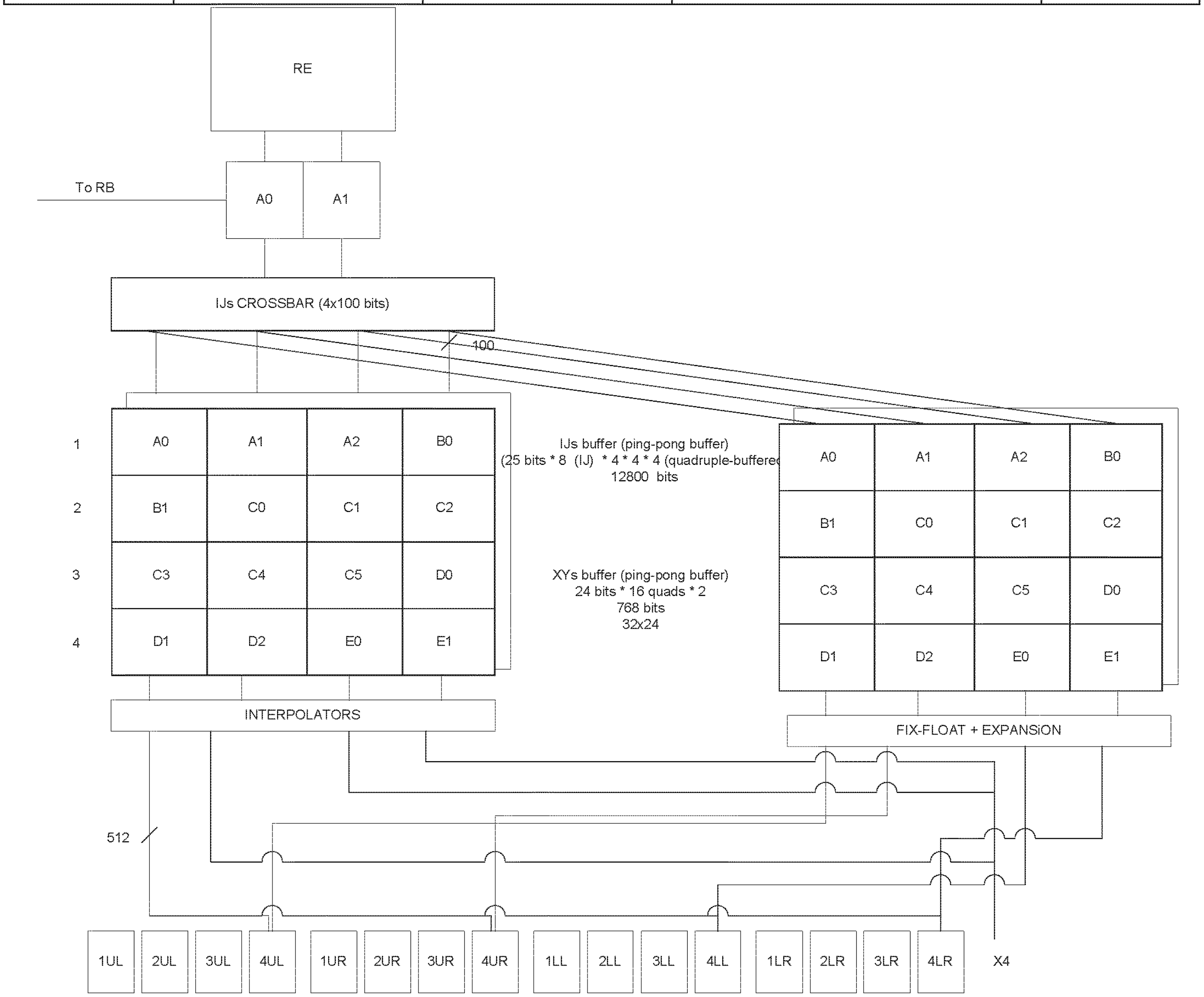


Figure 5: Interpolation buffers

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19	T20	T21	T22	T23
											WRITES													
SP 0	A0	A0	XY A0	B1	B1	XY B1	C3	C3	XY C3				D1	D1	XY D1									
SP 1	A1	A1	XY A1				C0	C0	XY C0	C4	C4	XY C4	D2	D2	XY D2									
SP 2	A2	A2	XY A2				C1	C1	XY C1	C5	C5	XY C5				E0	E0	XY E0						
SP 3				B0	B0	XY B0	C2	C2	XY C2				D0	D0	XY D0	E1	E1	XY E1						
											READS													
SP 0	XY 0-3	XY 16-19	XY 32-35	XY 48-51	A0	B1	C3	D1					A0	B1	C3	D1					V 0-3	V 16-19	V 32-35	V 48-51
SP 1	XY 4-7	XY 20-23	XY 36-39	XY 52-55	A1		C4	D2		C0			A1		C4	D2		C0			V 4-7	V 20-23	V 36-39	V 52-55
SP 2	XY 8-11	XY 24-27	XY 40-43	XY 56-59	A2		C5			C1		E0	A2		C5			C1		E0	V 8-11	V 24-27	V 40-43	V 56-59
SP 3	XY 12-15	XY 28-31	XY 44-47	XY 60-63					B0	C2	D0	E1					B0	C2	D0	E1	V 12-15	V 28-31	V 44-47	V 60-63
	XY				P1								P2								VTX			

Figure 6: Interpolation timing diagram

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 SIMD0
ALU 1 SIMD0 CF
ALU 1 SIMD0
Fetch CF
Fetch
VC CF
VC

ALU 0 SIMD1 CF
ALU 0 SIMD1
ALU 1 SIMD1 CF
ALU 1 SIMD1
Fetch CF
Fetch
VC CF
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 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

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 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).

```
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.

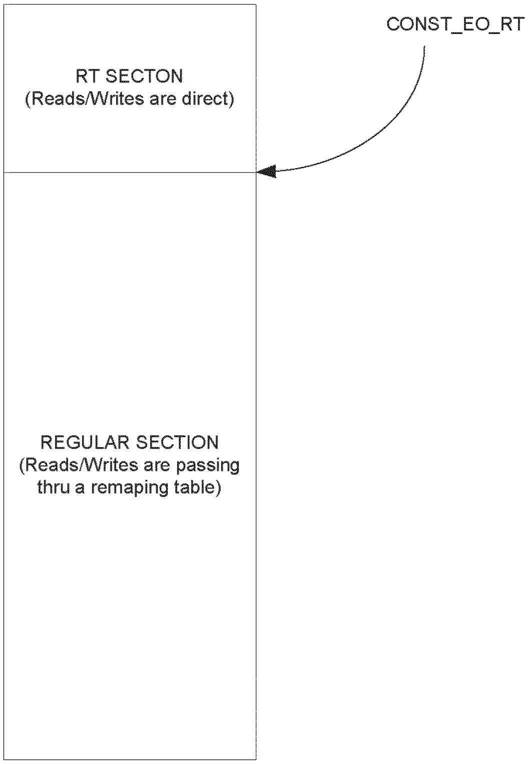

CONST_EO_RT
RT SECTON
(Reads/Writes are direct)
REGULAR SECTION
(Reads/Writes are passing thru a remaping table)

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
```

```
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	27 …16	15…12	11 … 0
0001	Addressing	RESERVED	Vertex Cache	Instructions type + serialize (6 instructions)	Count	Exec Address

Execute_End						
47 … 44	43	40 … 34	33 … 28	27 …16	15…12	11 … 0
0010	Addressing	RESERVED	Vertex Cache	Instructions type + serialize (6 instructions)	Count	Exec Address

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	43	42	41 … 34	33…28	27…16	15 …12	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	33…28	27…16	15 …12	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	33…28	27…16	15…12	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	33…28	27…16	15…12	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.

Conditional_Execute_Predicates_No_Stall								
47 … 44	43	42	41 … 36	35 … 34	33…28	27…16	15…12	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	33…28	27…16	15…12	11 … 0
1110	Addressing	Condition	RESERVED	Predicate vector	Vertex Cache	Instructions type + serialize (6 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	15…13	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	35…34	33… 22	21	20 … 16	15…13	12 … 0
1000	Addressing	Cond	RESERVED	Predicate Vector	RESERVED	Pred 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 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

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 address	FW only	RESERVED	Force Jump	Jump 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	42…41	40	39 … 3	2…0
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 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 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 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.

6.5 HW Detection of PV,PS

Because of the control program, the compiler cannot detect statically dependant instructions. In the case of non-masked 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.

{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).

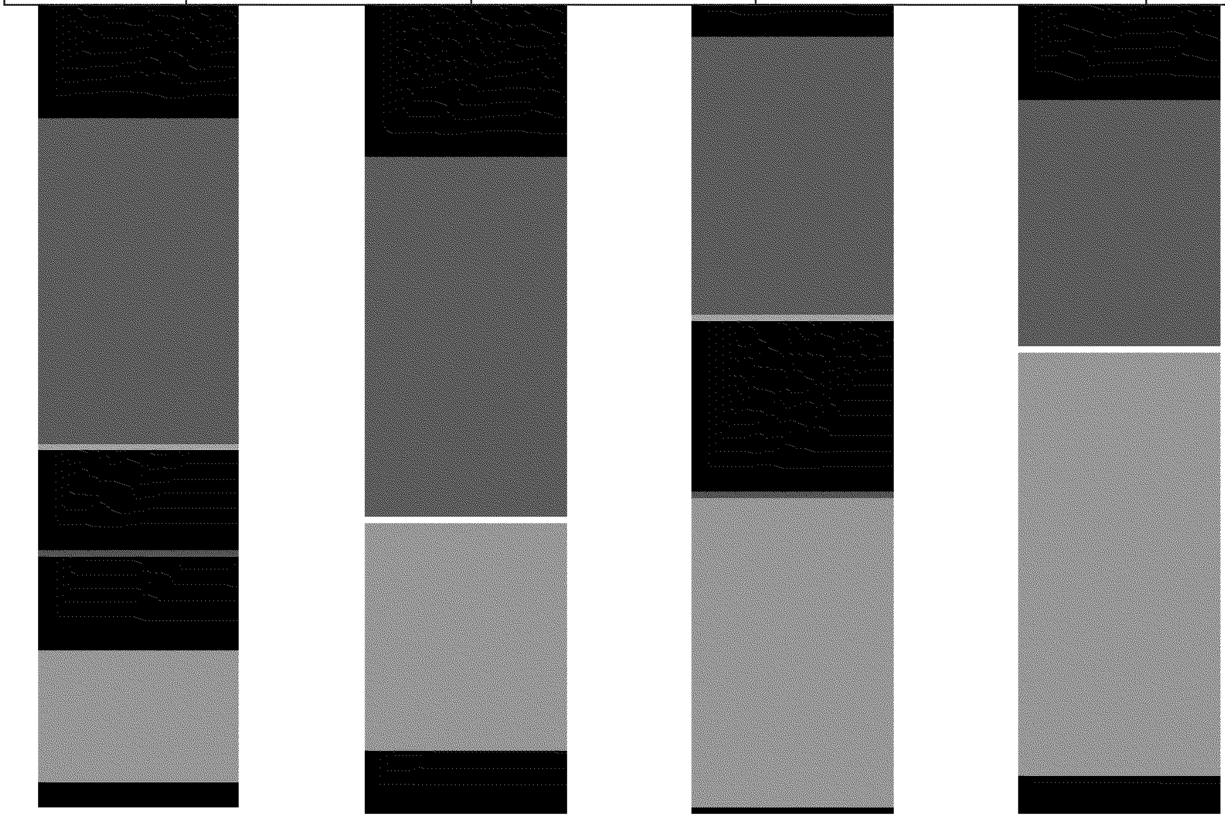
7. Pixel Kill Mask

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.



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.

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 arbiters, 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.

$$P0 = A + I(0) * (B - A) + J(0) * (C - A)$$
$$P1 = A + I(1) * (B - A) + J(1) * (C - A)$$
$$P2 = A + I(2) * (B - A) + J(2) * (C - A)$$
$$P3 = A + I(3) * (B - A) + J(3) * (C - A)$$

P0	P1
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.

Basis for 8-deep Latch Memory (from 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 μ^2 per converter

Method 1

Block	Quantity	Area
F2F	3	14136
8x96 Latch	16	744384
		758520 μ^2

Figure 8: Area Estimate for VGT to Shader Interface

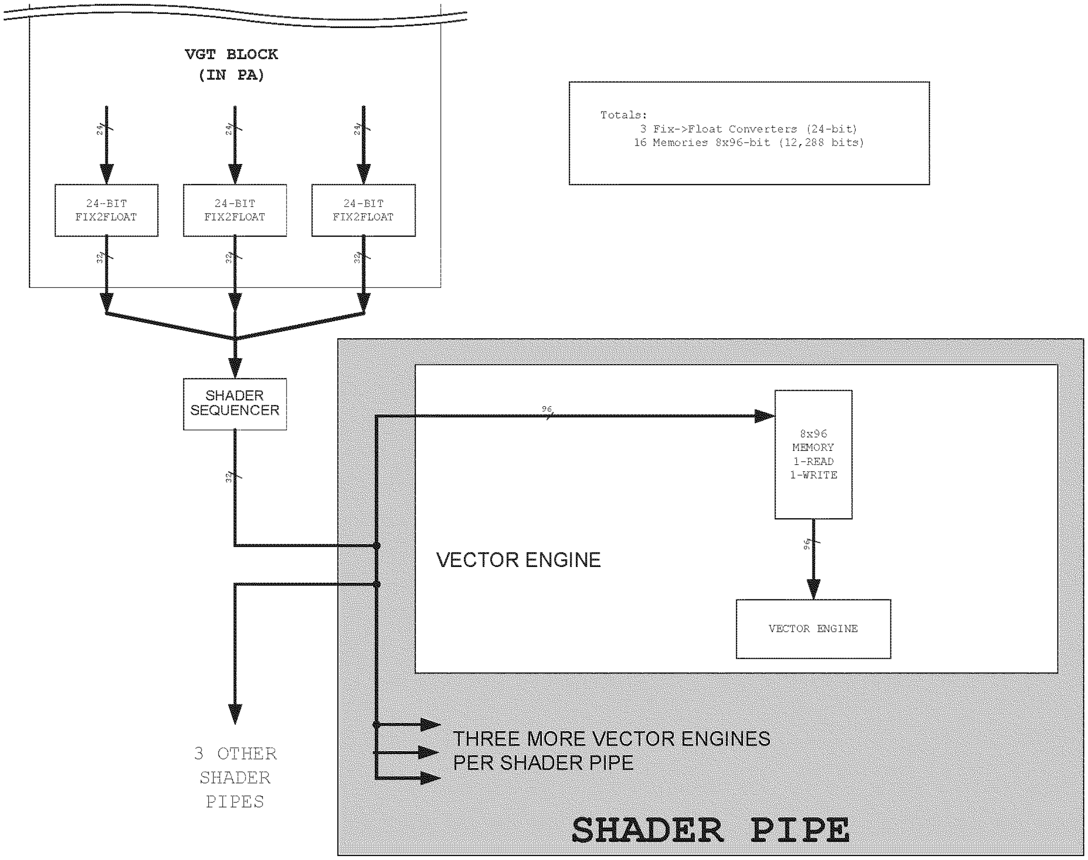


Figure 9: VGT to Shader Interface

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 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 00100000000 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:
.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 EXaddress:

.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 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)
6) 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

(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

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

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 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:

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:

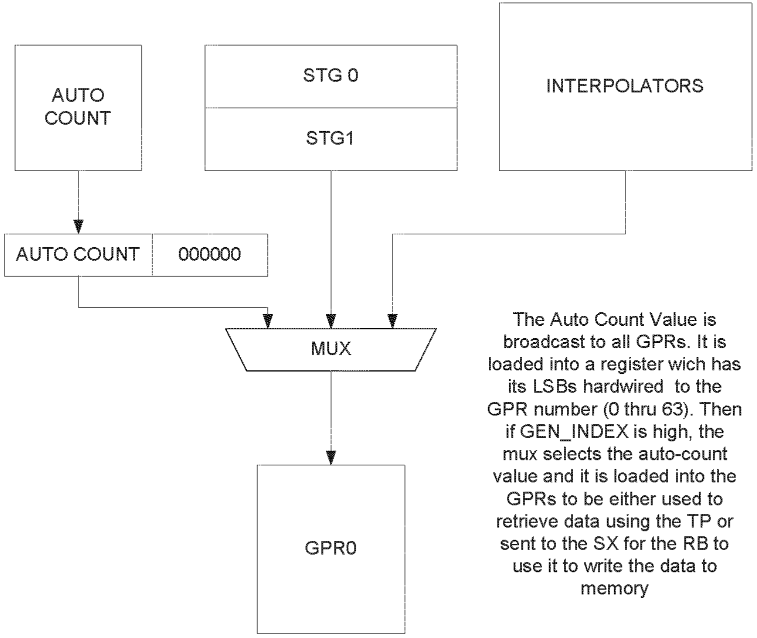
Counter (21 bits)	Hardwired (4 bits)

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.



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 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→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.

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→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
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

		interpolation 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
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, 2nd clk could be all zeroes

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

Name	BitField	Bits	Description
1st 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

			=> TBD
SC_SQ_state_id	[8:6]	3	State/constant pointer (6*3+3)
SC_SQ_pc_dealloc	[11:9]	3	Deallocation token for the Parameter Cache
SC_SQ_new_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_mask	[16:13]	4	Quad Write mask left to right SP0 => SP3
SC_SQ_end_of_prim	17	1	End Of the primitive
SC_SQ_pix_mask	[33:18]	16	Valid bits for all pixels SP0=>SP3 (UL,UR,LL,LR)
SC_SQ_provok_vtx	[35:34]	2	Provoking vertex for flat shading
SC_SQ_lod_correct_0	[44:36]	9	LOD correction for quad 0 (SP0) (9 bits per quad)
SC_SQ_lod_correct_1	[53:45]	9	LOD correction for quad 1 (SP1) (9 bits per quad)
2nd Clock Transfer			
SC_SQ_lod_correct_2	[8:0]	9	LOD correction for quad 2 (SP2) (9 bits per quad)
SC_SQ_lod_correct_3	[17:9]	9	LOD correction for quad 3 (SP3) (9 bits per quad)
SC_SQ_pc_ptr0	[28:18]	11	Parameter Cache pointer for vertex 0
SC_SQ_pc_ptr1	[39:29]	11	Parameter Cache pointer for vertex 1
SC_SQ_pc_ptr2	[50:40]	11	Parameter Cache pointer for vertex 2
SC_SQ_prim_type	[53:51]	3	Stippled line and Real time command need to load tex cords from alternate buffer 000: Sprite (point) 001: Line 010: Tri_rect 100: Realtime Sprite (point) 101: Realtime Line 110: Realtime Tri_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.)

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

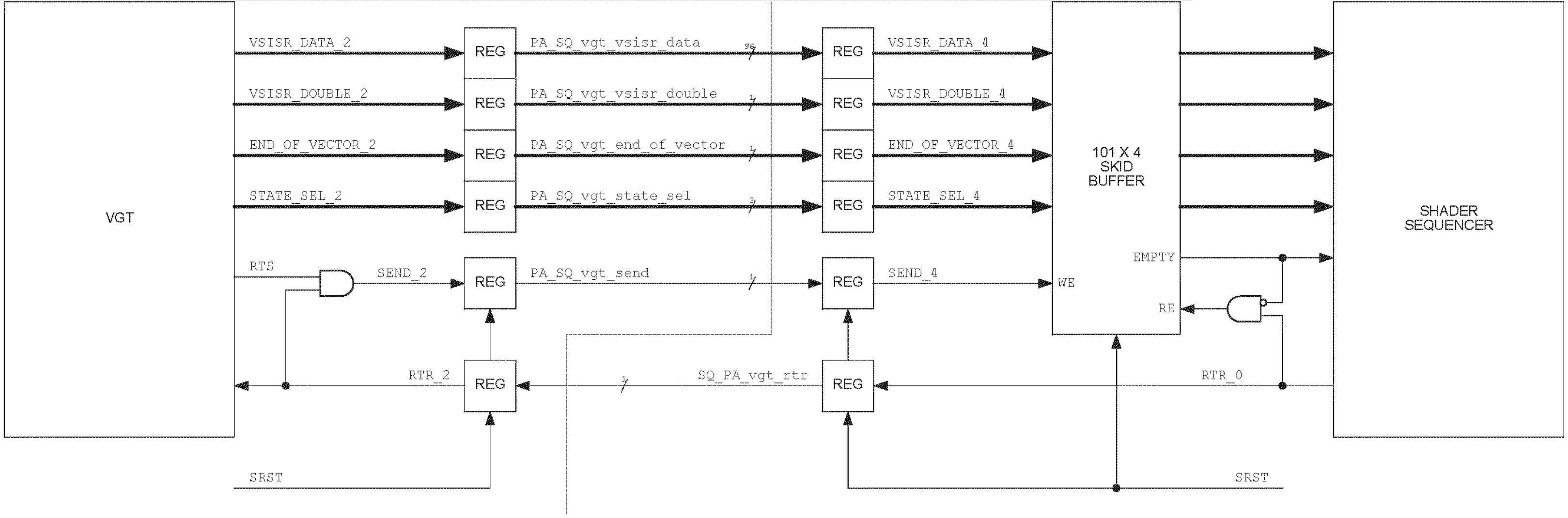
23.2.5 *VGT to SQ : Vertex interface*

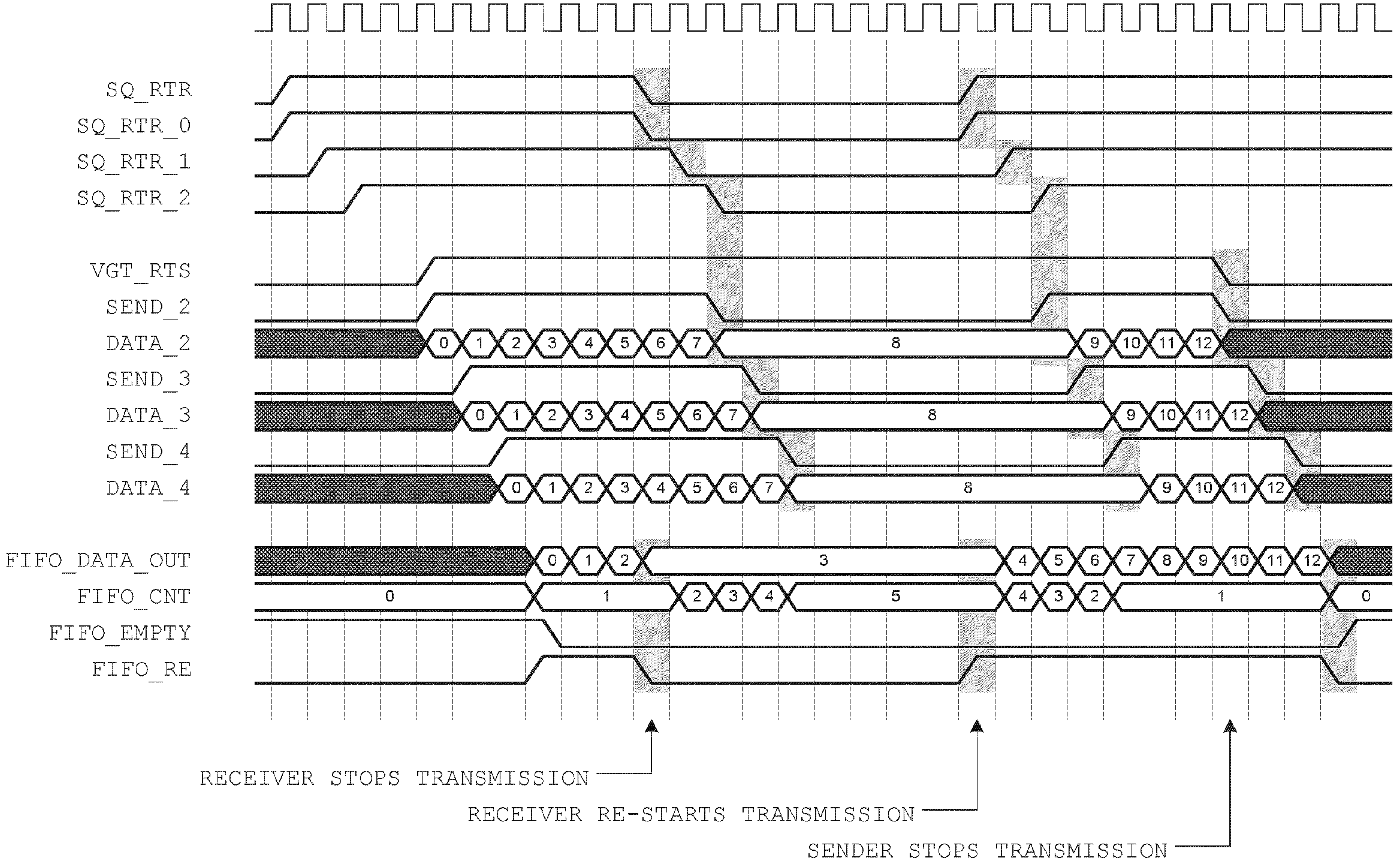
23.2.5.1 Interface Signal Table

The area difference between the two methods is not sufficient to warrant complicating the interface or the state requirements of the VSISRs. **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.** 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


VGT
VSISR_DATA_2
VSISR_DOUBLE_2
END_OF_VECTOR_2
STATE_SEL_2
RTS
SEND_2
RTR_2
SRST
REG
PA_SQ_vgt_vsisr_data
96
PA_SQ_vgt_vsisr_double
1
PA_SQ_vgt_end_of_vector
1
PA_SQ_vgt_state_sel
3
PA_SQ_vgt_send
1
SQ_PA_vgt_rtr
1
VSISR_DATA_4
VSISR_DOUBLE_4
END_OF_VECTOR_4
STATE_SEL_4
SEND_4
101 X 4
SKID
BUFFER
WE
EMPTY
RE
RTR_0
SRST
SHADER
SEQUENCER



RECEIVER STOPS TRANSMISSION

RECEIVER RE-STARTS TRANSMISSION

SENDER STOPS TRANSMISSION

Figure 1. Detailed Logical Diagram for PA SQ vgt Interface.

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
  - 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)
  - 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 = 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**.

23.2.7 *SX to SQ : Output file control*

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

SXx_SQ_pos_count1_valid	SXx→SQ	1	Free_count1 is valid
SXx_SQ_mem_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→ SQ	1	Data ready
TPx_SQ_rs_line_num	TPx→ 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→ SQ	1	Data ready
VCx_SQ_rs_line_num	VCx→ 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

SQ_VCx_write_gpr_index	SQ->VCx	7	Index into Register file for write of returned Fetch Data
SQ_VCx_SIMD	SQ->VCx	1	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→ 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→SPx	6	Swizzle code for the TP request (2 bits per channel ignore W as it is not used). Bits [1..0] X mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W Bits [3..2] Y mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W Bits [5..4] Z mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W
SQ_SPx_simd0_fetch_resource	SQ→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

SQ_SP1_simd1_gpr_pspv_wr_en	SQ→SP1 (SP5)	4	Write Enable for the GPRs of SP1 for PS and PV
SQ_SP2_simd1_gpr_pspv_wr_en	SQ→SP2 (SP6)	4	Write Enable for the GPRs of SP2 for PS and PV
SQ_SP3_simd1_gpr_pspv_wr_en	SQ→SP3 (SP7)	4	Write Enable for the GPRs of SP3 for PS and PV
SQ_SPx__simd1_channel_mask	SQ→SPx	4	The channel mask for SIMD1
SQ_SPx_simd1_fetch_resource	SQ→SPx	1	Resource in use currently 0: TP 1: VC
SQ_SPx_simd1_fetch_swizzle	SQ→SPx	6	Swizzle code for the TP request (2 bits per channel ignore W as it is not used). Bits [1..0] X mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W Bits [3..2] Y mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W Bits [5..4] Z mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W
SQ_SP0_simd1_gpr_int_wr_en	SQ→SP0	1	Write Enable for the GPRs of SP0 for Inputs (interp/vtx)
SQ_SP1_simd1_gpr_int_wr_en	SQ→SP1	1	Write Enable for the GPRs of SP1 for Inputs (interp/vtx)
SQ_SP2_simd1_gpr_int_wr_en	SQ→SP2	1	Write Enable for the GPRs of SP2 for Inputs (interp/vtx)
SQ_SP3_simd1_gpr_int_wr_en	SQ→SP3	1	Write Enable for the GPRs of SP3 for Inputs (interp/vtx)

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 all 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 all 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 all 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 channel 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 channel 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 logic).

			1: Use GPR for PV or PS settings. LET the 11 (constant) go thru unchanged
SQ_SPx_simd0_stall	SQ→SPx	1	Stall signal

SQ_SPx_simd0_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_simd1_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 all 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 all 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 all 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_simd1_pred_override	SQ→SP0 (SP4)	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_SP1_simd1_pred_override	SQ→SP1 (SP5)	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_SP2_simd1_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_simd1_pred_override	SQ→SP3 (SP7)	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_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.

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

SQ_SPx_simd0_const	SQ→SPx	128	Constant broadcast
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

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	TYPE	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:

```
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.

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:

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 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 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 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?

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	9 July, 2003	4 September, 2015	GEN-CXXXXX-REVA	1 of 54

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SQ

Version 2.11

Overview: 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 sub-blocks, and provides internal state diagrams.

AUTOMATICALLY UPDATED FIELDS:
Document Location: C:\perforce\r400\doc_lib\design\blocks\sq\R400_Sequencer.doc
Current Intranet Search Title: R400 Sequencer Specification

APPROVALS	
Name/Dept	Signature/Date

Remarks:

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

ATI 2039
LG v. ATI
IPR2015-00325

Table Of Contents

Revision Changes:

Rev 0.1 (Laurent Lefebvre) Date: May 7, 2001	First draft.
Rev 0.2 (Laurent Lefebvre) Date : July 9, 2001	Changed the interfaces to reflect the changes in the SP. Added some details in the arbitration section.
Rev 0.3 (Laurent Lefebvre) Date : August 6, 2001	Reviewed the Sequencer spec after the meeting on August 3, 2001.
Rev 0.4 (Laurent Lefebvre) Date : August 24, 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.
Rev 0.5 (Laurent Lefebvre) Date : September 7, 2001	Added timing diagrams (Vic)
Rev 0.6 (Laurent Lefebvre) Date : September 24, 2001	Changed the spec to reflect the new R400 architecture. Added interfaces.
Rev 0.7 (Laurent Lefebvre) Date : October 5, 2001	Added constant store management, instruction store management, control flow management and data dependant predication.
Rev 0.8 (Laurent Lefebvre) Date : October 8, 2001	Changed the control flow method to be more flexible. Also updated the external interfaces.
Rev 0.9 (Laurent Lefebvre) Date : October 17, 2001	Incorporated changes made in the 10/18/01 control flow meeting. Added a NOP instruction, removed the conditional_execute_or_jump. Added debug registers.
Rev 1.0 (Laurent Lefebvre) Date : October 19, 2001	Refined interfaces to RB. Added state registers.
Rev 1.1 (Laurent Lefebvre) Date : October 26, 2001	Added SEQ→SP0 interfaces. Changed delta precision. Changed VGT→SP0 interface. Debug Methods added.
Rev 1.2 (Laurent Lefebvre) Date : November 16, 2001	Interfaces greatly refined. Cleaned up the spec.
Rev 1.3 (Laurent Lefebvre) Date : November 26, 2001	Added the different interpolation modes.
Rev 1.4 (Laurent Lefebvre) Date : December 6, 2001	Added the auto incrementing counters. Changed the VGT→SQ interface. Added content on constant management. Updated GPRs.
Rev 1.5 (Laurent Lefebvre) Date : December 11, 2001	Removed from the spec all interfaces that weren't directly tied to the SQ. Added explanations on constant management. Added PA→SQ synchronization fields and explanation.
Rev 1.6 (Laurent Lefebvre) Date : January 7, 2002	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.
Rev 1.7 (Laurent Lefebvre) Date : February 4, 2002	Added Real Time parameter control in the SX interface. Updated the control flow section.
Rev 1.8 (Laurent Lefebvre) Date : March 4, 2002	New interfaces to the SX block. Added the end of clause modifier, removed the end of clause instructions.
Rev 1.9 (Laurent Lefebvre) Date : March 18, 2002	Rearangement of the CF instruction bits in order to ensure byte alignement.
Rev 1.10 (Laurent Lefebvre) Date : March 25, 2002	Updated the interfaces and added a section on exporting rules.
Rev 1.11 (Laurent Lefebvre) Date : April 19, 2002	Added CP state report interface. Last version of the spec with the old control flow scheme
Rev 2.0 (Laurent Lefebvre) Date : April 19, 2002	New control flow scheme

Rev 2.01 (Laurent Lefebvre) Date : May 2, 2002	Changed slightly the control flow instructions to allow force jumps and calls.
Rev 2.02 (Laurent Lefebvre) Date : May 13, 2002	Updated the Opcodes. Added type field to the constant/pred interface. Added Last field to the SQ→SP instruction load interface.
Rev 2.03 (Laurent Lefebvre) Date : July 15, 2002	SP interface updated to include predication optimizations. Added the predicate no stall instructions,
Rev 2.04 (Laurent Lefebvre) Date :August 2, 2002	Documented the new parameter generation scheme for XY coordinates points and lines STs.
Rev 2.05 (Laurent Lefebvre) Date : September 10, 2002	Some interface changes and an architectural change to the auto-counter scheme.
Rev 2.06 (Laurent Lefebvre) Date : October 11, 2002	Widened the event interface to 5 bits. Some other little typos corrected.
Rev 2.07 (Laurent Lefebvre) Date : October 14, 2002	Loops, jumps and calls are now using a 13 bit address which allows to jump and call and loop around any control flow addresses (does not requires to be even anymore).
Rev 2.08 (Laurent Lefebvre) Date : October 16, 2002	Clarification updates after discussion with Clay.
Rev 2.09 (Laurent Lefebvre) Date : January 7, 2003	Corrected the SQ→SP staging register interface.
Rev 2.10 (Laurent Lefebvre) Date : April 8, 2003	Adding R500 modifications
Rev 2.11 (Laurent Lefebvre) Date : May 1, 2003	Adding SQ->SP updated interfaces

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.

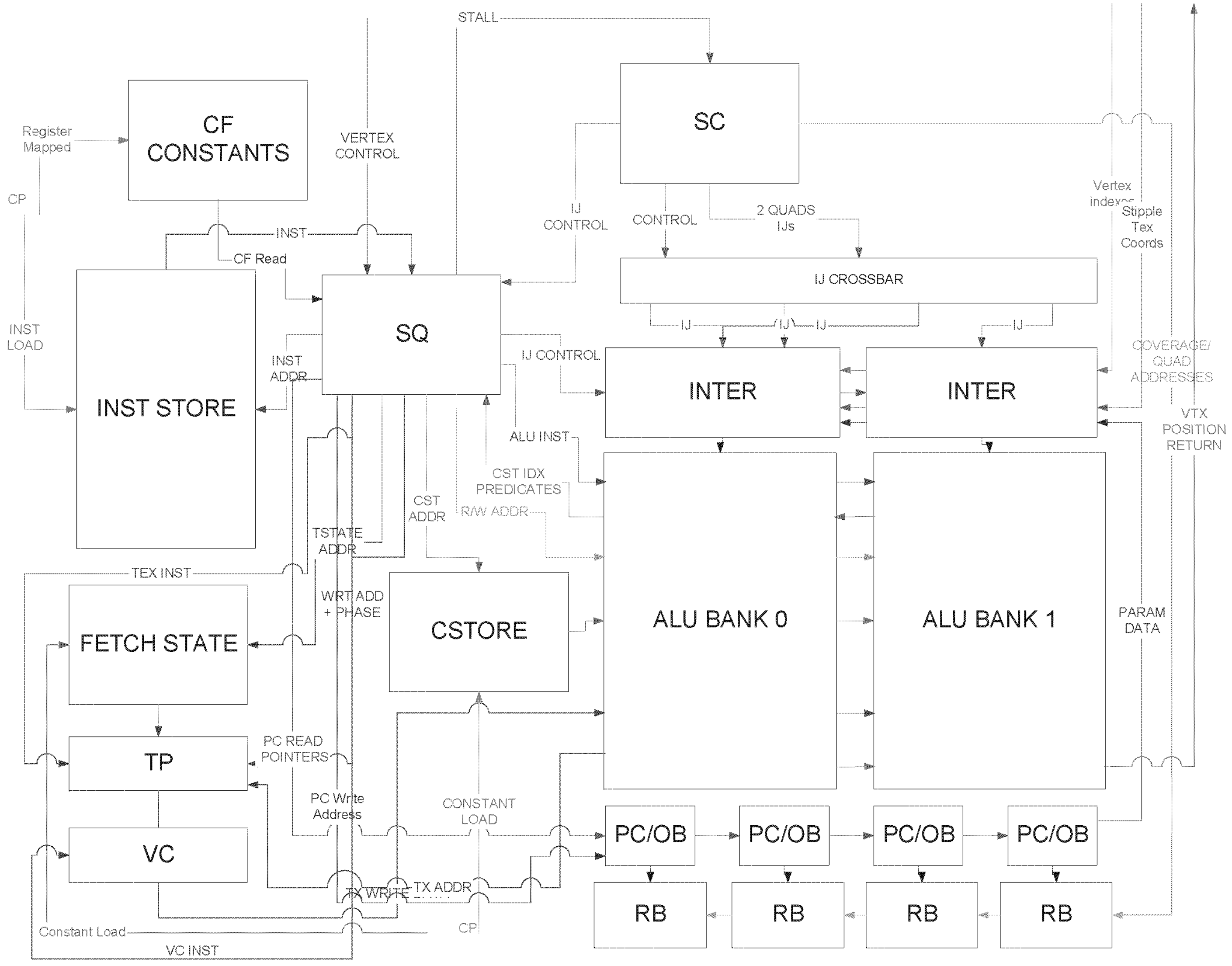


Figure 1: General Sequencer overview

1.1 Top Level Block Diagram

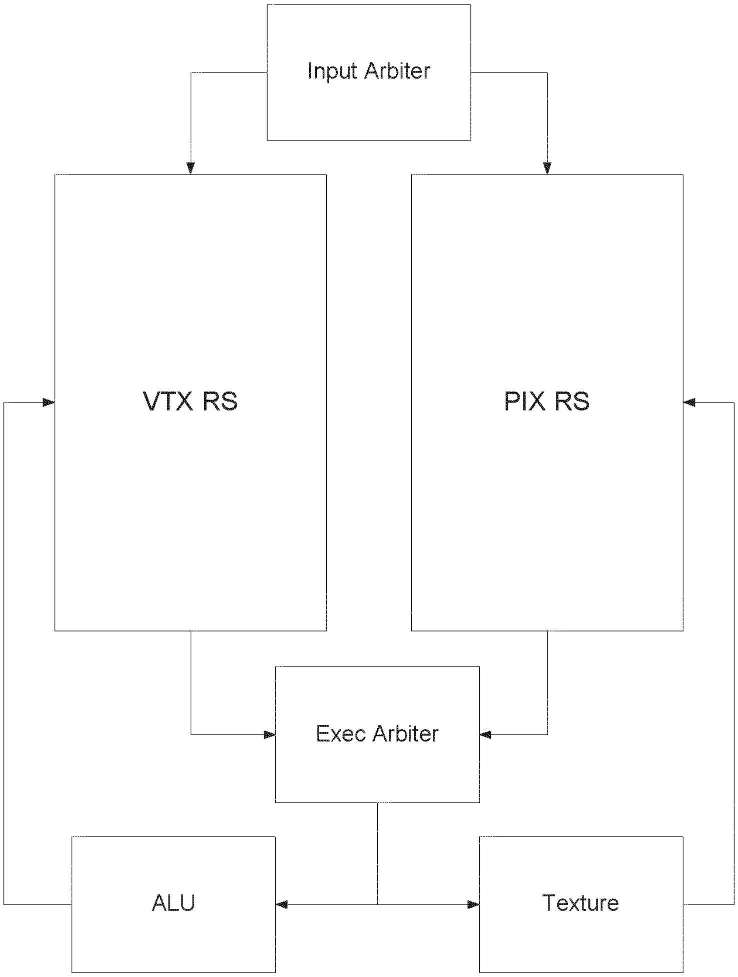
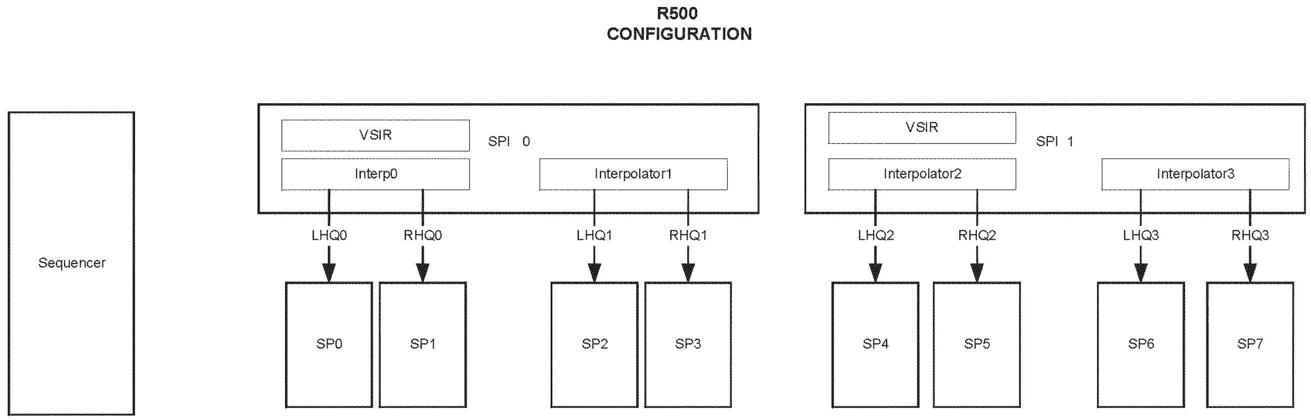


Figure 2: Reservation stations and arbiters

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).

1.2 Data Flow graph (SP)



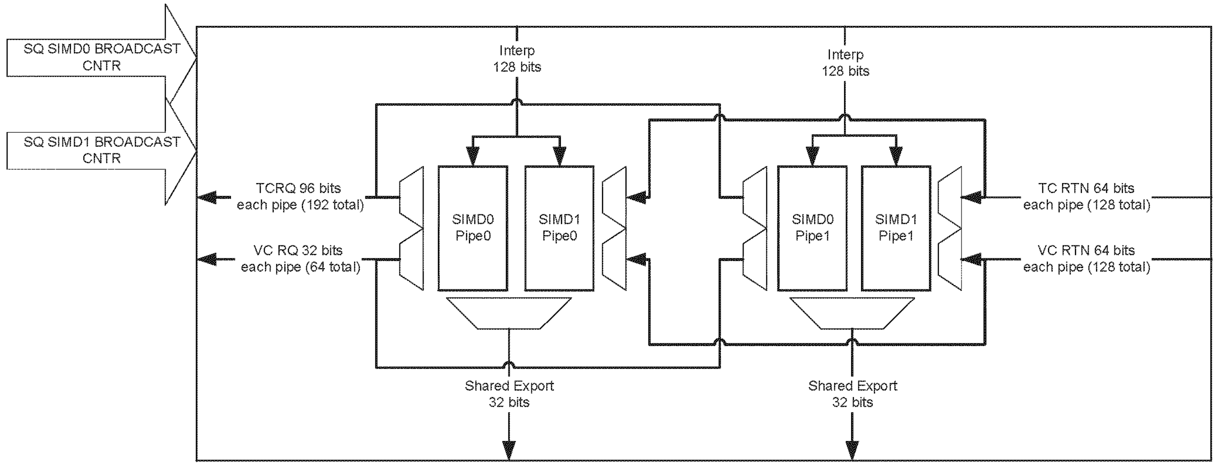


Figure 3: The shader Pipe

1.3 Control Graph

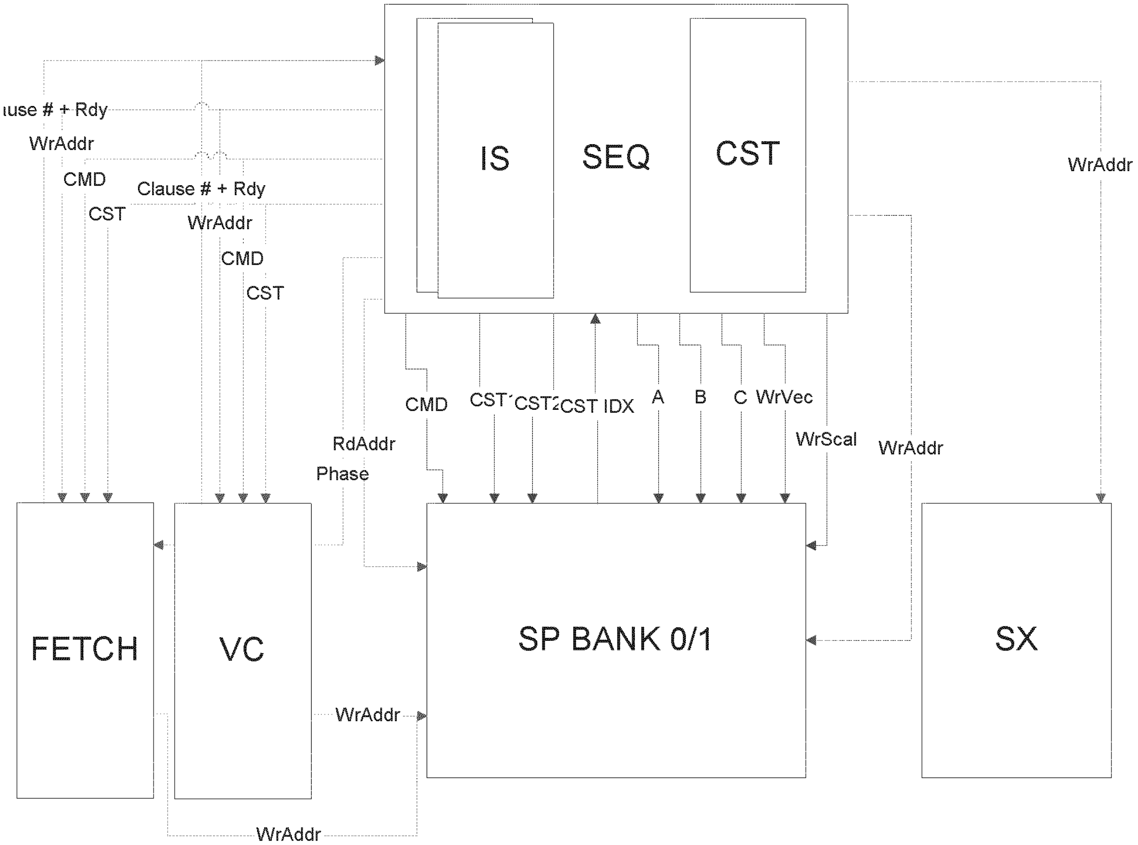


Figure 4: Sequencer Control interfaces

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.

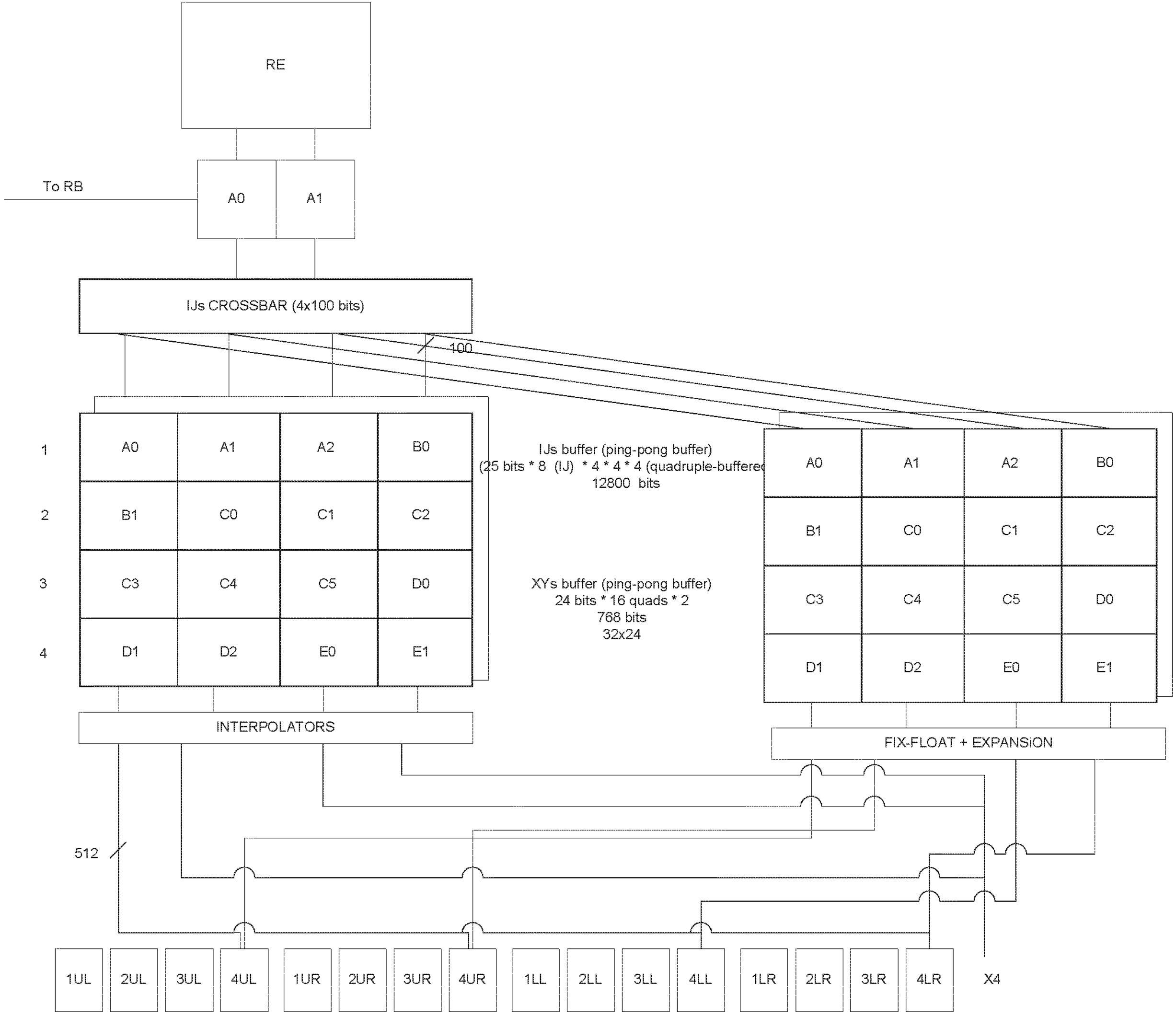


Figure 5: Interpolation buffers

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19	T20	T21	T22	T23
												WRITES												
SP 0	A0	A0	XY A0	B1	B1	XY B1	C3	C3	XY C3				D1	D1	XY D1									
SP 1	A1	A1	XY A1				C0	C0	XY C0	C4	C4	XY C4	D2	D2	XY D2									
SP 2	A2	A2	XY A2				C1	C1	XY C1	C5	C5	XY C5				E0	E0	XY E0						
SP 3				B0	B0	XY B0	C2	C2	XY C2				D0	D0	XY D0	E1	E1	XY E1						
												READS												
SP 0	XY 0-3	XY 16-19	XY 32-35	XY 48-51	A0	B1	C3	D1					A0	B1	C3	D1					V 0-3	V 16-19	V 32-35	V 48-51
SP 1	XY 4-7	XY 20-23	XY 36-39	XY 52-55	A1		C4	D2		C0			A1		C4	D2		C0			V 4-7	V 20-23	V 36-39	V 52-55
SP 2	XY 8-11	XY 24-27	XY 40-43	XY 56-59	A2		C5			C1		E0	A2		C5			C1		E0	V 8-11	V 24-27	V 40-43	V 56-59
SP 3	XY 12-15	XY 28-31	XY 44-47	XY 60-63					B0	C2	D0	E1					B0	C2	D0	E1	V 12-15	V 28-31	V 44-47	V 60-63
			XY					P1								P2						VTX		

Figure 6: Interpolation timing diagram

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 SIMD0
ALU 1 SIMD0 CF
ALU 1 SIMD0
Fetch CF
Fetch
VC CF
VC

ALU 0 SIMD1 CF
ALU 0 SIMD1
ALU 1 SIMD1 CF
ALU 1 SIMD1
Fetch CF
Fetch
VC CF
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 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

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 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).

```
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.

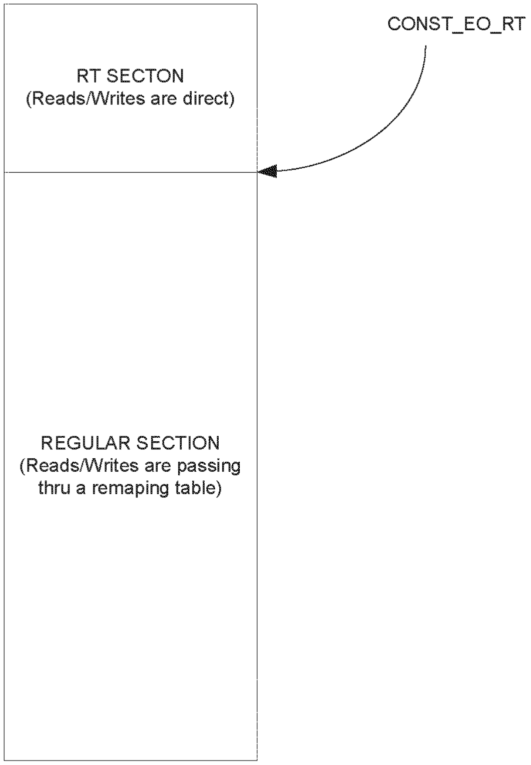

CONST_EO_RT
RT SECTON
(Reads/Writes are direct)
REGULAR SECTION
(Reads/Writes are passing thru a remaping table)

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
```

```
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	27 …16	15…12	11 … 0
0001	Addressing	RESERVED	Vertex Cache	Instructions type + serialize (6 instructions)	Count	Exec Address

Execute_End						
47 … 44	43	40 … 34	33 … 28	27 …16	15…12	11 … 0
0010	Addressing	RESERVED	Vertex Cache	Instructions type + serialize (6 instructions)	Count	Exec Address

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	33…28	27…16	15 …12	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	33…28	27…16	15 …12	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	33…28	27…16	15…12	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	33…28	27…16	15…12	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.

Conditional_Execute_Predicates_No_Stall								
47 … 44	43	42	41 … 36	35 … 34	33…28	27…16	15…12	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	33…28	27…16	15…12	11 … 0
1110	Addressing	Condition	RESERVED	Predicate vector	Vertex Cache	Instructions type + serialize (6 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	15…13	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	35…34	33… 22	21	20 … 16	15…13	12 … 0
1000	Addressing	Cond	RESERVED	Predicate Vector	RESERVED	Pred 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 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

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 address	FW only	RESERVED	Force Jump	Jump 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	42…41	40	39 … 3	2…0
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 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.

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 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.

6.5 HW Detection of PV,PS

Because of the control program, the compiler cannot detect statically dependant instructions. In the case of non-masked 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.

{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).

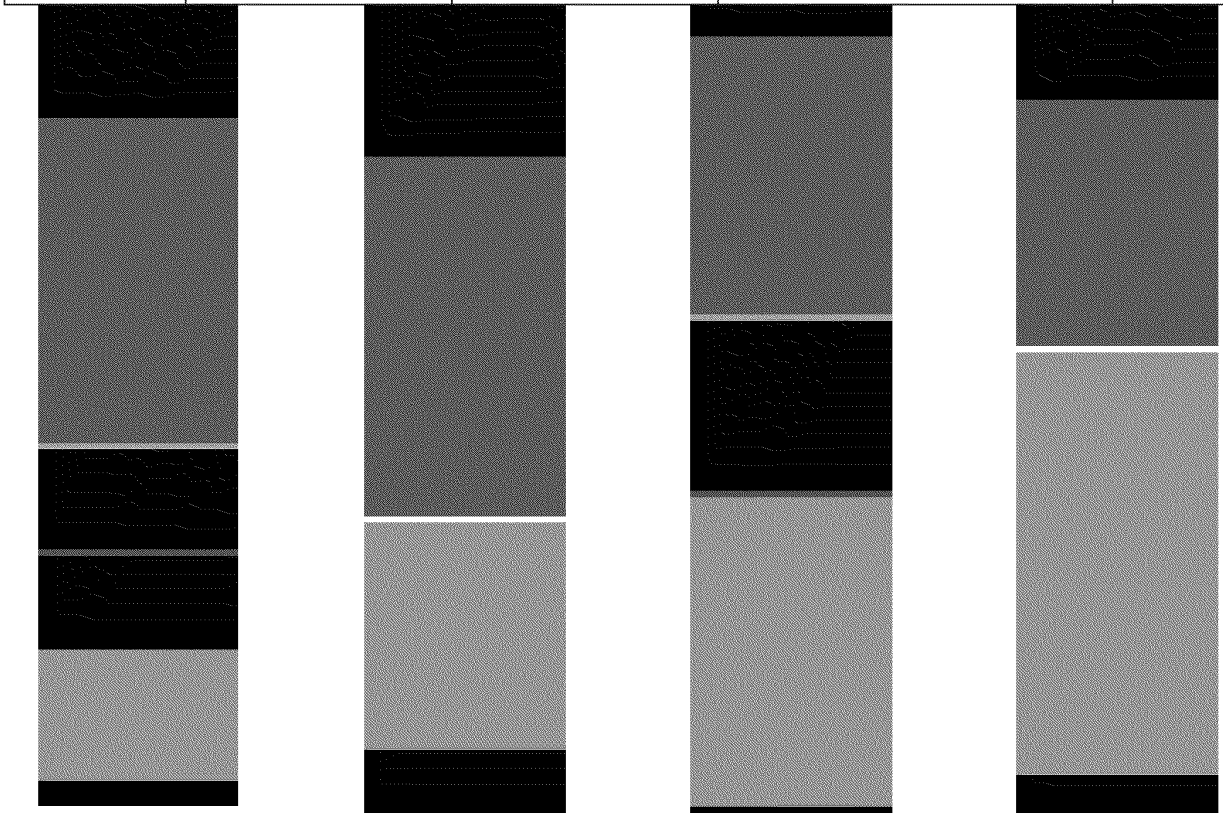
7. Pixel Kill Mask

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.



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.

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 arbiters, 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.

$$P0 = A + I(0) * (B - A) + J(0) * (C - A)$$
$$P1 = A + I(1) * (B - A) + J(1) * (C - A)$$
$$P2 = A + I(2) * (B - A) + J(2) * (C - A)$$
$$P3 = A + I(3) * (B - A) + J(3) * (C - A)$$

P0	P1
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:

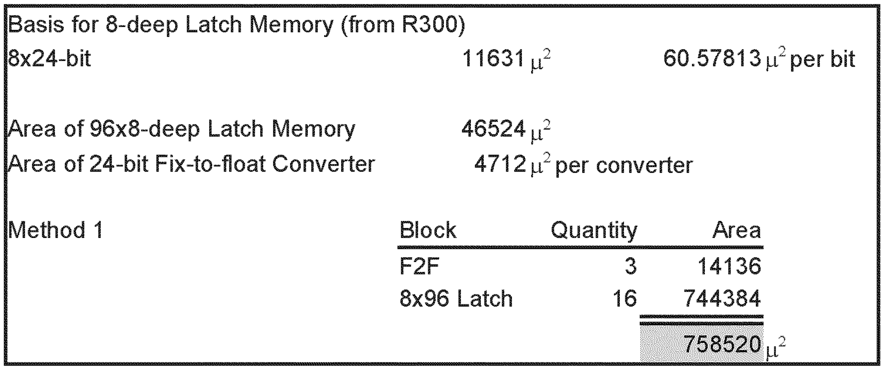
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 || 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 || 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 || 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

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.



Basis for 8-deep Latch Memory (from 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 μ^2 per converter	

Method 1

Block	Quantity	Area
F2F	3	14136
8x96 Latch	16	744384
		758520 μ^2

Figure 8:Area Estimate for VGT to Shader Interface

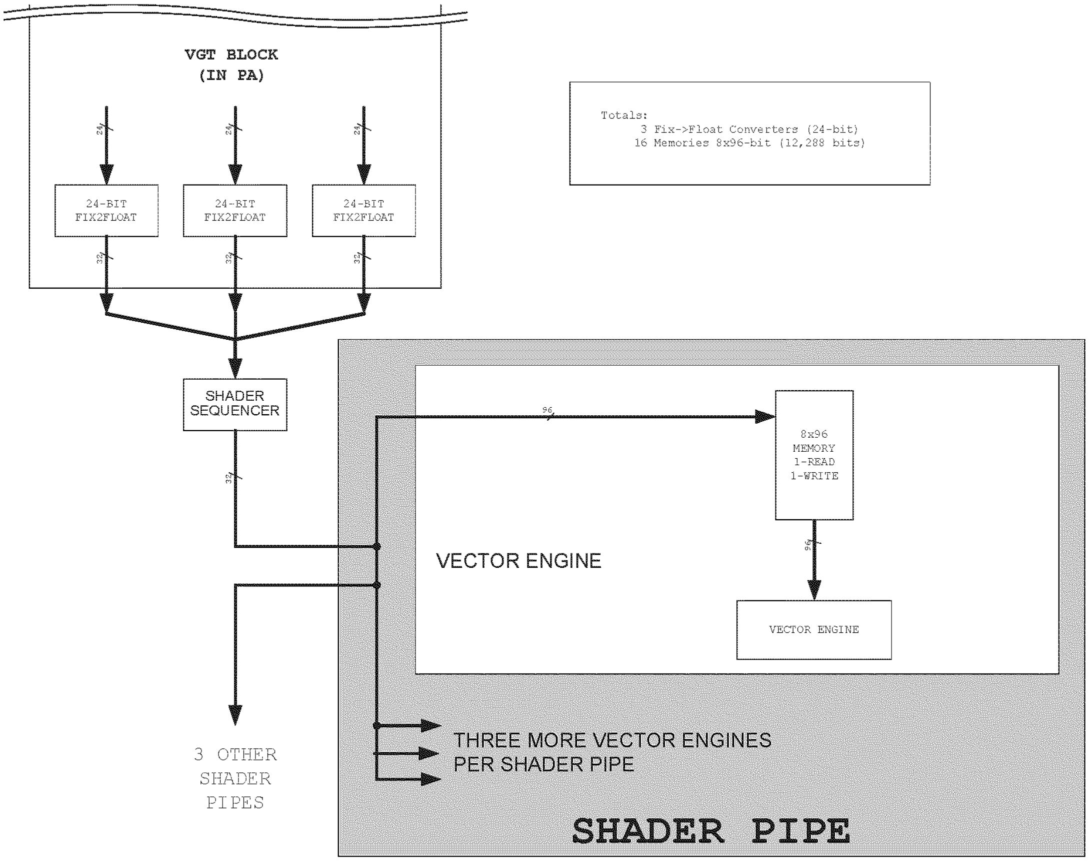


Figure 9:VGT to Shader Interface

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 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 00100000000 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:
.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 EXaddress:

.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 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)
6) 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

(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

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

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 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 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:

Counter (17 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:

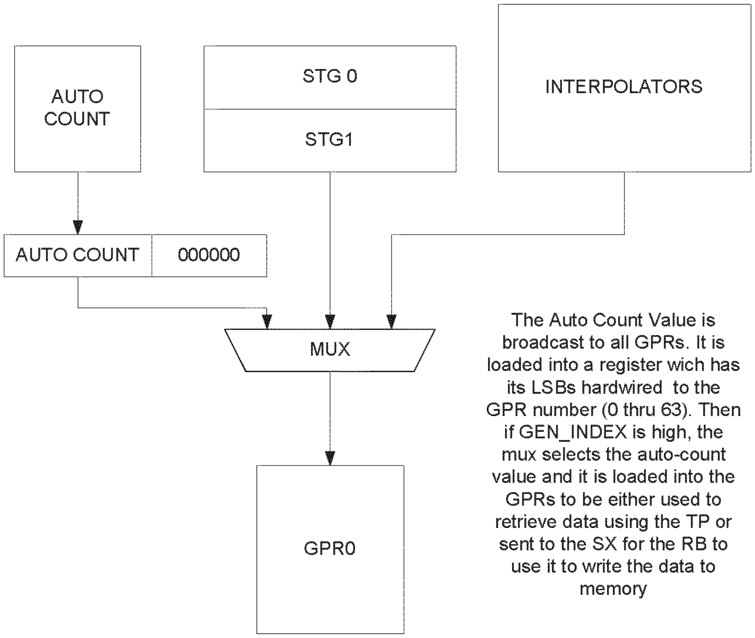
Counter (19 bits)	Hardwired (4 bits)

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).

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.



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 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→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.

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→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
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 interpolation 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
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, 2nd clk could be all zeroes

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

Name	BitField	Bits	Description

1st 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 => TBD
SC_SQ_state_id	[8:6]	3	State/constant pointer (6*3+3)
SC_SQ_pc_dealloc	[11:9]	3	Deallocation token for the Parameter Cache
SC_SQ_new_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_mask	[16:13]	4	Quad Write mask left to right SP0 => SP3
SC_SQ_end_of_prim	17	1	End Of the primitive
SC_SQ_pix_mask	[33:18]	16	Valid bits for all pixels SP0=>SP3 (UL,UR,LL,LR)
SC_SQ_provok_vtx	[35:34]	2	Provoking vertex for flat shading
SC_SQ_lod_correct_0	[44:36]	9	LOD correction for quad 0 (SP0) (9 bits per quad)
SC_SQ_lod_correct_1	[53:45]	9	LOD correction for quad 1 (SP1) (9 bits per quad)
2nd Clock Transfer			
SC_SQ_lod_correct_2	[8:0]	9	LOD correction for quad 2 (SP2) (9 bits per quad)
SC_SQ_lod_correct_3	[17:9]	9	LOD correction for quad 3 (SP3) (9 bits per quad)
SC_SQ_pc_ptr0	[28:18]	11	Parameter Cache pointer for vertex 0
SC_SQ_pc_ptr1	[39:29]	11	Parameter Cache pointer for vertex 1
SC_SQ_pc_ptr2	[50:40]	11	Parameter Cache pointer for vertex 2
SC_SQ_prim_type	[53:51]	3	Stippled line and Real time command need to load tex cords from alternate buffer 000: Sprite (point) 001: Line 010: Tri_rect 100: Realtime Sprite (point) 101: Realtime Line 110: Realtime Tri_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.)

23.2.3 SQ to SX(SP): Interpolator bus

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

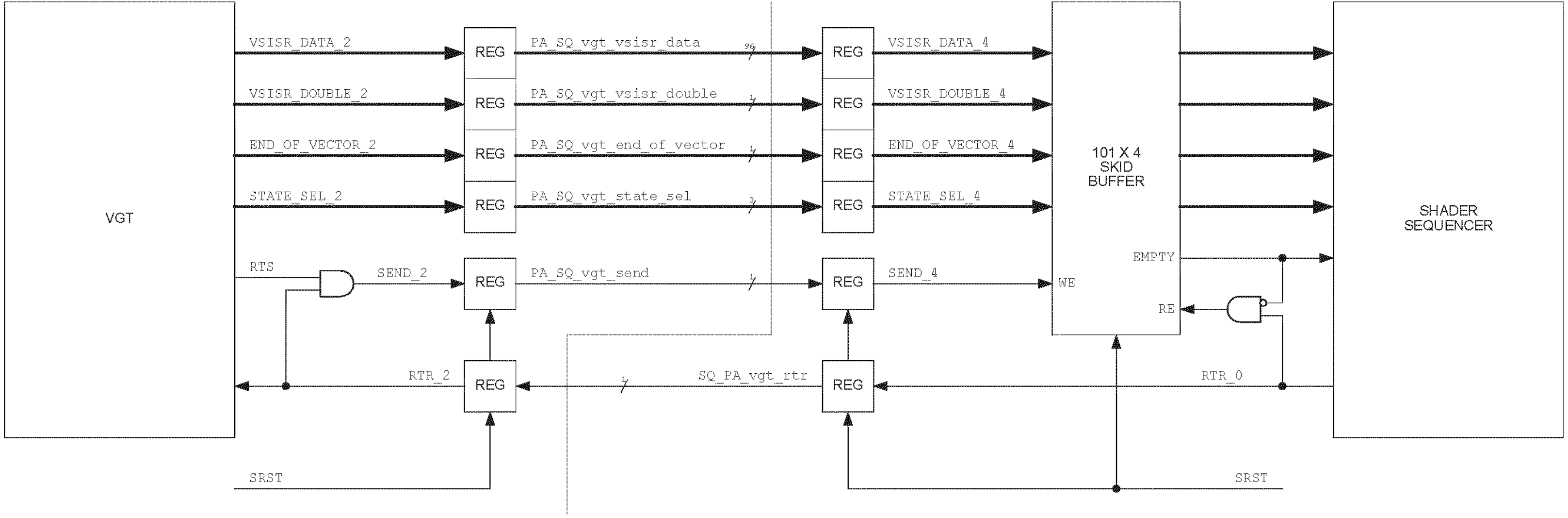
23.2.5 VGT to SQ : Vertex interface

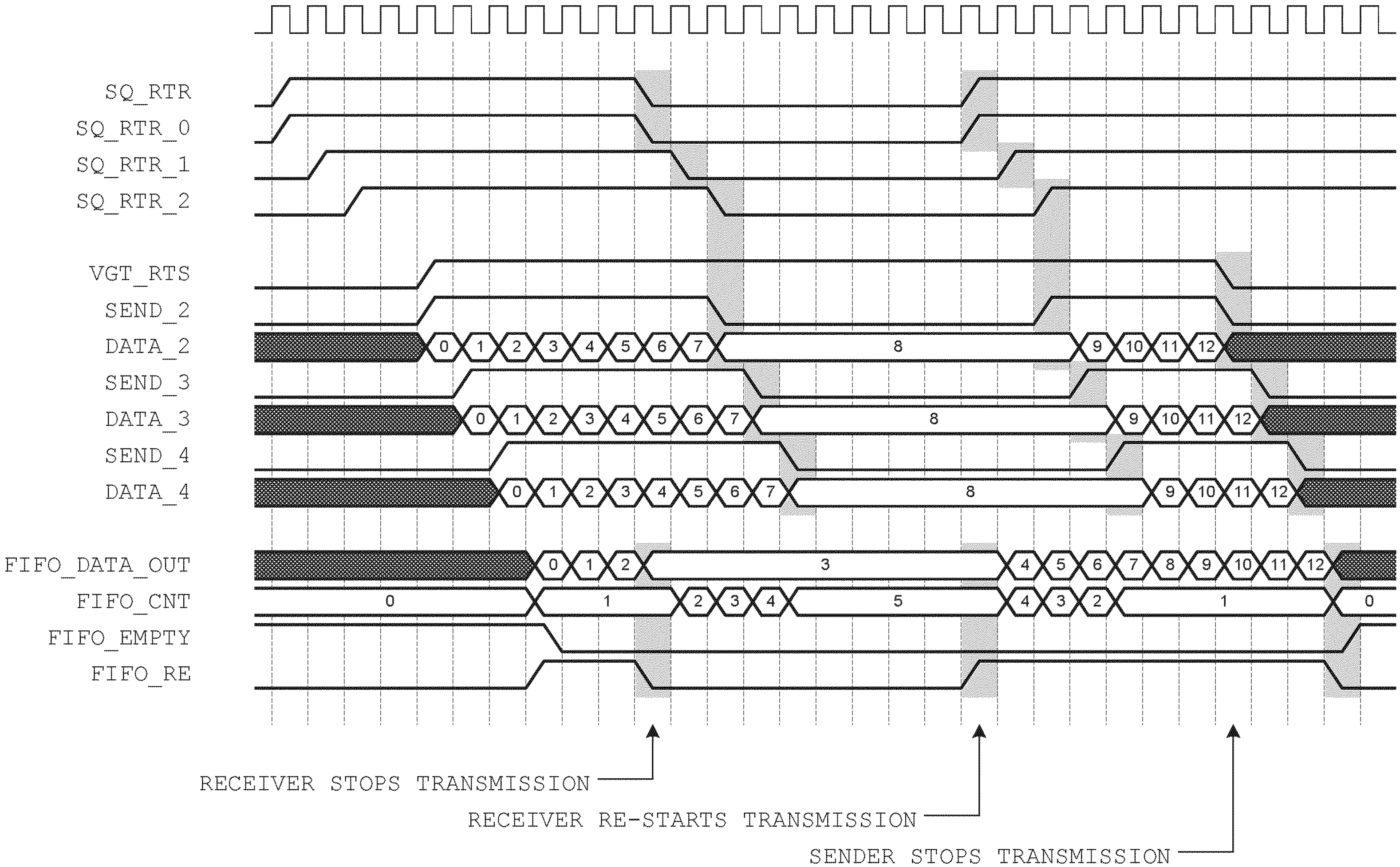
23.2.5.1 Interface Signal Table

The area difference between the two methods is not sufficient to warrant complicating the interface or the state requirements of the VSISRs. **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.** 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


VGT
VSISR_DATA_2
VSISR_DOUBLE_2
END_OF_VECTOR_2
STATE_SEL_2
RTS
SEND_2
RTR_2
SRST
REG
PA_SQ_vgt_vsisr_data
PA_SQ_vgt_vsisr_double
PA_SQ_vgt_end_of_vector
PA_SQ_vgt_state_sel
PA_SQ_vgt_send
SQ_PA_vgt_rtr
96
1
1
3
1
1
VSISR_DATA_4
VSISR_DOUBLE_4
END_OF_VECTOR_4
STATE_SEL_4
SEND_4
101 X 4
SKID
BUFFER
WE
EMPTY
RE
RTR_0
SRST
SHADER
SEQUENCER



RECEIVER STOPS TRANSMISSION

RECEIVER RE-STARTS TRANSMISSION

SENDER STOPS TRANSMISSION

Figure 1. Detailed Logical Diagram for PA_SQ_vgt Interface.

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
  - 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)
  - 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 = 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**.

23.2.7 SX to SQ : Output file control

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

SXx_SQ_pos_count1_valid	SXx→SQ	1	Free_count1 is valid
SXx_SQ_mem_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→ SQ	1	Data ready
TPx_SQ_rs_line_num	TPx→ 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→ SQ	1	Data ready
VCx_SQ_rs_line_num	VCx→ 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

SQ_VCx_write_gpr_index	SQ->VCx	7	Index into Register file for write of returned Fetch Data
SQ_VCx_SIMD	SQ->VCx	1	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 frees up a buffer.

Name	Direction	Bits	Description
TP_SQ_fetch_dec	TP→ 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→ 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*

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→SPx	6	Swizzle code for the TP request (2 bits per channel ignore W as it is not used). Bits [1..0] X mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W Bits [3..2] Y mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W Bits [5..4] 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

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 (SP1)	4	Write Enable for the GPRs of SP0-1 for PS and PV
SQ_SP2_simd1_gpr_pspv_wr_en	SQ→SP2 (SP3)	4	Write Enable for the GPRs of SP2-3 for PS and PV
SQ_SP4_simd1_gpr_pspv_wr_en	SQ→SP4 (SP5)	4	Write Enable for the GPRs of SP4-5 for PS and PV
SQ_SP6_simd1_gpr_pspv_wr_en	SQ→SP6 (SP7)	4	Write Enable for the GPRs of SP6-7 for PS and PV
SQ_SPx__simd1_channel_mask	SQ→SPx	4	The channel mask for SIMD1
SQ_SPx_simd1_fetch_swizzle	SQ→SPx	6	Swizzle code for the TP request (2 bits per channel ignore W as it is not used). Bits [1..0] X mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W Bits [3..2] Y mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W Bits [5..4] Z mode select: 0=GPR_X 1=GPR_Y 2=GPR_Z 3=GPR_W
SQ_SP0_simd1_gpr_int_wr_en	SQ→SP0	1	Write Enable for the GPRs of SP0-1 for Inputs (interp/vtx)
SQ_SP2_simd1_gpr_int_wr_en	SQ→SP2	1	Write Enable for the GPRs of SP2-3 for Inputs (interp/vtx)
SQ_SP4_simd1_gpr_int_wr_en	SQ→SP4	1	Write Enable for the GPRs of SP4-5 for Inputs (interp/vtx)
SQ_SP6_simd1_gpr_int_wr_en	SQ→SP6	1	Write Enable for the GPRs of SP6-7 for Inputs (interp/vtx)

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 all 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 all 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 all 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 (SP1)	4	SP0 receives 2 bits and SP1 two bits as well. 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_SP2_simd0_pred_override	SQ→SP2 (SP3)	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_SP4_simd0_pred_override	SQ→SP4 (SP5)	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_SP6_simd0_pred_override	SQ→SP6 (SP7)	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_SPx_simd0_stall	SQ→SPx	1	Stall signal
SQ_SPx_simd1_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 all 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 all 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 all 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 (SP1)	4	SP0 receives 2 bits and SP1 two bits as well. 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_SP2_simd0_pred_override	SQ→SP2 (SP3)	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_SP4_simd0_pred_override	SQ→SP4 (SP5)	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_SP6_simd0_pred_override	SQ→SP6 (SP7)	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_SPx_simd1_stall	SQ→SPx	1	Stall signal
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→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.

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_valid	SP6->SQ	1	Data valid
SP6_SQ_simd1_pred_kill_type	SP6->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

SQ_SPx_simd0_const	SQ→SPx	128	Constant broadcast
SQ_SPx_simd1_const	SQ→SPx	128	Constant broadcast
SQ_SPx_simd0_const_sel	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_simd1_const_sel	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

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

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	TYPE	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:

```
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.

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:

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 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 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 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?

Author: Steve Morein

Issue To:	**Copy No:**

R400 Architecture Proposal

ver 0.1

Overview: The is a proposal for the overall architecture of the R400. It is also just a proposal, and nothing is decided yet.

AUTOMATICALLY UPDATED FIELDS:

Document Location:	VST FireWire HD:r400 spec
Current Intranet Search Title :	R400 Top Level Spec

APPROVALS

Name/Dept	Signature/Date

Remarks:

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

ATI 2040
LG v. ATI
IPR2015-00325

Table Of Contents

Revision Changes:

Rev 0.0 (Steve Morein) Date: November 6, 2000 Initial revision.	Document started
Rev 0.01 Steve Morein Date: November 10,2000	Document continued

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.

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.

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)

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 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 2x2 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 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.

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 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

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.

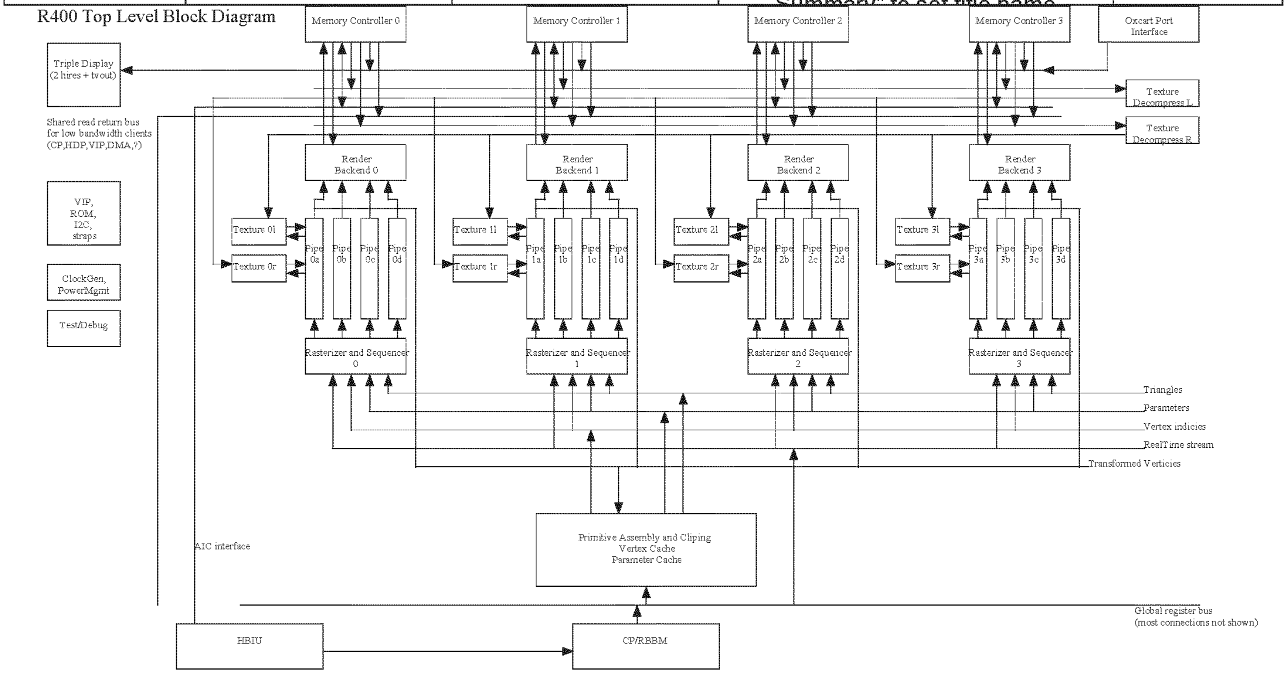
A LUT would exist in the display for gamma de-correction and pallet support.

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

R400 Top Level Block Diagram



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

9.1.2 *HDP*

9.1.3 *MISC*

9.1.4 *Rom*

9.1.5 *VIP*

9.1.6 *I2C*

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'*

Author: Steve Morein

Issue To:

Copy No:

R400 Top Level Specification

ver 0.2

Overview: This replaces the R400 architecture specification.

AUTOMATICALLY UPDATED FIELDS:
Document Location: Document1
Current Intranet Search Title : R400 Top Level Spec

APPROVALS	
Name/Dept	Signature/Date

Remarks:

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

Table Of Contents

Revision Changes:

Rev 0.0 (Steve Morein) Date: March 11, 2001 Initial revision.	Document recreated from earlier documents
Date March 14,2001	Finally got back to editing it.

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
RV400	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

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:

I/O	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 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 mip-mapping 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.

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. Schedule

		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 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.

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:

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, and read by the 3D pipe at the direct physical addresses. There are two options for the parameter memory. The parameter memory is not visible 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
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 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.

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 pixels	2560	
two lines	2	
two displays	2	
96 bits of color	96	(32 overlay + 64 bits primary)
total bits	960K 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 pixels	1600	
two lines	2	
two displays	2	
96 bits of color	96	(32 overlay + 64 bits primary)
total bits	600K bits	

which 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:

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.

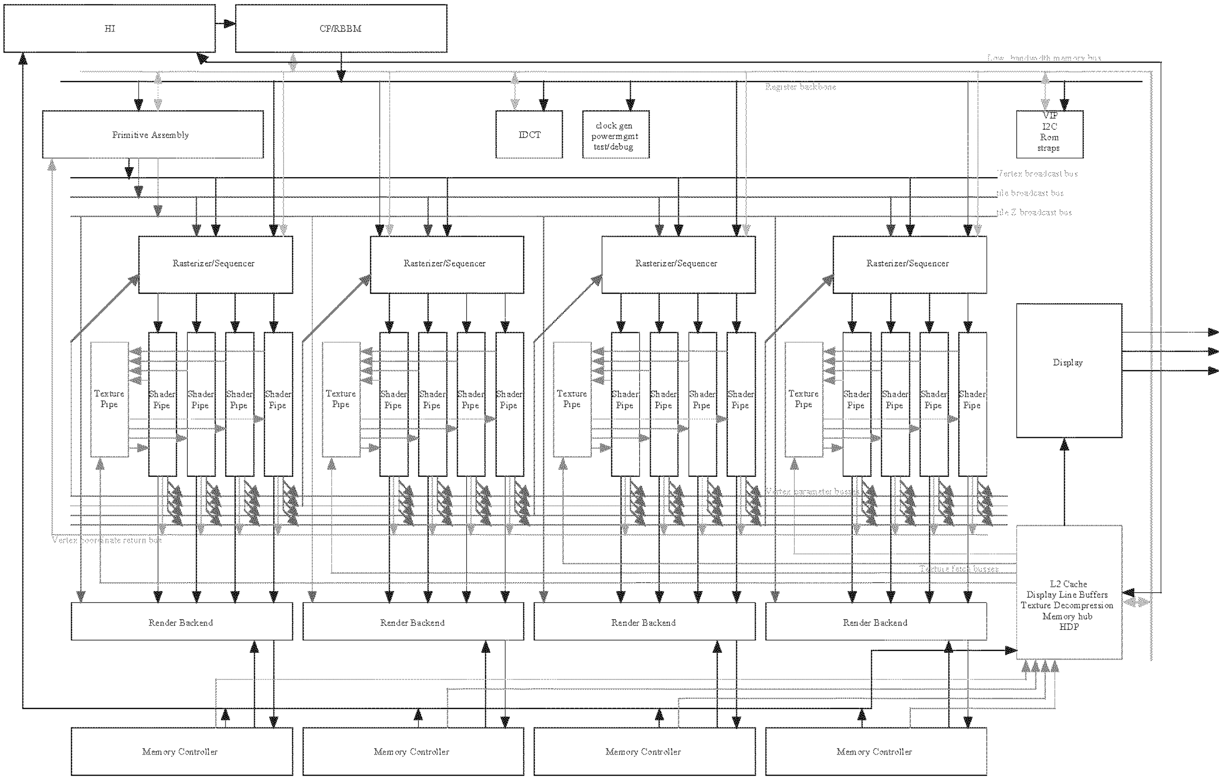
The display lives mostly in the core clock domain. There is a FIFO per pixel clock that crosses into the DAC/TMDS clock domain.

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

R400 Top Level 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

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?

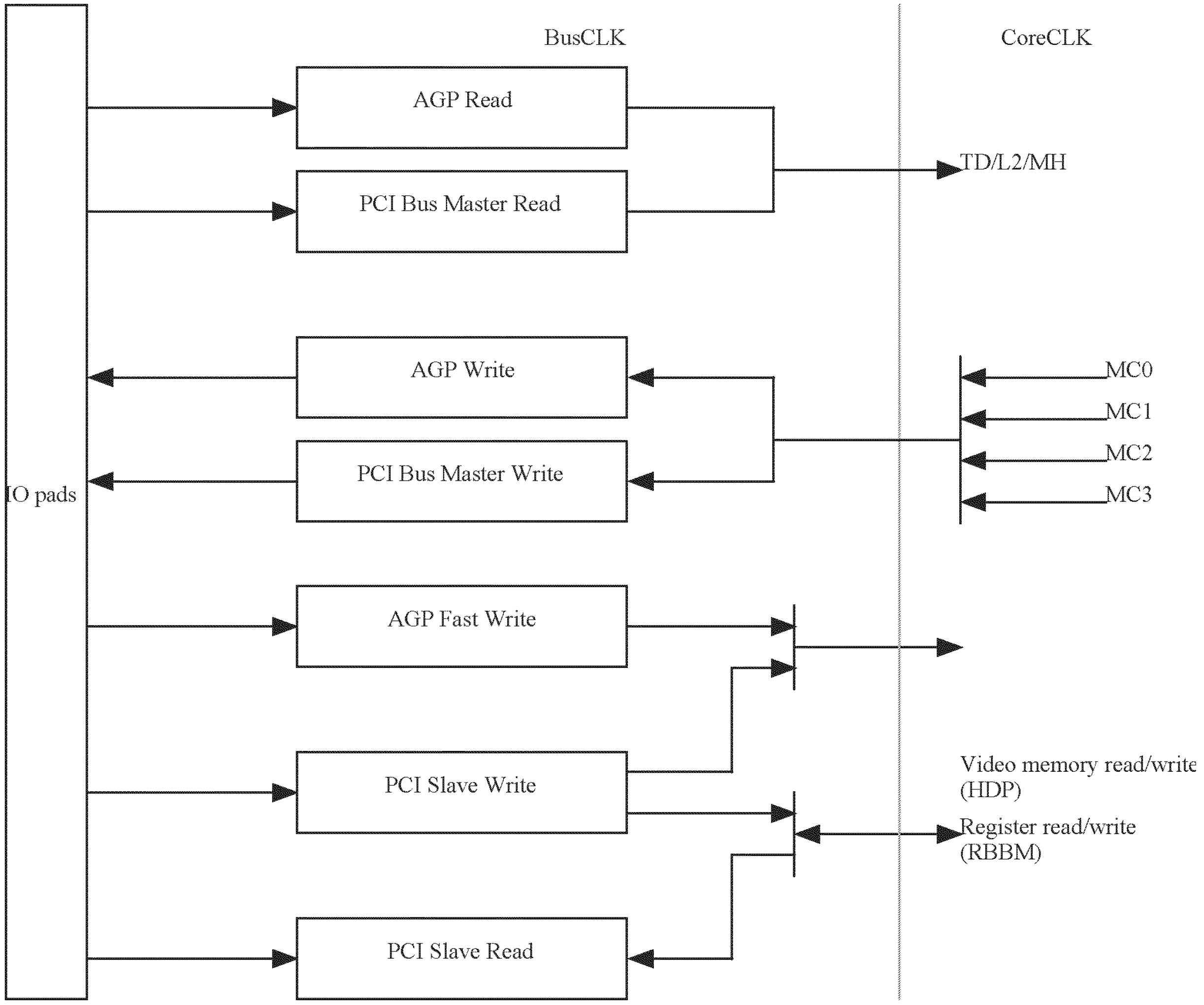
8.1.2 *Major interfaces*

The following busses connect the HBIU to the rest of the chip:

Bus	Chip client	Bus client	Description

Host register	CP/RBBM	PCI Slave	CPU reads and writes to chip registers
Host Data	HDP	PCI Slave AGP fast write	CPU reads and writes to video memory
AIC Write	MC	PCI Master (writes) AGP writes	Primarily blits to system memory, and control semaphore writes
AIC Read	TD/MH	PCI Master (reads) AGP reads	CP PM4 reads PA index reads State/vertex program loading Vertex loads AGP texture
Oddites for VGA			

8.1.3 *Block diagram*



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.

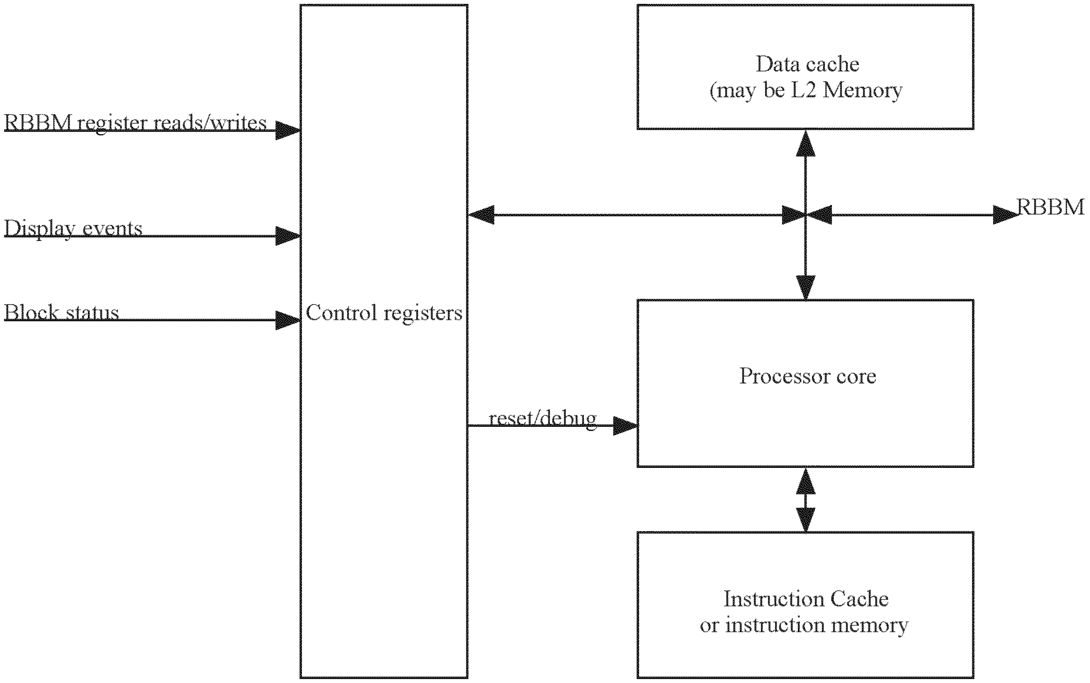
An alternative is to base the CP on a more generic RISC processor. It appears that this will save area, and make it possible to write the CP control program in C. The ARC core, for example, is less than 20K gates.

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
CP→MH	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

8.2.3 *Block diagram*



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:

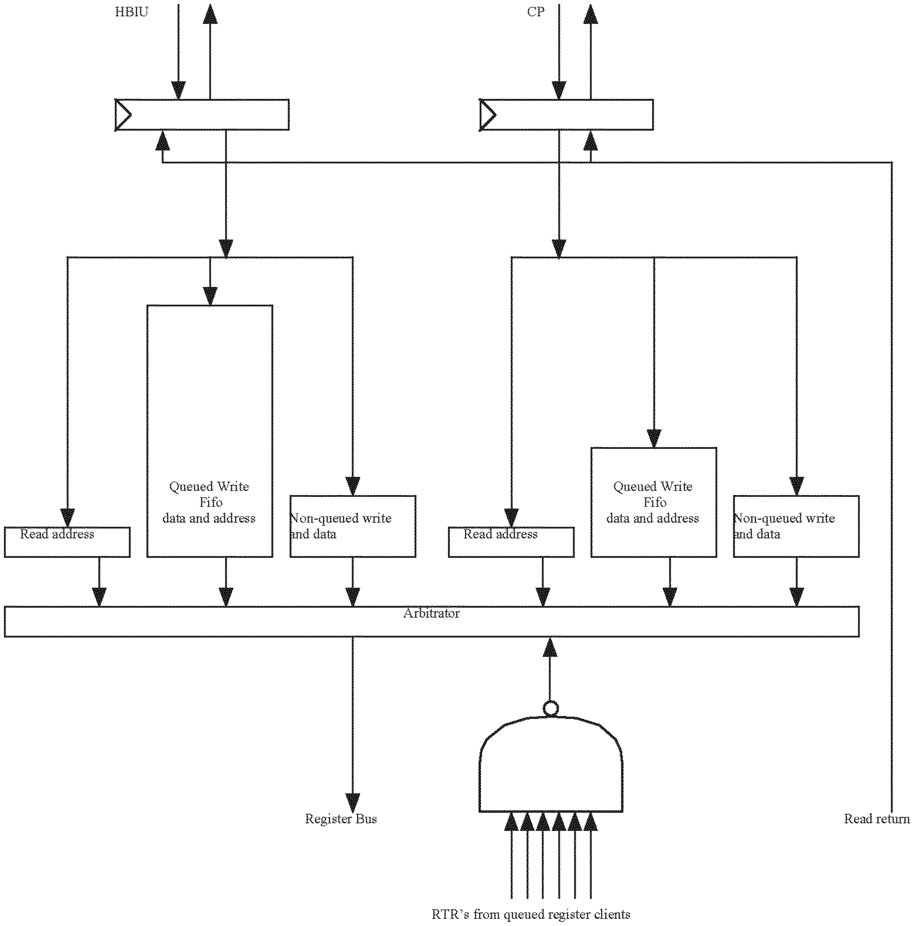
1) A much simpler register decoding scheme that does not need the RBBM to be aware of autoreg files.
2) A simpler register bus protocol that (for most registers) does not involve any feedback signals to the RBBM
3) Support for simple pipelining of the register bus to meet timing goals.
4) Much of the 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

8.3.3 *Block diagram*



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

WE	RBBM→	1	Write enable, address and data are valid
Addr[19:2]	RBBM→	18	Register address
Wm[3:0]	RBBM→	4	Register write mask, should be ignored by most clients
Wd[31:0]	RBBM→	32	Data
RE	RBBM→	1	Read Enable, address is 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

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

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

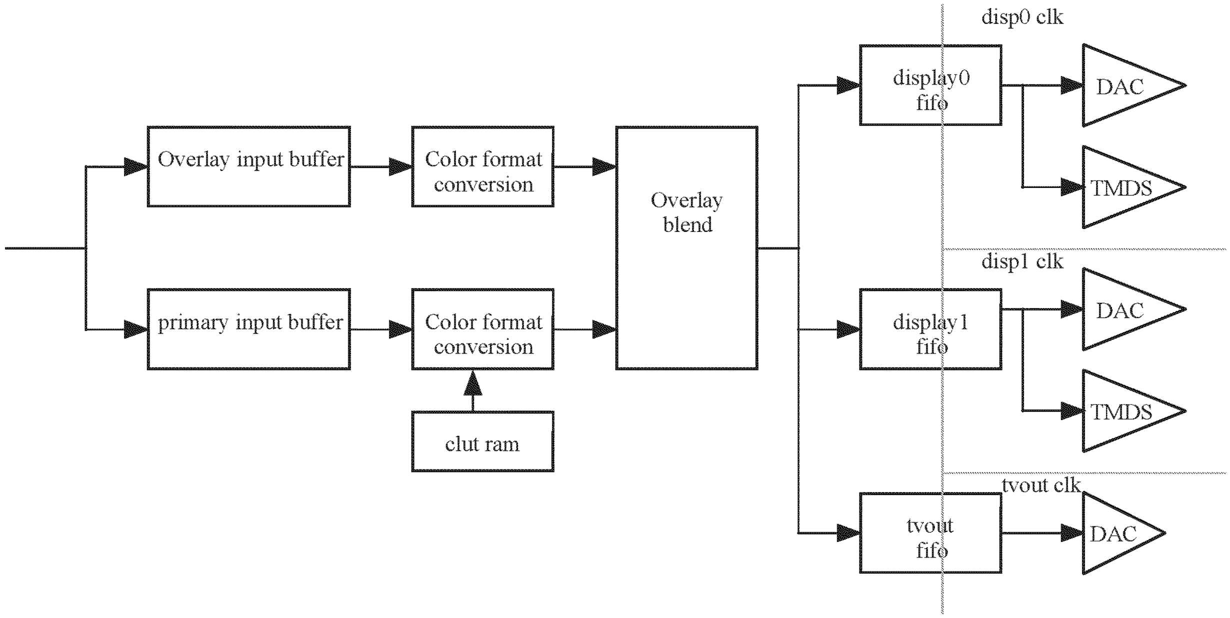
Overlay:
32bpp 8888 RGB
4:2:2 YUYV
the color conversion for the overlay is controlled by a programmable matrix, so the choice of color space is arbitrary.

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

8.9.3 *Block diagram*



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.

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

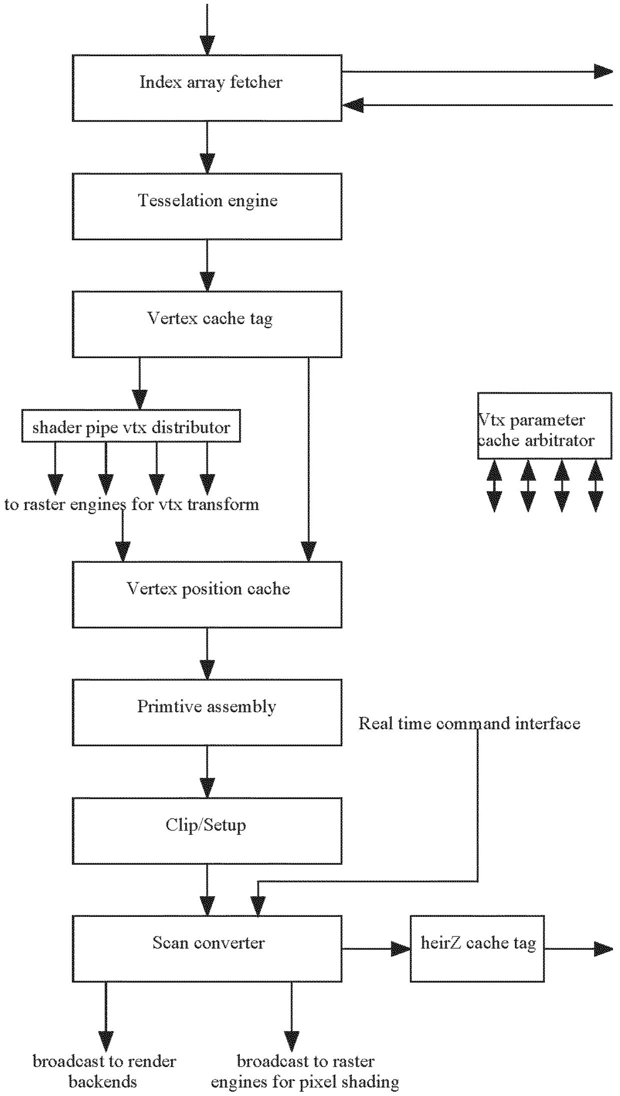
8.13.1 *Description*

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 rasterizer and makes the needed memory read requests. An arbitrator, centrally located in the PA block arbitrates 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
PA→MH	Index fetch path
PA→RE	Vertex transform packets
PA→MH	Hierarchical Z read request
PA→RBn	Coverage mask, position, and Z slope
PA→SPn	8x8 tiles to be rasterized

8.13.3 *Block diagram*


Index array fetcher
Tesselation engine
Vertex cache tag
shader pipe vtx distributor
to raster engines for vtx transform
Vtx parameter cache arbitrator
Vertex position cache
Primtive assembly
Real time command interface
Clip/Setup
Scan converter
heirZ cache tag
broadcast to render backends
broadcast to raster engines for pixel shading

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:

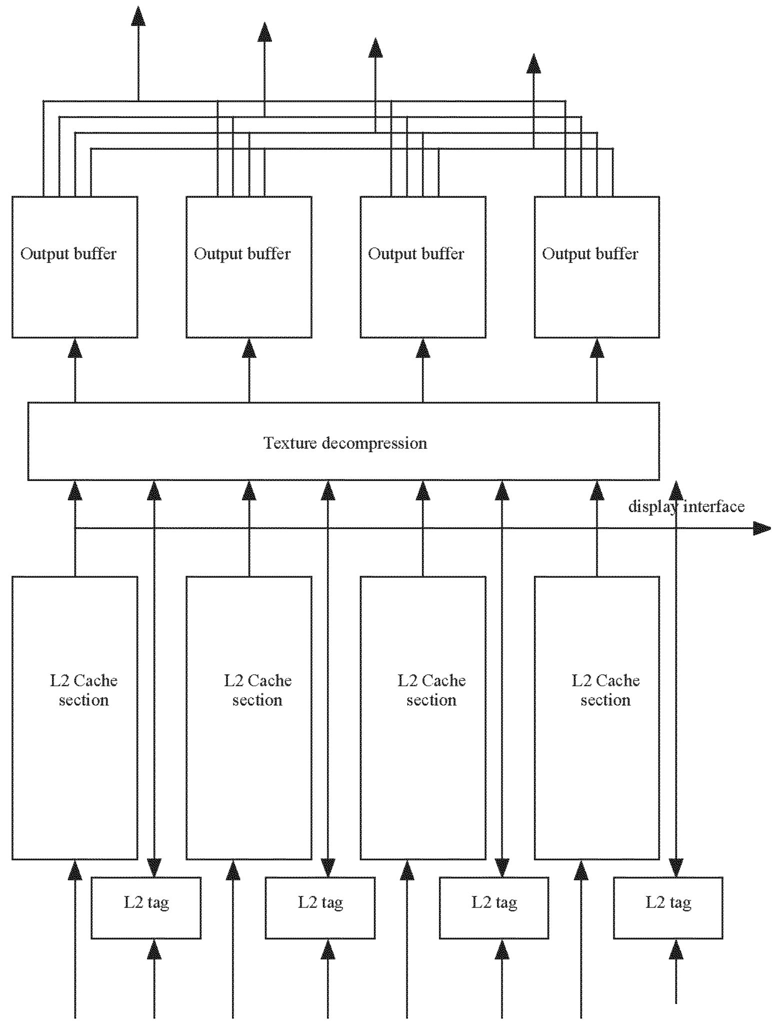
32 bpp (8888) unsigned
32 bpp (8888) signed
32 bpp (16,16) unsigned and signed
32 bpp (32) unsigned and signed

we may also support a 8bpp mono format to improve the performance of shadow buffering.

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 RE – Raster Engine

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.

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 l0 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 perspectivly 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 stage, holding up vectors until the vector currently occupying a reservation station has left.. A vector at a 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.

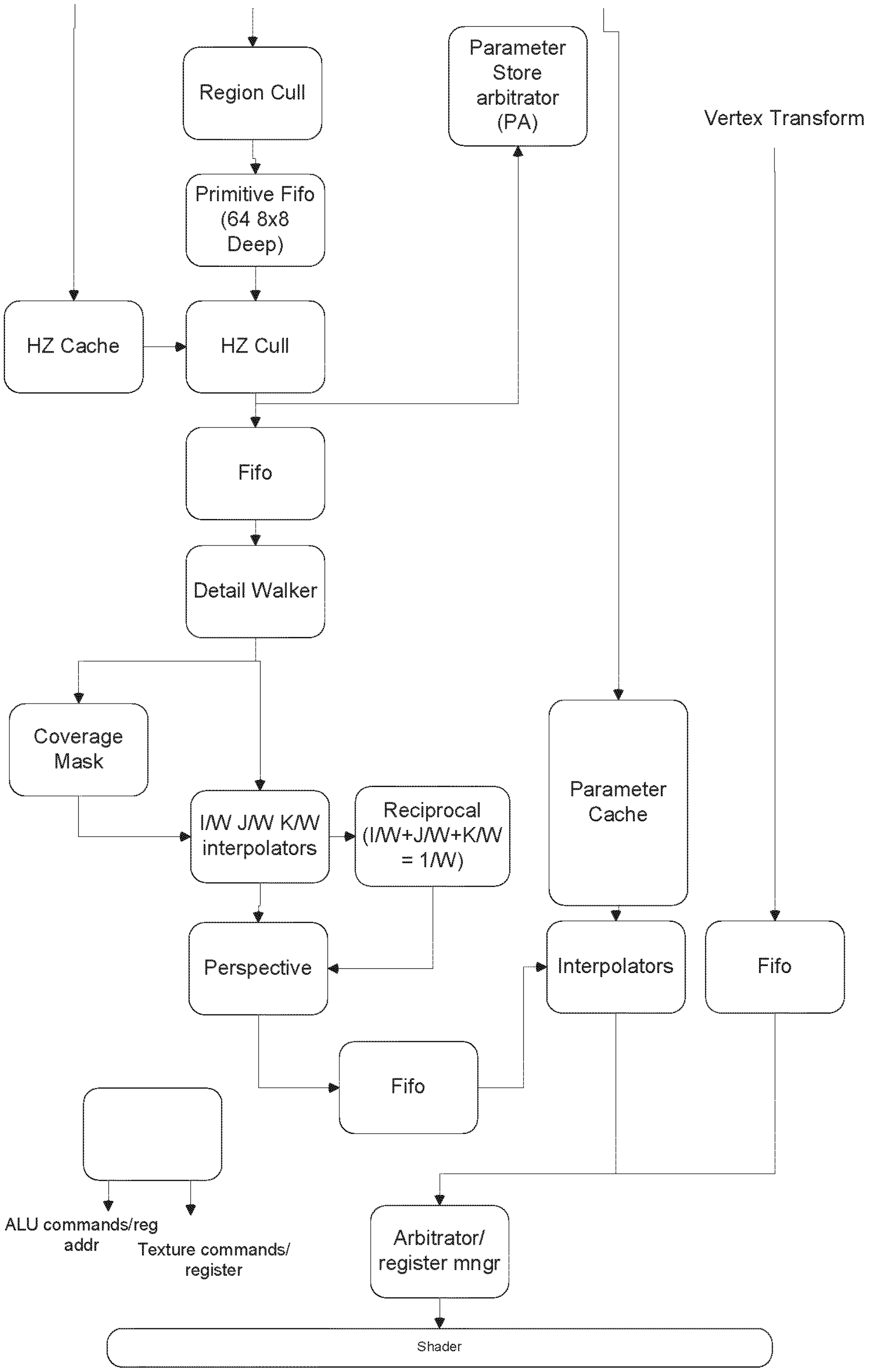
To support the shader pipe the raster engine also contains the shader instruction cache and constant store.

8.15.2 *Major interfaces*

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

8.15.3 *Block diagram*

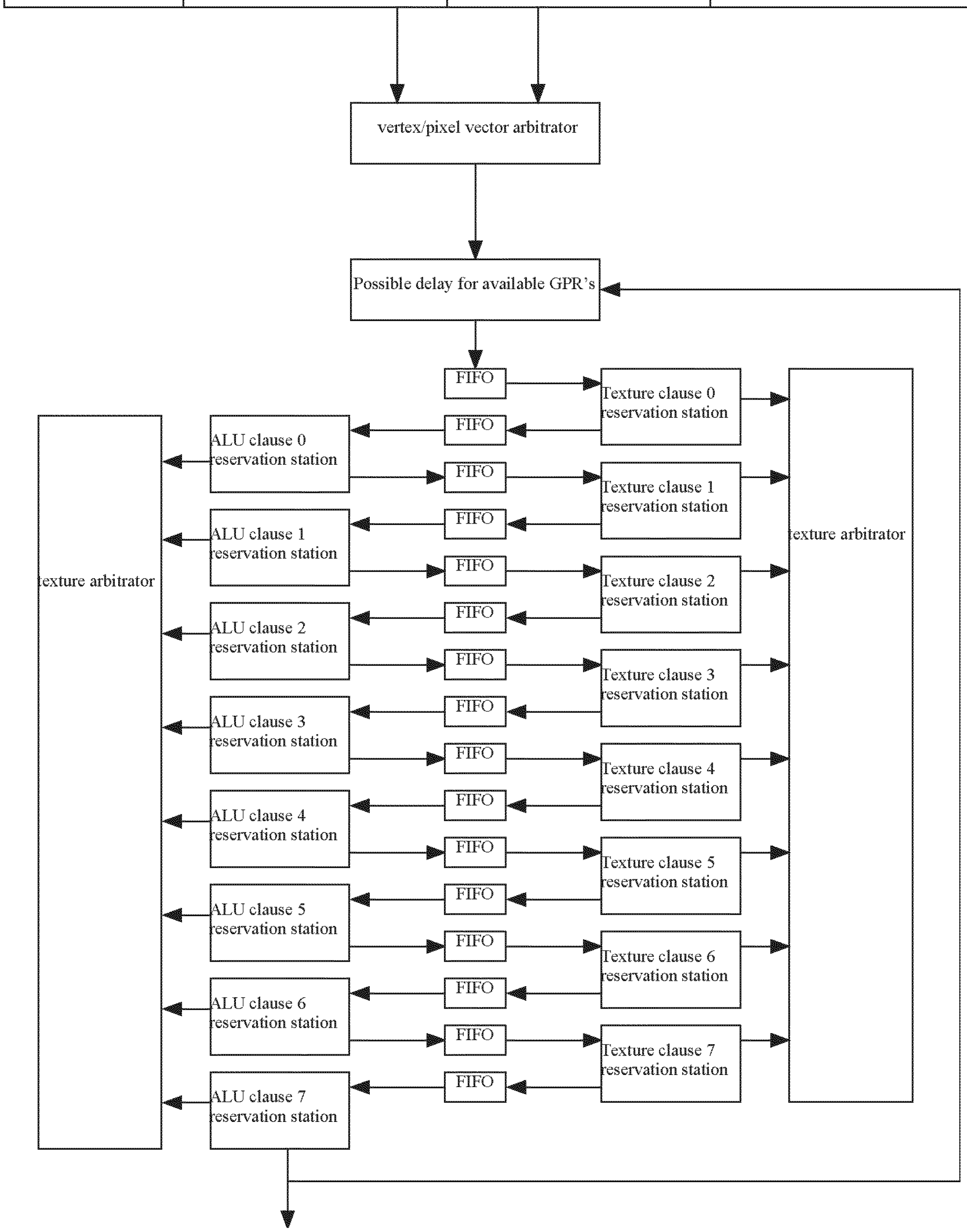
8.15.3.1 RE Block diagram



Region Cull
Parameter Store arbitrator (PA)
Vertex Transform
Primitive Fifo (64 8x8 Deep)
HZ Cache
HZ Cull
Fifo
Detail Walker
Coverage Mask
I/W J/W K/W interpolators
Reciprocal (I/W+J/W+K/W = 1/W)
Parameter Cache
Perspective
Interpolators
Fifo
Fifo
ALU commands/reg addr
Texture commands/ register
Arbitrator/ register mngr
Shader

8.15.3.2 RE sequencer

8.15.3.3 RE sequencer arbitrator


vertex/pixel vector arbitrator
Possible delay for available GPR's
FIFO
texture arbitrator
ALU clause 0 reservation station
ALU clause 1 reservation station
ALU clause 2 reservation station
ALU clause 3 reservation station
ALU clause 4 reservation station
ALU clause 5 reservation station
ALU clause 6 reservation station
ALU clause 7 reservation station
Texture clause 0 reservation station
Texture clause 1 reservation station
Texture clause 2 reservation station
Texture clause 3 reservation station
Texture clause 4 reservation station
Texture clause 5 reservation station
Texture clause 6 reservation station
Texture clause 7 reservation station
texture arbitrator

8.16 SP – Shader Pipe

8.16.1 *Description*

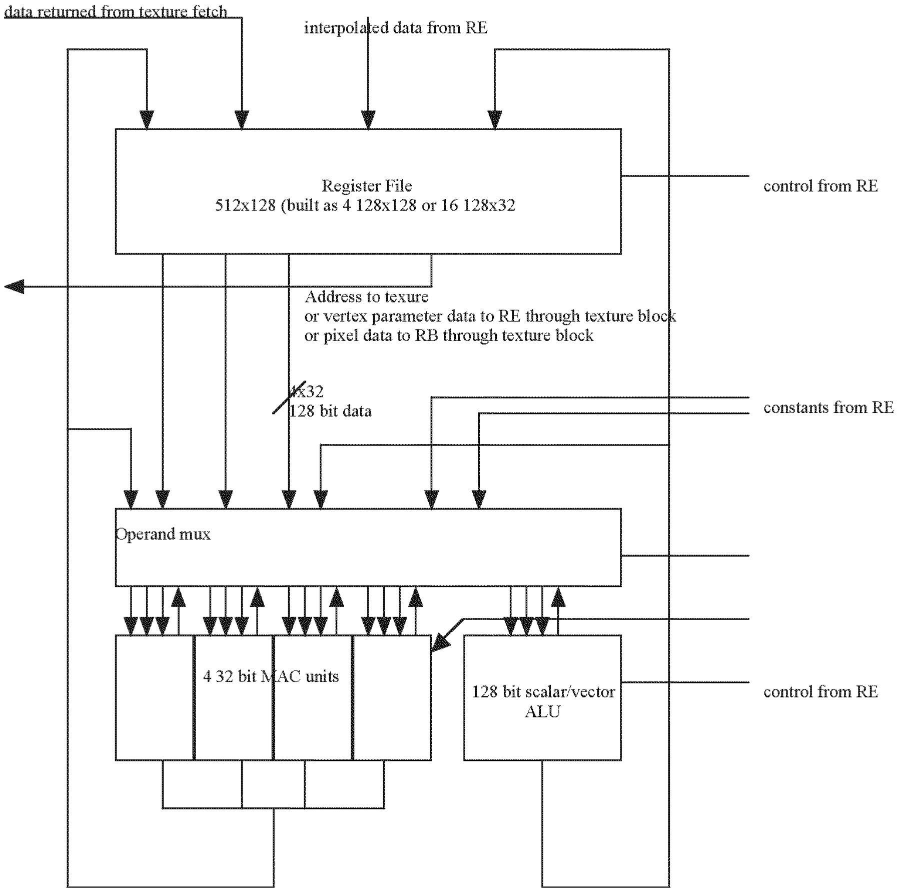
The shader pipe implements the math pipeline of the R400. It has no sequencing/control logic; the control is located in the raster engine.
The shader pipe contains four floating point MAC's, and an

8.16.2 *Major interfaces*

Bus	Description
RE→SP	Interpolated data
RE→SP	Control

SP→TX	Texture requests + vertex parameters + pixels to render backend
TX→SP	Returned texture data
RE→SP	Constants
SP→SP	Local w bus for derivative opcode

8.16.3 *Block diagram*


data returned from texture fetch
interpolated data from RE
Register File
512x128 (built as 4 128x128 or 16 128x32
control from RE
Address to texure
or vertex parameter data to RE through texture block
or pixel data to RB through texture block
4X32
128 bit data
constants from RE
Operand mux
4 32 bit MAC units
128 bit scalar/vector ALU
control from RE

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

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	23 January, 2001	4 September, 2015	GEN-CXXXXX-REVA	1 of 12

Author: Andrew Gruber, Andi Skende

Issue To:

Copy No:

Shader Processor

ver 0.1

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

AUTOMATICALLY UPDATED FIELDS:
Document Location : HD PC
Current Intranet Search Title: Shader Processor

APPROVALS	
Name/Dept	Signature/Date

Remarks:

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

Table Of Contents

Revision Changes:

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

Introduction

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

1. State

1.1 Shader State

1.1.1 *GPR*

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

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

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

It is possible for a single program/vector to request all 128 registers. This will make it impossible to hide memory latency, but the program will still execute and generate the correct result.

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

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

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

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

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

1.1.2 *Constant Registers*

There are also (192?) constant registers:

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

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

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	B	G	R		Tt0
					Tt1

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

Tt0 can be viewed as an 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.

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_f2	4	
TXWIDTH_f3	4	
TXHEIGHT_f3	4	
TXWIDTH_f4	4	
TXHEIGHT_f4	4	
TXWIDTH_f5	4	
TXHEIGHT_f5	4	
Alpha_mask?	1	
Chroma_key?	1	
Tex_coord_type	3	
LOD_BIAS	14	
TX_PITCH	14	Number of bits will decrease, used with non power 2 textures
Offset	32	Texture offset (includes endian and tile control)
Limit	32	Any memory accesses > limit will be killed, and the pixel that made the request will also be killed. If the access was from a vertex shader, then the vertex shader will for the x value of the vertex to be NAN which will kill all triangles that attempt to use the vertex.

1.3 Initial state

1.3.1 *Vertex Shader*

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

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

SRC C swizzle	12	3 bits for each component
Scalar/Alpha Output Mod	3	Output modifier for the result of the Scalar or Alpha channel operation
Vector Output Modifier	3	Output modifier on the vector result (RGB) when alpha operation is being coissued or ARGB when scalar operation is being coissued
Scalar/Alpha Clamp	1	
Vector Clamp	1	
Scalar/Alpha Write Mask	4	Defines which out of 32 bits words (four of them) in the result is written back In the register file
Vector Write Mask		Defines which out of 32 bits words (four of them) in the result is written back
Scalar/Alpha result pointer		Specifies the address into the register files for result of scalar/alpha operation
Vector result pointer		
Scalar Export flag		TBD
Vector Export flag		TBD

3.2 ALU Opcodes

The following opcodes are native, core opcodes:

Name	Function	Notes
MACC	D = A * B + C	(add is A*1.0 + C) (mul is A*B + 0.0) (nop/passthrough/move is A*1.0 + 0.0)
MSUB	D = A * B – C	(sub is A*1.0 – C)
MRSB	D = C – A*B	
DOT2	D = (A.x * B.x) + (A.y * B.y)	
DOT3	D = (A.x * B.x) + (A.y * B.y) + (A.z * B.z)	
DOT4	D = (A.x * B.x) + (A.y * B.y) + (A.z * B.z) + (A.w * B.w)	
RECP	D = 1/A.w	Use broadcast to select something other than w
CMxx	D = A if C 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.
Exp	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.

	The export opcodes must be in the last alu clause:	
EXPP	Export position: position = A, clipcodes = B	Export position and clip codes of vertex, end vertex shader
EXP	Export vertex component, {D} = A	Export vertex component to vertex cache, 16 possible destinations
EXPXY	Placeholder, {0} = 1,0,0,0	Placeholder for the rasterizer to put pixel position and Z into pixel shader.
EXPC	Export color	Export color of pixel, end pixel shader
EXPZ	Export Z	Export Z of pixel

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

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

3.3 Macro opcodes

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

4. Texture/Memory

4.1 Instruction Format

Destination control:

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

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

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

4.2 Opcodes

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

4.3 Example

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

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

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_type	3	?:1D array
LOD_BIAS	14	0
TX_PITCH	14	Number of bits will decrease, used with non power 2 textures
Offset	32	Texture offset (includes endian and tile control)
Limit	32	Any memory accesses > limit will be killed, and the pixel that made the request will also be killed. If the access was from a vertex shader, then the vertex shader will for the x value of the vertex to be NAN which will kill all triangles that attempt to use the vertex.