[redacted]

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 201519 April, 2002	GEN-CXXXXX-REVA	1 of 58

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SQ

Version ~~1.11~~2.0

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 [redacted] 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

1. Overview

~~The sequencer is based on the R300 design. It~~The sequencer chooses two ALU ~~clauses~~ threads and a fetch ~~clause~~ thread to execute, and executes all of the instructions in a ~~clause~~ block before looking for a new clause of the same type. Two ALU ~~clauses~~ threads are executed interleaved to hide the ALU latency. ~~Each vector will have eight fetch 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 fetch 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 choose an alu clause to execute and all eight fetch stations to choose a fetch clause to execute.~~ The arbitrator will give priority to ~~clauses/reservation stations closer to the bottom of the pipeline~~older threads. ~~It will not execute an alu clause until the fetch fetches initiated by the previous fetch clause have completed.~~ There are two separate ~~sets of~~ 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.

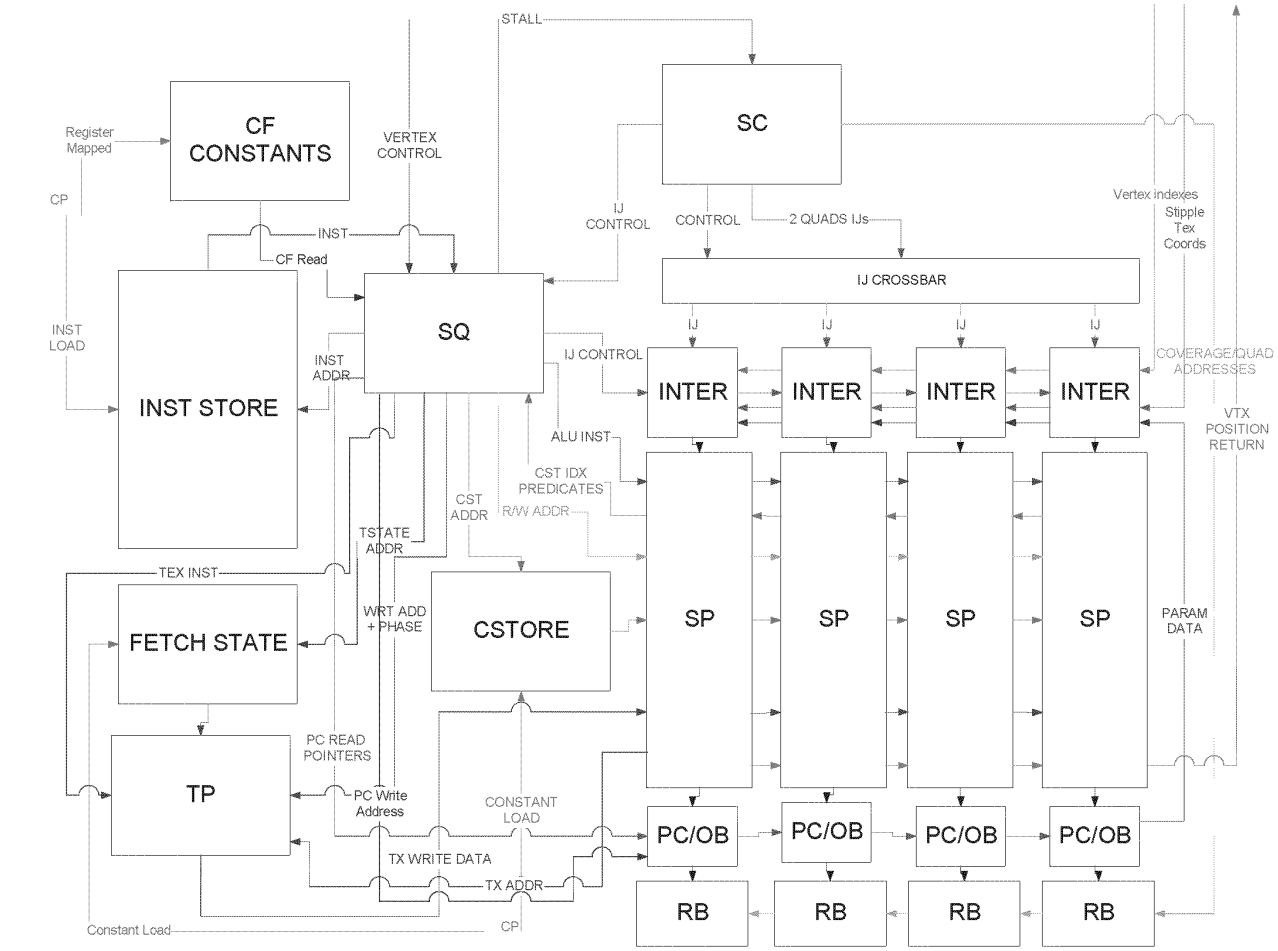
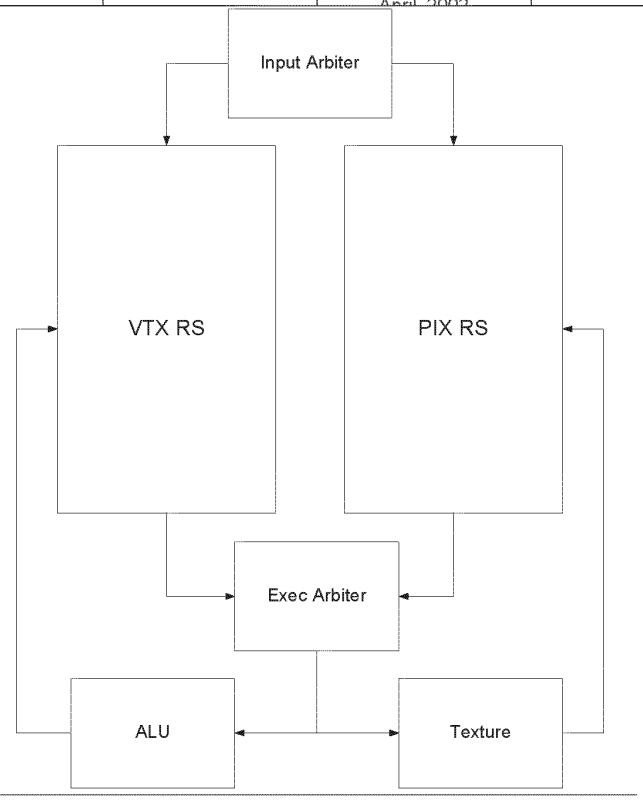


Figure 1: General Sequencer overview

1.1 Top Level Block Diagram


Input Arbiter
VTX RS
PIX RS
Exec Arbiter
ALU
Texture

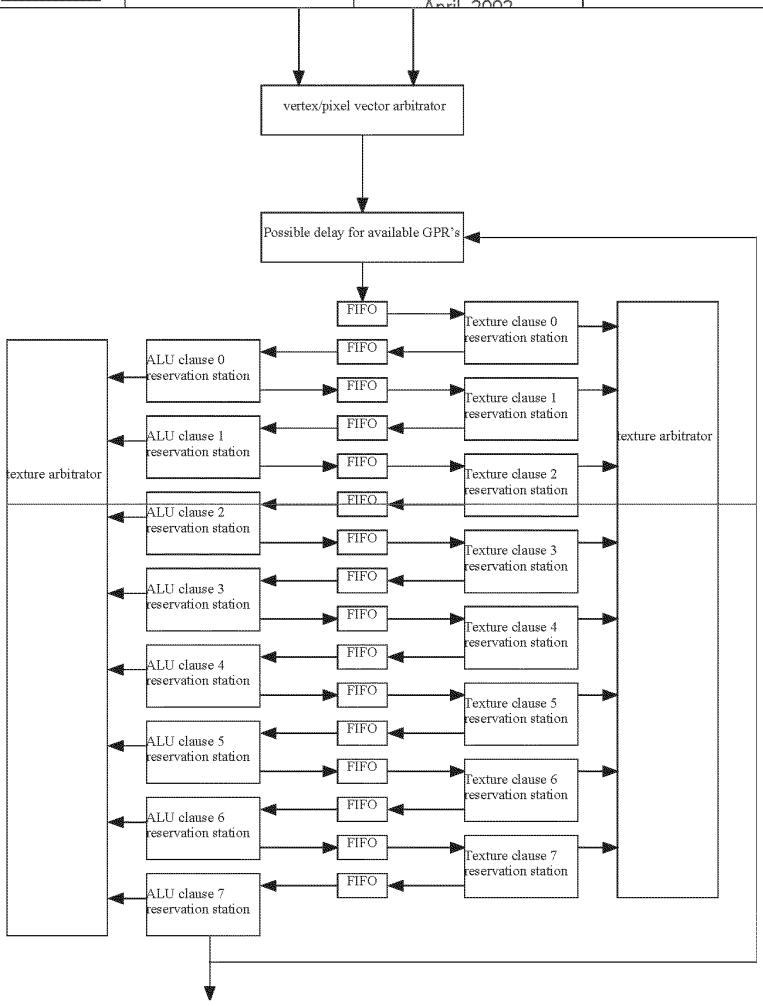


Figure 2: Reservation stations and arbiters

~~There are two sets of the above figure, one for vertices and one for pixels.~~

~~Depending on the arbitration state, the sequencer will either choose a vertex or a pixel packet. The control packet consists of 3 bits of state, 7 bits for the base address of the Shader program and some information on the coverage to determine fetch LOD plus other various small state bits.~~

~~On receipt of a packet, the input state machine (not pictured but just before the first FIFO) allocated enough space in the GPRs to store the interpolated values and temporaries. Following this, the barycentric coordinates (and XY screen position if needed) are sent to the interpolator, which will use them to interpolate the parameters and place the results into the GPRs. Then, the input state machine stacks the packet in the first FIFO.~~

~~On receipt of a command, the level 0 fetch machine issues a fetch request to the TP and corresponding GPR address for the fetch address (ta). A small command (tcmd) is passed to the fetch system identifying the current level number (0) as well as the GPR write address for the fetch return data. One fetch request is sent every 4 clocks causing the texturing of sixteen 2x2s worth of data (or 64 vertices). Once all the requests are sent the packet is put in FIFO 1.~~

~~Upon receipt of the return data, the fetch unit writes the data to the register file using the write address that was provided by the level 0 fetch machine and sends the clause number (0) to the level 0 fetch state machine to signify that the write is done and thus the data is ready. Then, the level 0 fetch machine increments the counter of FIFO 1 to signify to the ALU 0 that the data is ready to be processed.~~

~~On receipt of a command, the level 0 ALU machine first decrements the input FIFO 1 counter and then issues a complete set of level 0 shader instructions. For each instruction, the ALU state machine generates 3 source addresses, one destination address and an instruction. Once the last instruction has been issued, the packet is put into FIFO 2.~~

~~There will always be two active ALU clauses at any given time (and two arbiters). One arbiter will arbitrate over the odd instructions (4 clocks cycles) and the other one will arbitrate over the even instructions (4 clocks cycles). The only constraints between the two arbiters is that they are not allowed to pick the same clause number as the other one is currently working on if the packet is not of the same type (render state).~~

~~If the packet is a vertex packet, upon reaching ALU clause 3, it can export the position if the position is ready. So the arbiter must prevent ALU clause 3 to be selected if the positional buffer is full (or can't be accessed). Along with the positional data, if needed the sprite size and/or edge flags can also be sent.~~

~~A special case is for multipass vertex shaders, which can export 12 parameters per last 6 clauses to the output buffer. If the output buffer is full or doesn't have enough space the sequencer will prevent such a vertex group to enter an exporting clause.~~

~~Multipass pixel shaders can export 12 parameters to memory from the last clause only (7).~~

~~All other clauses process in the same way until the packet finally reaches the last ALU machine (7).~~

~~Only one pair of interleaved ALU state machines may have access to the register file address bus or the instruction decode bus at one time. Similarly, only one fetch state machine may have access to the register file address bus at one time. Arbitration is performed by three arbiter blocks (two for the ALU state machines and one for the fetch state machines). The arbiters always favor the higher number state machines, preventing a bunch of half finished jobs from clogging up the register files.~~
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)

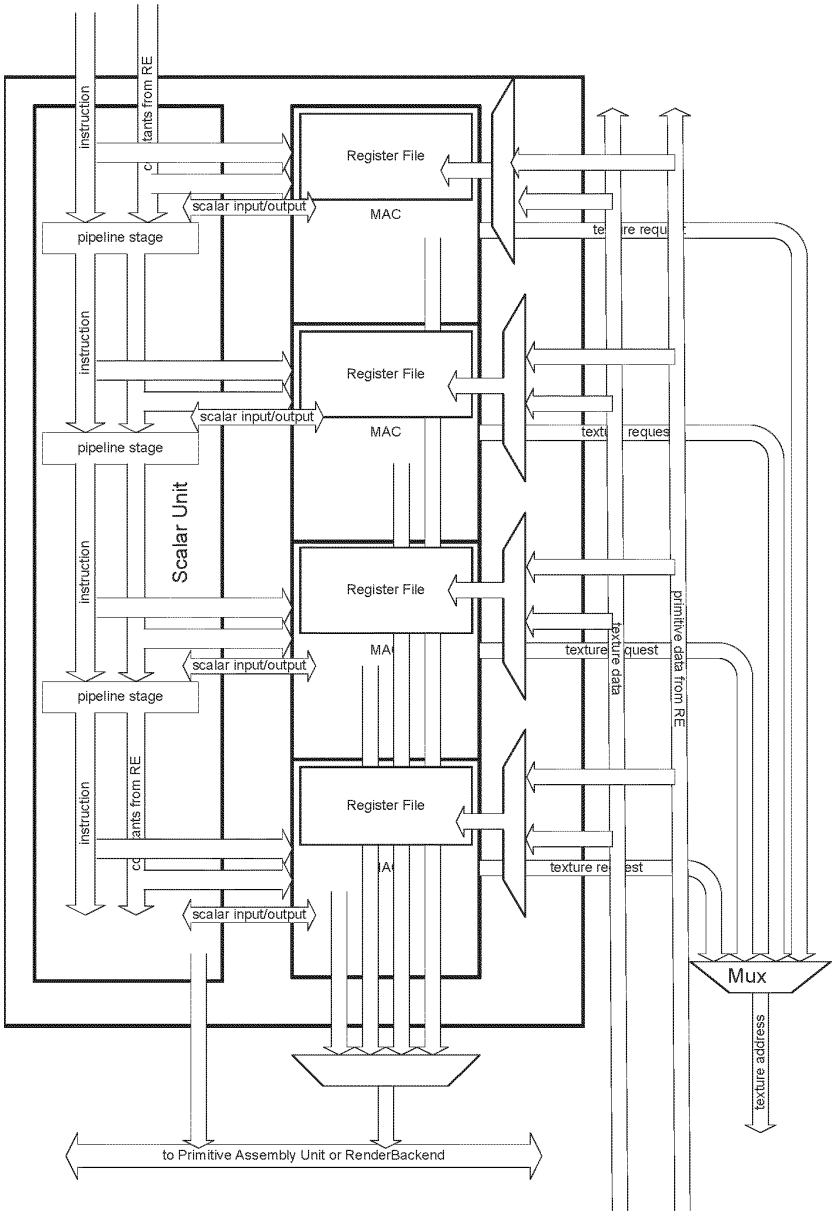


Figure 3: The shader Pipe

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

1.3 Control Graph

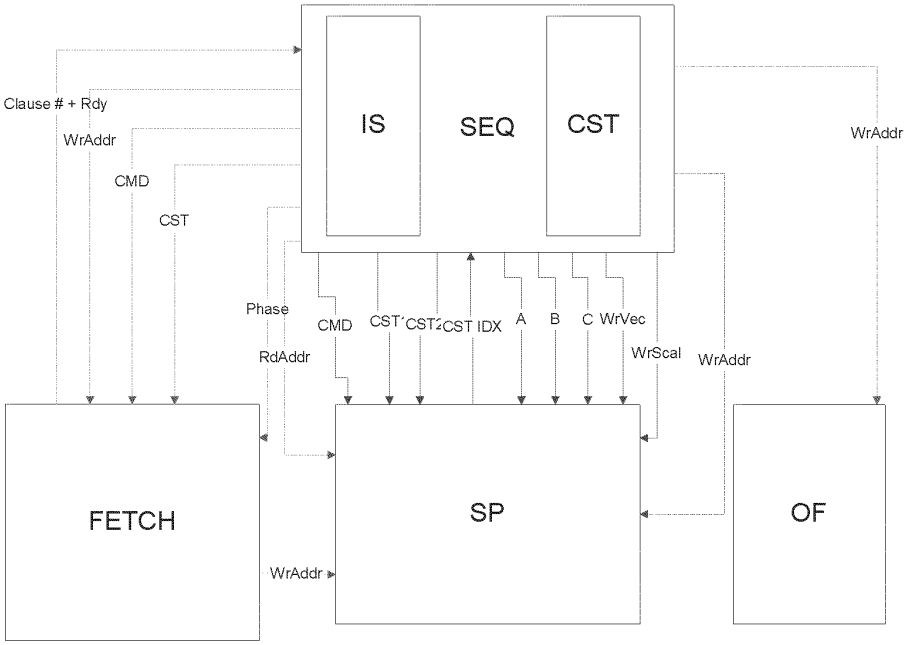


Figure 4: Sequencer Control interfaces

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.

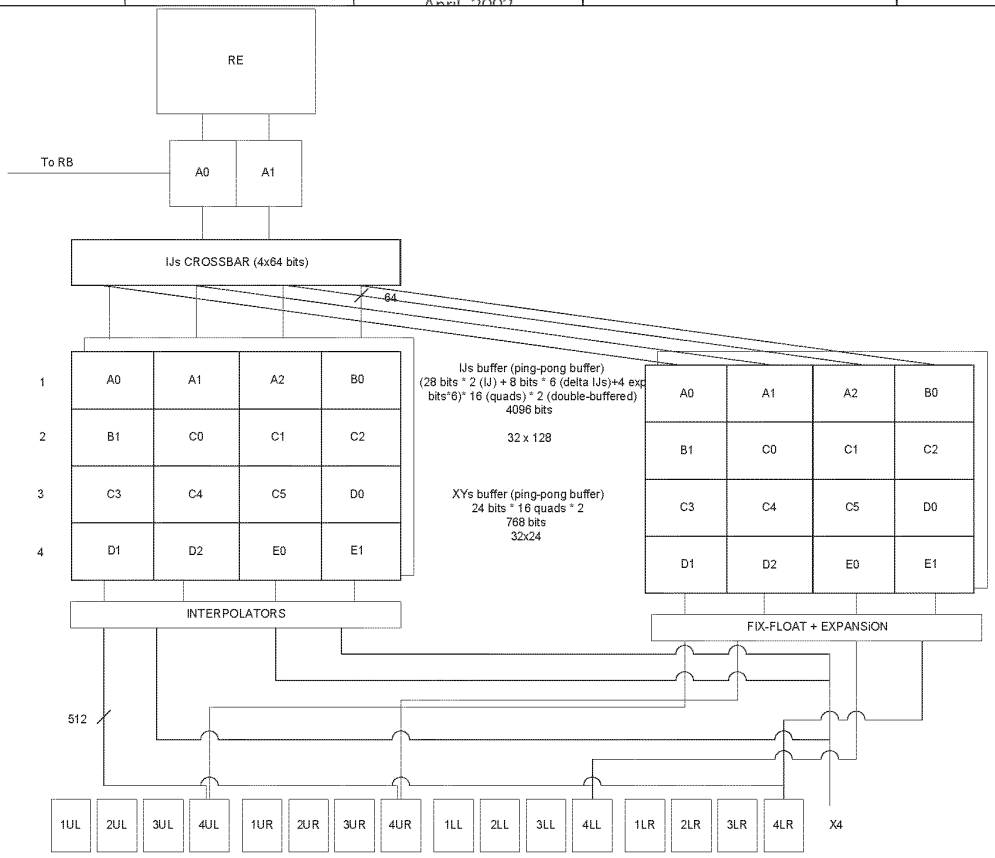


Figure 5: Interpolation buffers

	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 9: 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).

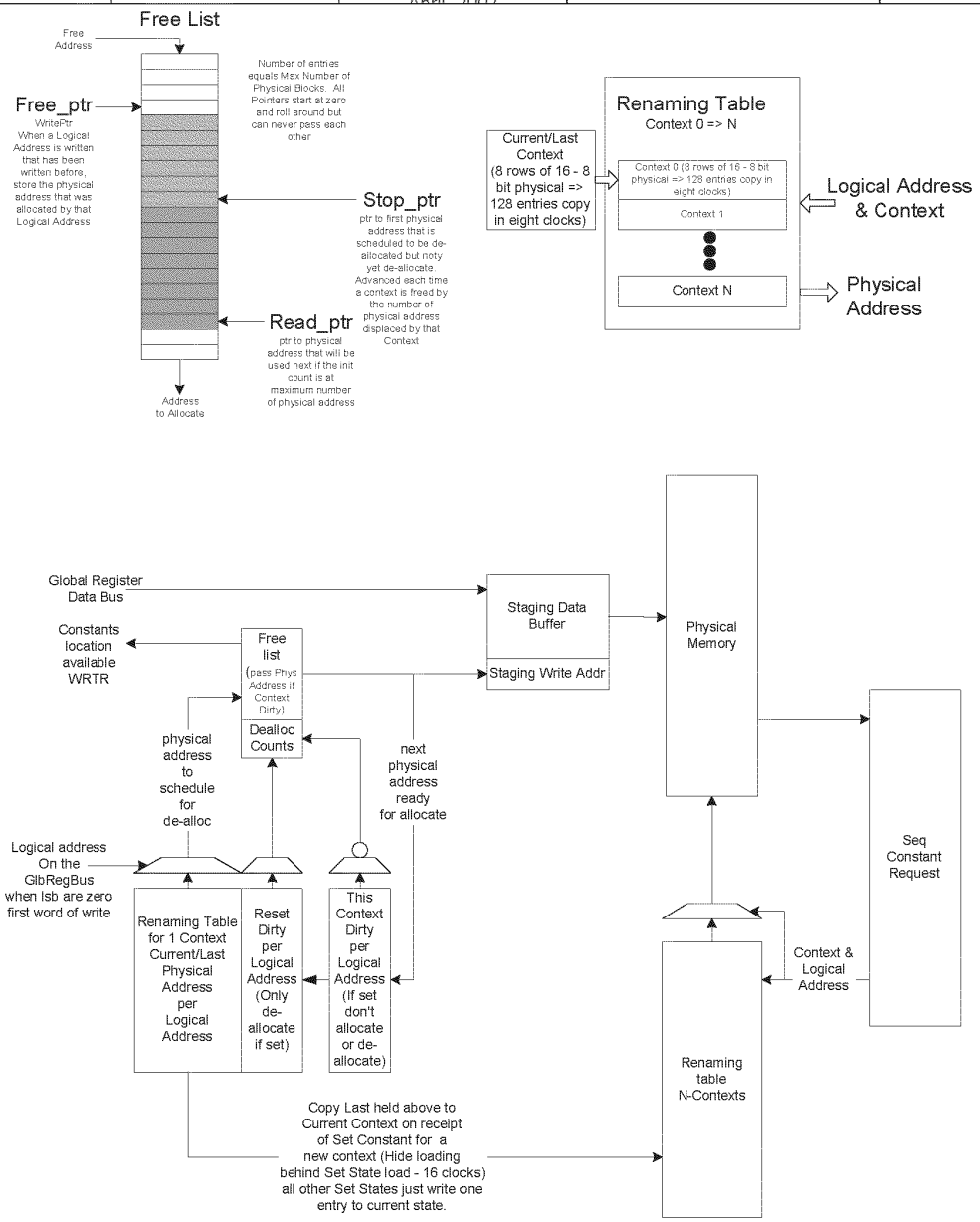


Figure 78: Constant management

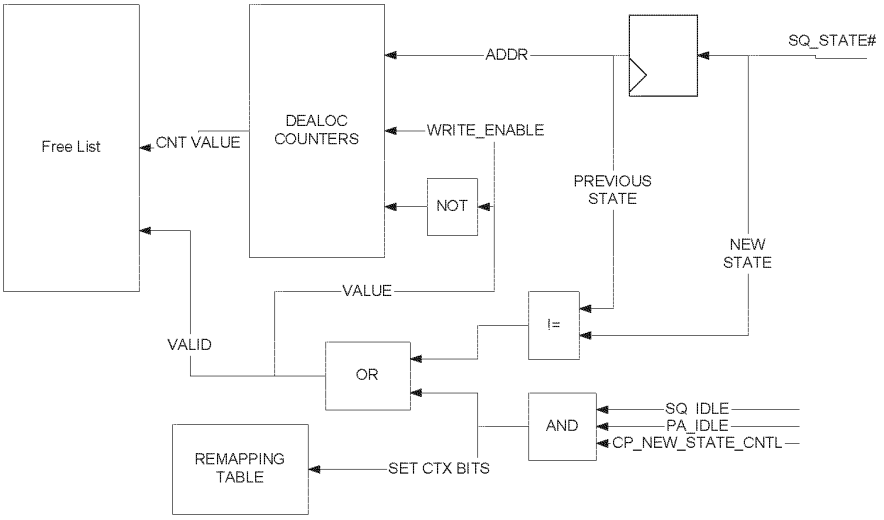


Figure 89: De-allocation mechanism for R400LE

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.

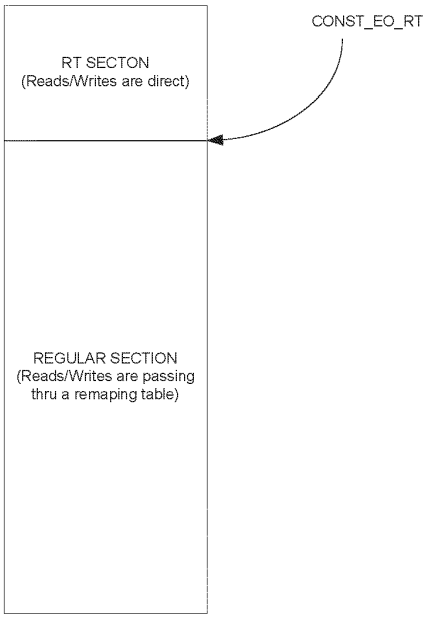


Figure 910: The instruction store

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.

Formatted: Bullets and Numbering

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).

Execute					
47	46... 43	40 ... 34	33 ...16	15...12	11 ... 0
Addressing	0001	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).

NOP		
47	46 ... 43	42 ... 0
Addressing	0010	RESERVED

This is a regular NOP.

Conditional_Execute						
47	46 ... 43	42	41 ... 34	33...16	15 ...12	11 ... 0
Addressing	0011	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.

Conditional_Execute_Predicates							
47	46 ... 43	42	41 ... 36	35 ... 34	33...16	15...12	11 ... 0
Addressing	0010	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.

Loop_Start				
47	46 ... 43	42 ... 17	16 ... 12	11 ... 0
Addressing	0101	RESERVED	loop ID	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	46 … 43	42 … 2017	19… 17	16 … 12	11 … 0
Addressing	0011	RESERVED	Predicate break	loop ID	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	46 … 43	42	41 …37	35 … 34	33 … 12	11 … 0
Addressing	0111	Condition	RESERVED	Predicate vector	RESERVED	Jump address

If the condition is met, jumps to the specified address and pushes the control flow program counter on the stack.

Return		
47	46 … 43	42 … 0
Addressing	1000	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	46 … 43	42	41… 34	33	32 … 12	11 … 0
Addressing	1001	Condition	Boolean address	FW only	RESERVED	Jump address

Allocate				
47	46 … 43	42…41	40 … 4	3 …0
Debug	1010	Buffer Select	RESERVED	Allocation 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).

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

End Of Program		
47	46 … 43	42… 0
RESERVED	1011	RESERVED

Marks the end of the program.

6.3 Implementation

Formatted: Bullets and Numbering

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 (12 bits),
2. Execution Count Marker 4 bits),
3. Loop Iterators (4x9 bits),
4. Call return pointers (4x12 bits),
5. Predicate Bits(4x64 bits),
6. Export ID (1 bit),
7. Parameter Cache base Ptr (7 bits),
8. GPR Base Ptr (8 bits),
9. Context Ptr (3 bits).
10. LOD corrections (6x16 bits)

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
- First thread of a new context
- Event thread (NULL thread that needs to trickle down the pipe)
- Last (1 bit)

Formatted: Bullets and Numbering

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.

~~Examples of control flow programs are located in the R400 programming guide document.~~

~~The basic model is as follows:~~

~~The render state defined the clause boundaries:~~
~~Vertex_shader_fetch[7:0][7:0] // eight 8 bit pointers to the location where each clauses control program is located~~
~~Vertex_shader_alu[7:0][7:0] // eight 8 bit pointers to the location where each clauses control program is located~~
~~Pixel_shader_fetch[7:0][7:0] // eight 8 bit pointers to the location where each clauses control program is located~~
~~Pixel_shader_alu[7:0][7:0] // eight 8 bit pointers to the location where each clauses control program is located~~

~~A pointer value of FF means that the clause doesn't contain any instructions.~~

~~The control program for a given clause is executed to completion before moving to another clause, (with the exception of the pick two nature of the alu execution). The control program is the only program aware of the clause boundaries.~~

~~The control program has nine basic instructions:~~

~~Execute~~
~~Conditional_execute~~
~~Conditional_Execute_Predicates~~
~~Conditional_jump~~
~~Conditionnal_Call~~
~~Return~~
~~Loop_start~~
~~Loop_end~~
~~NOP~~

~~Execute, causes the specified number of instructions in instruction store to be executed.~~
~~Conditional_execute checks a condition first, and if true, causes the specified number of instructions in instruction store to be executed.~~
~~Loop_start resets the corresponding loop counter to the start value on the first pass after it checks for the end condition and if met jumps over to a specified address.~~
~~Loop_end increments (decrements?) the loop counter and jumps back the specified number of instructions.~~
~~Conditionnal_Call jumps to an address and pushes the IP counter on the stack if the condition is met. On the return instruction, the IP is popped from the stack.~~

Conditional_execute_Predicates executes a block of instructions if all bits in the predicate vectors meet the condition.
Conditional_jumps jumps to an address if the condition is met.
NOP is a regular NOP

NOTE THAT ALL JUMPS MUST JUMP TO EVEN CFP ADDRESSES since there are two control flow instructions per memory line. Thus the compiler must insert NOPs where needed to align the jumps on even CFP addresses.

Also if the jump is logically bigger than pshader_cntl_size (or vshader_cntl_size) we break the program (clause) and set the debug registers. If an execute or conditional_execute is lower than cntl_size or bigger than size we also break the program (clause) and set the debug registers.

We have to fit instructions into 48 bits in order to be able to put two control flow instruction per line in the instruction store.

A value of 1 in the Addressing means that the address specified in the Exec Address field (or in the jump address field) is an ABSOLUTE address. If the addressing field is cleared (should be the default) then the address is relative to the base of the current shader program.

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

Execute up to 4k instructions at the specified address in the instruction memory. If Last is set, this is the last group of instructions of the clause.

This is a regular NOP. If Last is set, this is the last instruction of the clause.

If the specified Boolean (8 bits can address 256 Booleans) meets the specified condition then execute the specified instructions (up to 4k instructions). If Last is set, then if the condition is met, this is the last group of instructions to be executed in the clause. If the condition is not met, we go on to the next control flow instruction.

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 Last is set, then if the condition is met, this is the last group of instructions to be executed in the clause. If the condition is not met, we go on to the next control flow instruction.

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. 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.

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

If the condition is met, jumps to the specified address and pushes the control flow program counter on the stack.

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.

If condition met, jumps to the address. FORWARD jump only allowed if bit 31 set. Bit 31 is only an optimization for the compiler and should NOT be exposed to the API.

~~To prevent infinite loops, we will keep 9 bits loop iterators instead of 8 (we are only able to loop 256 times). If the counter goes higher than 255 then the loop_end or the loop_start instruction is going to break the loop and set the debug GPRs.~~

~~6.3~~6.4 Data dependant predicate instructions

Formatted: Bullets and Numbering

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?}

~~6.4~~6.5 HW Detection of PV,PS

Formatted: Bullets and Numbering

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.

~~6.5~~6.6 Register file indexing

Formatted: Bullets and Numbering

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.

~~Predicated Instruction support for Texture clauses~~

~~For texture clauses, we support the following optimization: we keep 1 bit (thus 4 bits for the four predicate vectors) per predicate vector in the reservation stations. A value of 1 means that one ore more elements in the vector have a value of one (thus we have to do the texture fetches for the whole vector). A value of 0 means that no elements in the vector have his predicate bit set and we can thus skip over the texture fetch. We have to make sure the invalid pixels aren't considered with this optimization.~~

~~6.66.7~~ Debugging the Shaders

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

~~6.6.16.7.1~~ *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?}

~~6.6.26.7.2~~ *Method 2: Exporting the values in the GPRs ~~(12)~~*

The sequencer will have a debug active, count register and an address register for this mode ~~and 3 bits per clause specifying the execution mode for each clause. The modes can be :~~

~~Normal~~

~~2)Debug Kill~~

~~2)1)Debug Addr + Count~~

Formatted: Bullets and Numbering

Under the normal mode execution follows the normal course. Under the kill mode, all control flow instructions are executed but all normal shader instructions of the clause are replaced by NOPs. Only debug_export instructions of clause 7 will be executed under the debug kill setting. Under the other mode, normal execution is done until we reach an address specified by the address register and instruction count (useful for loops) specified by the count register. After we have hit the instruction n times (n=count) we switch the clause to the kill mode.

Under the debug mode (debug kill OR debug Addr + count), it is assumed that the programclause 7 is always exporting 12 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:

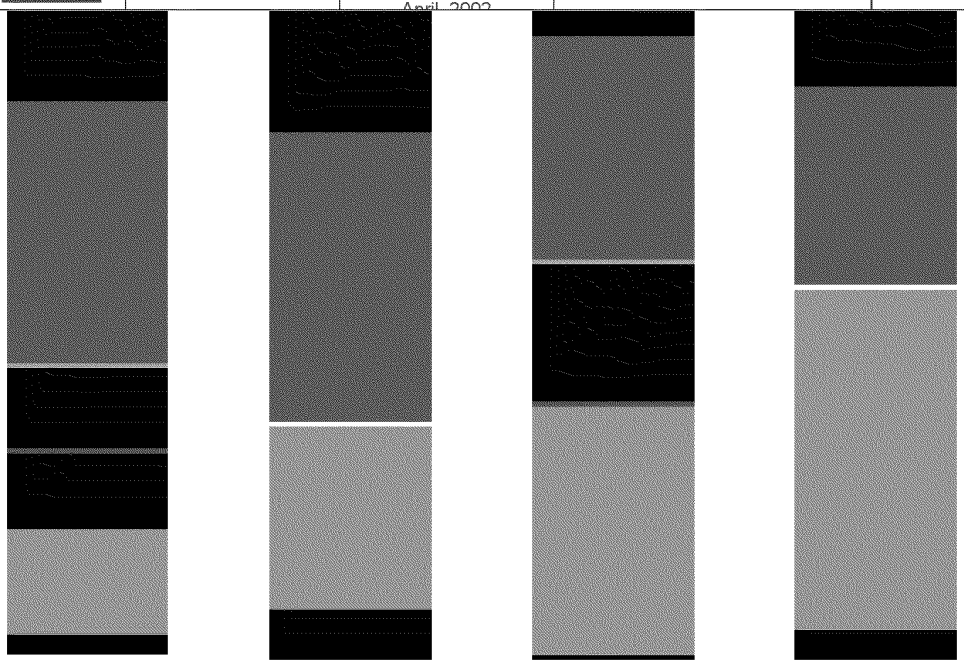
MASK_SETE
MASK_SETNE
MASK_SETGT
MASK_SETGTE

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 8 potentially pending fetch clauses to be executed. The choice is made by looking at the fifos from 7 to 0 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 8 potentially pending ALU clauses to be executed. The choice is made by looking at the fifos from 7 to 0 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 the exporting clause (3?). 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). The interpolation is done at a different precision across the 2x2. The upper left pixel's parameters are always interpolated at full 20x24 mantissa precision. Then the result of the interpolation along with the difference in IJ in reduced precision is used to interpolate the parameter for the other three pixels of the 2x2. Here is how we do it:

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

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

P0	P1
P2	P3

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

P0 is computed at 20x24 mantissa precision and P1 to P3 are computed at 8X24 mantissa precision. So far no visual degradation of the image was seen using this scheme.

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

Adds: 8

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

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

Total number of bits : 20*2 + 8*6 + 4*8 + 4*2 = 128

All numbers are kept using the un-normalized floating point convention: if exponent is different than 0 the number is normalized if not, then the number is un-normalized. The maximum range for the IJs (Full precision) is +/- 63 and the range for the Deltas is +/- 127.

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 barycentric coordinates are the same.

We start with the premise that if A = B and B = C and C = A, then P0,1,2,3 = A. Since one or more of the IJ terms may be zero, so we extend this to:

```
if (A=B and B=C and C=A)
  P0,1,2,3 = A;
else if ((I = 0) or (J = 0)) and
        ((J = 0) or (1-I-J = 0)) and
        ((1-J-I = 0) or (I = 0))) {
            if(I != 0) {
               P0 = A;
            } else if(J != 0) {
               P0 = B;
            } else {
               P0 = C;
            }
          //rest of the quad interpolated normally
}
else
{
          normal interpolation
}
```

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 VGT will send 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 still be sent by the VGT to the SQ **BUT** will not be processed by the SP and thus should be considered invalid (by the SU and VGT).

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 12~~. 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 11~~.

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~~11~~:Area Estimate for VGT to Shader Interface

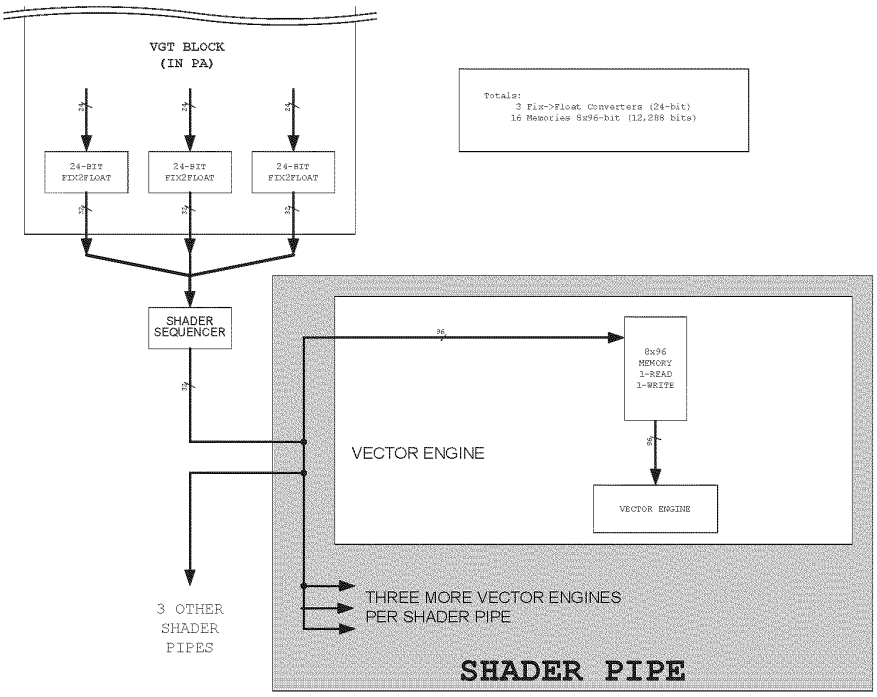


Figure 1112:VGT to Shader Interface

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	ADDRESS
4 bits	7 bits

The PA generates the parameter cache addresses as the positions come from the SQ. All it needs to do is keep a Current_Location pointer (7 bits only) and as the positions comes increment the memory number. When the memory number field wraps around, the PA increments the Current_Location by VS_EXPORT_COUNT_7 (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_7 = 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 _7to 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. Vertex position exporting~~

~~On clause 3 the vertex shader can export to the PA both the vertex position and the point sprite. It can also do so at clause 7 if not done at clause 3. The storage needed to perform the position export is at least 64x128 memories for the position and 64x32 memories for the sprite size. It is going to be taken in the pixel output fifo from the SX blocks. The clause where the position export occurs is specified by the EXPORT_LATE register. If turned on, it means that the export is going to occur at ALU clause 7 if unset position export occurs at clause 3.~~

~~19. Exporting Arbitration~~

~~Here are the rules for co-issuing exporting ALU clauses.~~

~~1)Position exports and position exports cannot be co-issued.~~

~~All other types of exports can be co-issued as long as there is place in the receiving buffer.~~

Formatted: Bullets and Numbering

~~20.~~ Exporting Rules

~~20.1~~ Parameter caches exports

~~We support masking and out of order exports to the parameter caches. So one can export multiple times to the same PC line using different masks.~~

~~20.2~~ Memory exports

~~Memory exports don't support masking. However, you can export out of order to memory locations.~~

~~20.3~~ Position exports

~~Position exports have to be done IN ORDER and don't support masking.~~

~~21.~~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:

~~21.1~~18.1 Vertex Shading

0:15 - 16 parameter cache
16:31 - Empty (Reserved?)
32 - Export Address
33:40 - 8 vertex exports to the frame buffer and index
41:47 - Empty
48:55 - 8 debug export (interpret as normal vertex export)
60 - export addressing mode
61 - Empty
62 - position
63 - sprite size export that goes with position export (point_h,point_w,edgeflag,misc)

~~21.2~~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
4:7 - Empty
8 - Buffer 0 Color/Fog (primary)
9 - Buffer 1 Color/Fog
10 - Buffer 2 Color/Fog
11 - Buffer 3 Color/Fog
12:15 - Empty
16:31 - Empty (Reserved?)
32 - Export Address
33:40 - 8 exports for multipass pixel shaders.
41:47 - Empty
48:55 - 8 debug exports (interpret as normal pixel export)
60 - export addressing mode
61:62 - Empty
63 - Z for primary buffer (Z exported to 'alpha' component)

Formatted: Bullets and Numbering
Formatted: Bullets and Numbering
Formatted: Bullets and Numbering
Formatted: Bullets and Numbering
Formatted: Bullets and Numbering
Formatted: Bullets and Numbering

[redacted]

22.19. Special Interpolation modes

22.119.1 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.

22.219.2 Sprites/ XY screen coordinates/ FB information

When working with sprites, one may want to overwrite the parameter 0 with SC generated data. Also, XY screen coordinates may be needed in the shader program. This functionality is controlled by the gen_l0 register (in SQ) in conjunction with the SND_XY register (in SC). 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:

Gen_st is a bit taken from the interface between the SC and the SQ. This is the MSB of the primitive type. If the bit is set, it means we are dealing with Point AA, Line AA or sprite and in this case the vertex values are going to generated between 0 and 1.

Param_Gen_l0 disable, snd_xy disable, no gen_st – l0 = No modification
Param_Gen_l0 disable, snd_xy disable, gen_st – l0 = No modification
Param_Gen_l0 disable, snd_xy enable, no gen_st – l0 = No modification
Param_Gen_l0 disable, snd_xy enable, gen_st – l0 = No modification
Param_Gen_l0 enable, snd_xy disable, no gen_st – l0 = garbage, garbage, garbage, faceness
Param_Gen_l0 enable, snd_xy disable, gen_st – l0 = garbage, garbage, s, t
Param_Gen_l0 enable, snd_xy enable, no gen_st – l0 = screen x, screen y, garbage, faceness
Param_Gen_l0 enable, snd_xy enable, gen_st – l0 = screen x, screen y, s, t

22.319.3 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 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 state change is detected, 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.

22.3.119.3.1 *Vertex shaders*

In the case of vertex shaders, if GEN_INDEX 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).

22.3.219.3.2 *Pixel shaders*

In the case of pixel shaders, if GEN_INDEX is set and Param_Gen_l0 is enabled, the data will be put in the x field of the 2nd register (R1.x), else if GEN_INDEX is set the data will be put into the x field of the 1st register (R0.x).

Formatted: Bullets and Numbering
Formatted: Bullets and Numbering
Formatted: Bullets and Numbering
Formatted: Bullets and Numbering
Formatted: Bullets and Numbering

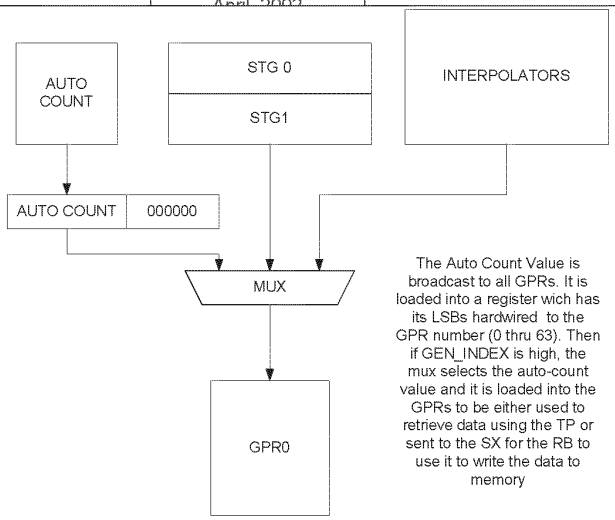


Figure 1213: GPR input mux Control

23.20. 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.

23.120.1 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.

24.21. 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 19.222.2 for details on how to control the interpolation in this mode.

24.121.1 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

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

25.22. Registers

25.122.1 Control

REG_DYNAMIC	Dynamic allocation (pixel/vertex) of the register file on or off.
REG_SIZE_PIX	Size of the register file's pixel portion (minimal size when dynamic allocation turned on)
REG_SIZE_VTX	Size of the register file's vertex portion (minimal size when dynamic allocation turned on)
ARBITRATION_POLICY	policy of the arbitration between vertexes and pixels
INST_BASE_VTX	start point for the vertex instruction store (RT always ends at vertex_base and Begins at 0)
INST_BASE_PIX	start point for the pixel shader instruction store
ONE_THREAD	debug state register. Only allows one program at a time into the GPRs
ONE_ALU	debug state register. Only allows one ALU program at a time to be executed (instead of 2)
INSTRUCTION	This is where the CP puts the base address of the instruction writes and type (auto-incremented on reads/writes) Register mapped
CONSTANTS	512*4 ALU constants + 32*6 Texture state 32 bits registers (logically mapped)
CONSTANTS_RT	256*4 ALU constants + 32*6 texture states? (physically mapped)
CONSTANT_EO_RT	This is the size of the space reserved for real time in the constant store (from 0 to CONSTANT_EO_RT). The re-mapping table operates on the rest of the memory
TSTATE_EO_RT	This is the size of the space reserved for real time in the fetch state store (from 0 to TSTATE_EO_RT). The re-mapping table operates on the rest of the memory

~~EXPORT_LATEControls whether or not we are exporting position from clause 3. If set, position exports occur at clause 7.~~

25.222.2 Context

~~VS_FETCH_{0...7}~~	~~eight 8 bit pointers to the location where each clauses control program is located~~
~~VS_ALU_{0...7}~~	~~eight 8 bit pointers to the location where each clauses control program is located~~
~~PS_FETCH_{0...7}~~	~~eight 8 bit pointers to the location where each clauses control program is located~~
~~PS_ALU_{0...7}~~	~~eight 8 bit pointers to the location where each clauses control program is located~~
PS_BASE	base pointer for the pixel shader in the instruction store
VS_BASE	base pointer for the vertex shader in the instruction store
VS_CF_SIZE	size of the vertex shader (# of instructions in control program/2)
PS_CF_SIZE	size of the pixel shader (# of instructions in control program/2)
PS_SIZE	size of the pixel shader (cntl+instructions)
VS_SIZE	size of the vertex shader (cntl+instructions)
PS_NUM_REG	number of GPRs to allocate for pixel shader programs
VS_NUM_REG	number of GPRs to allocate for vertex shader programs
PARAM_SHADE	One 16 bit register specifying which parameters are to be gouraud shaded (0 = flat, 1 = gouraud)
~~PROVO_VERT~~	~~0 : vertex 0, 1: vertex 1, 2: vertex 2, 3: Last vertex of the primitive~~
PARAM_WRAP	64 bits: for which parameters (and channels (xyzw)) do we do the cyl wrapping (0=linear, 1=cylindrical).
PS_EXPORT_MODE	0xxxx : Normal mode 1xxxx : Multipass mode If normal, bbbz where bbb is how many colors (0-4) and z is export z or not If multipass 1-12 exports for color.
VS_EXPORT_MODE	0: position (1 vector), 1: position (2 vectors), 3:multipass
VS_EXPORT _COUNT	Number of locations exported by the VS (and thus number of interpolated parameters) ~~{0...6} Six 4 bit counters representing the # of interpolated parameters exported in clause 7 (located in VS_EXPORT_COUNT_6) OR # of exported vectors to memory per clause in multipass mode (per clause)~~
PARAM_GEN_I0	Do we overwrite or not the parameter 0 with XY data and generated T and S values

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

GEN_INDEX	Auto generates an address from 0 to XX. Puts the results into R0-1 for pixel shaders and R2 for vertex shaders
CONST_BASE_VTX (9 bits)	Logical Base address for the constants of the Vertex shader
CONST_BASE_PIX (9 bits)	Logical Base address for the constants of the Pixel shader
CONST_SIZE_PIX (8 bits)	Size of the logical constant store for pixel shaders
CONST_SIZE_VTX (8 bits)	Size of the logical constant store for vertex shaders
INST_PRED_OPTIMIZE	Turns on the predicate bit optimization (if of, conditional_execute_predicates is always executed).
CF_BOOLEANS	256 boolean bits
CF_LOOP_COUNT	32x8 bit counters (number of times we traverse the loop)
CF_LOOP_START	32x8 bit counters (init value used in index computation)
CF_LOOP_STEP	32x8 bit counters (step value used in index computation)

~~26.~~23. DEBUG Registers

~~26.1~~23.1 Context

DB_PROB_ADDR	instruction address where the first problem occurred
DB_PROB_COUNT	number of problems encountered during the execution of the program
DB_PROB_BREAK	break the clause if an error is found.
DB_ON	turns on an off debug method 2
DB_INST_COUNT	instruction counter for debug method 2
DB_BREAK_ADDR	break address for method number 2
~~DB_CLAUSE_MODE_ALU_{0…7}~~	~~clause mode for debug method 2 (0: normal, 1: addr, 2: kill)~~
~~DB_CLAUSE_MODE_FETCH_{0…7}~~	~~clause mode for debug method 2 (0: normal, 1: addr, 2: kill)~~

~~26.2~~23.2 Control

DB_ALUCST_MEMSIZE	Size of the physical ALU constant memory
DB_TSTATE_MEMSIZE	Size of the physical texture state memory

~~27.~~24. Interfaces

~~27.1~~24.1 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.

~~27.2~~24.2 SC to SP Interfaces

~~27.2.1~~24.2.1 *~~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
- Ref Pix J => S4.20 Floating Point J value
- Delta Pix I (x3) => S4.8 Floating Point Delta I value
- Delta Pix J (x3) => S4.8 Floating Point Delta J value

This equates to a total of 128 bits which transferred over 2 clocks
and therefor needs an interface 64 bits wide

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

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	64	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 SE4M8 SE4M8 SE4M8 **Type 2** Field Face X Y Bits [63] [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.

~~27.2.2~~24.2.2 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 92 bits) could be folded in half to approx 47 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	[2:1]	2	This field identifies the event 0 => denotes an End Of State Event 1 => TBD
SC_SQ_pc_dealloc	[5:3]	3	Deallocation token for the Parameter Cache
SC_SQ_new_vector	6	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	[10:7]	4	Quad Write mask left to right SP0 => SP3
SC_SQ_end_of_prim	11	1	End Of the primitive
SC_SQ_state_id	[14:12]	3	State/constant pointer (6*3+3)
SC_SQ_pix_mask	[30:15]	16	Valid bits for all pixels SP0=>SP3 (UL,UR,LL,LR)
SC_SQ_prim_type	[33:31]	3	Stippled line and Real time command need to load tex cords from alternate buffer 000: Normal 010: Realtime 101: Line AA 110: Point AA (Sprite)
SC_SQ_provok_vtx	[35:34]	2	Provoking vertex for flat shading
SC_SQ_pc_ptr0	[46:36]	11	Parameter Cache pointer for vertex 0
2nd Clock Transfer			
SC_SQ_pc_ptr1	[10:0]	11	Parameter Cache pointer for vertex 1
SC_SQ_pc_ptr2	[21:11]	11	Parameter Cache pointer for vertex 2
SC_SQ_lod_correct	[45:22]	24	LOD correction per quad (6 bits per quad)

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.)

~~27.2.3~~24.2.3 SQ to SX: Interpolator bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SXx_interp_flat_vtx	SQ→SPx	2	Provoking vertex for flat shading
SQ_SXx_interp_flat_gouraud	SQ→SPx	1	Flat or gouraud shading
SQ_SXx_interp_cyl_wrap	SQ→SPx	4	Wich channel needs to be cylindrical wrapped
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
SQ_SXx_pc_wr_en	SQ→SXx	1	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

~~27.2.4~~24.2.4 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_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

~~27.2.5~~24.2.5 VGT to SQ : Vertex interface

Formatted: Bullets and Numbering

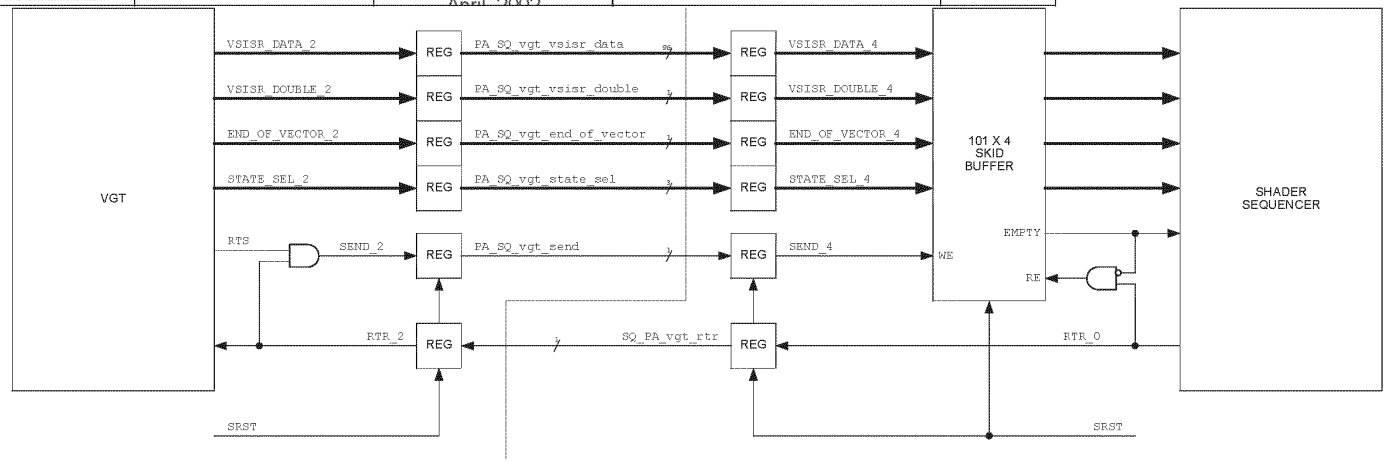
~~27.2.5.1~~24.2.5.1 Interface Signal Table

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.

Name	Bits	Description
VGT_SQ_vsisr_data	96	Pointers of indexes or HOS surface information
VGT_SQ_vsisr_double	1	0: Normal 96 bits per vert 1: double 192 bits per vert
VGT_SQ_end_of_vector	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)

~~27.2.5.2~~24.2.5.2 Interface Diagrams

Formatted: Bullets and Numbering



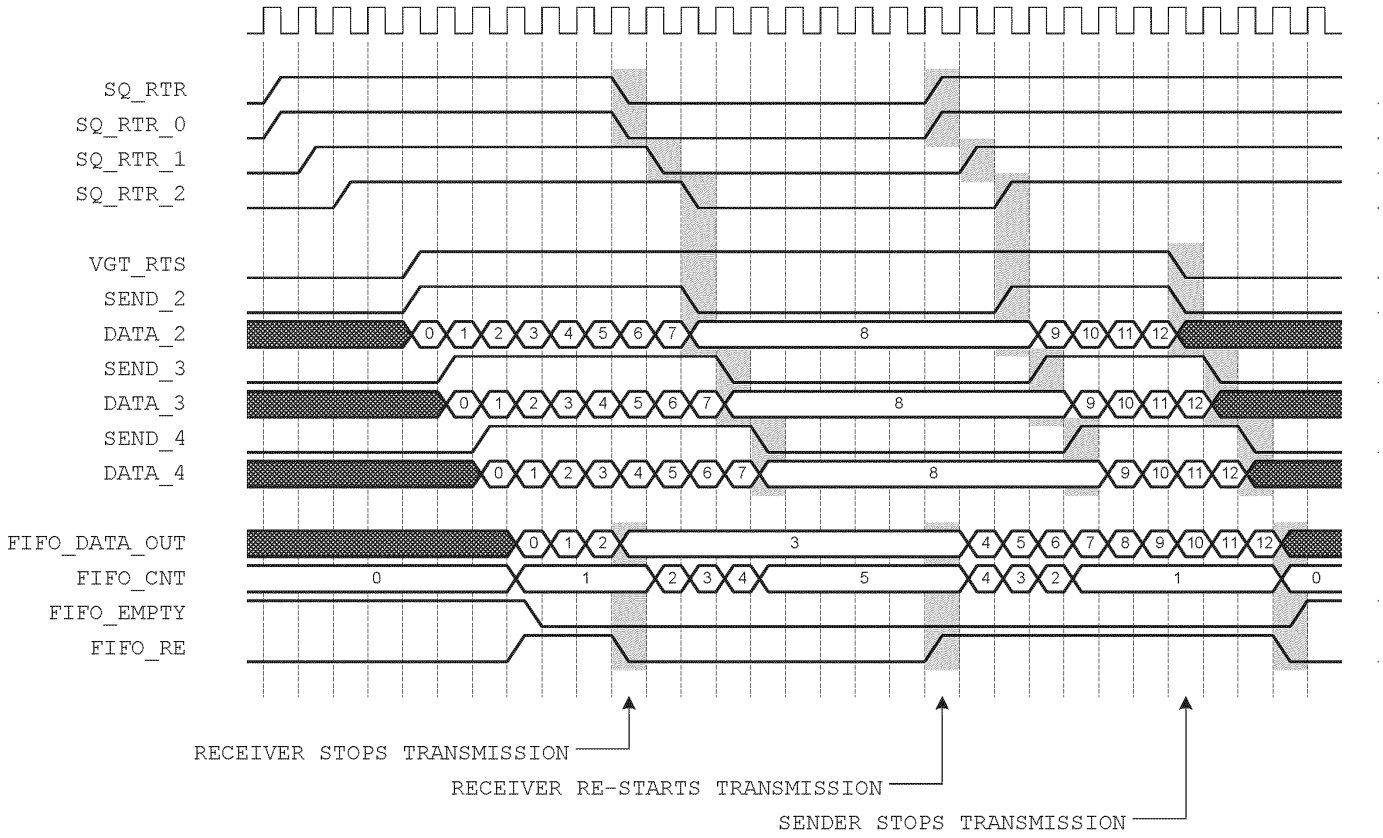


Figure 1. Detailed Logical Diagram for PA_SQ_vgt Interface.

~~27.2.6~~24.2.6 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 to indicate that the previous export is finished (this can be sent with or without the other fields of the interface)
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.~~These fields are sent every time the sequencer picks an exporting clause for execution.~~

~~27.2.7~~24.2.7 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	1	Specifies whether there is room for another position.
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

Formatted: Bullets and Numbering
Formatted
Formatted
Formatted
Formatted
Formatted
Formatted
Formatted
Formatted: Bullets and Numbering
Formatted: Bullets and Numbering

27.2.824.2.8 SQ to TP: Control bus

Once every clock, the fetch unit sends to the sequencer on which clause 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 counters flags for the reservation station fifos. 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 Name	DirectionDirection	BitsBits	DescriptionDescription
TPx_SQ_data_rdyTPx_SQ_data_rdy	TPx→ SQTPx→ SQ	11	Data readyData ready
TPx_SQ_rs_line_numTPx_SQ_clause_num	TPx→ SQTPx→ SQ	63	Line number in the Reservation stationClause number
TPx_SQ_typeTPx_SQ_type	TPx→ SQTPx→ SQ	11	Type of data sent (0:PIXEL, 1:VERTEX)Type of data sent (0:PIXEL, 1:VERTEX)
SQ_TPx_sendSQ_TPx_send	SQ→TPxSQ→TPx	11	Sending valid dataSending valid data
SQ_TPx_constSQ_TPx_const	SQ→TPxSQ→TPx	4848	Fetch state sent over 4 clocks (192 bits total)Fetch state sent over 4 clocks (192 bits total)
SQ_TPx_instrSQ_TPx_instr	SQ→TPxSQ→TPx	2424	Fetch instruction sent over 4 clocksFetch instruction sent over 4 clocks
SQ_TPx_end_of_groupSQ_TPx_end_of_clause	SQ→TPxSQ→TPx	11	Last instruction of the groupLast instruction of the clause
SQ_TPx_TypeSQ_TPx_Type	SQ→TPxSQ→TPx	11	Type of data sent (0:PIXEL, 1:VERTEX)Type of data sent (0:PIXEL, 1:VERTEX)
SQ_TPx_gpr_phaseSQ_TPx_gpr_phase	SQ→TPxSQ→TPx	22	Write phase signalWrite phase signal
SQ_TP0_lod_correctSQ_TP0_lod_correct	SQ→TP0SQ→TP0	66	LOD correct 3 bits per comp 2 components per quad LOD correct 3 bits per comp 2 components per quad
SQ_TP0_pix_maskSQ_TP0_pix_mask	SQ→TP0SQ→TP0	44	Pixel mask 1 bit per pixelPixel mask 1 bit per pixel
SQ_TP1_lod_correctSQ_TP1_lod_correct	SQ→TP1SQ→TP1	66	LOD correct 3 bits per comp 2 components per quad LOD correct 3 bits per comp 2 components per quad
SQ_TP1_pix_maskSQ_TP1_pix_mask	SQ→TP1SQ→TP1	44	Pixel mask 1 bit per pixelPixel mask 1 bit per pixel
SQ_TP2_lod_correctSQ_TP2_lod_correct	SQ→TP2SQ→TP2	66	LOD correct 3 bits per comp 2 components per quad LOD correct 3 bits per comp 2 components per quad
SQ_TP2_pix_maskSQ_TP2_pix_mask	SQ→TP2SQ→TP2	44	Pixel mask 1 bit per pixelPixel mask 1 bit per pixel
SQ_TP3_lod_correctSQ_TP3_lod_correct	SQ→TP3SQ→TP3	66	LOD correct 3 bits per comp 2 components per quad LOD correct 3 bits per comp 2 components per quad
SQ_TP3_pix_maskSQ_TP3_pix_mask	SQ→TP3SQ→TP3	44	Pixel mask 1 bit per pixelPixel mask 1 bit per pixel

Formatted: Bullets and Numbering

Formatted

Formatted

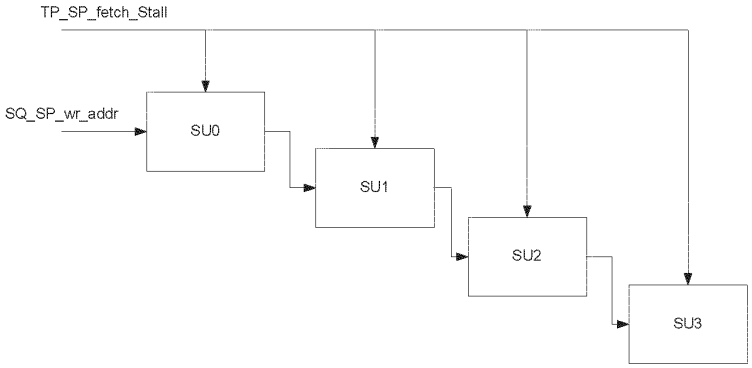
SQ_TPx_rs_line_numSQ_TPx_clause_num	SQ→TPxSQ→TPx	63	Line number in the Reservation stationClause number
SQ_TPx_write_gpr_indexSQ_TPx_write_gpr_index	SQ->TPxSQ->TPx	77	Index into Register file for write of returned Fetch DataIndex into Register file for write of returned Fetch Data

Formatted

27.2.924.2.9 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

27.2.1024.2.10 SQ to SP: Texture stall

Formatted: Bullets and Numbering

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

27.2.1124.2.11 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_SPx_gpr_wr_en	SQ→SPx	1	Write Enable for the GPRs
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	12?	Auto count generated by the SQ, common for all shader pipes

Formatted: Bullets and Numbering

~~27.2.12~~24.2.12 SQ to SPx: Instructions

Name	Direction	Bits	Description
SQ_SPx_instr_start	SQ→SPx	1	Instruction start
SQ_SP_instr	SQ→SPx	21	Transferred over 4 cycles 0: SRC A Select 2:0 SRC A Argument Modifier 3:3 SRC A swizzle 11:4 VectorDst 17:12 Unused 20:18 --- 1: SRC B Select 2:0 SRC B Argument Modifier 3:3 SRC B swizzle 11:4 ScalarDst 17:12 Unused 20:18 --- 2: SRC C Select 2:0 SRC C Argument Modifier 3:3 SRC C swizzle 11:4 Unused 20:12 --- 3: Vector Opcode 4:0 Scalar Opcode 10:5 Vector Clamp 11:11 Scalar Clamp 12:12 Vector Write Mask 16:13 Scalar Write Mask 20:17
SQ_SPx_exp_alu_id	SQ→SPx	1	ALU ID
SQ_SPx_exporting	SQ→SPx	2	0: Not Exporting 1: Vector Exporting 2: Scalar Exporting
SQ_SPx_stall	SQ→SPx	1	Stall signal
SQ_SP0_write_mask	SQ→SP0	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP1_ write_mask	SQ→SP1	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP2_ write_mask	SQ→SP2	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP3_ write_mask	SQ→SP3	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock

Formatted: Bullets and Numbering

~~27.2.13~~24.2.13 SP to SQ: Constant address load/ Predicate Set

Name	Direction	Bits	Description
SP0_SQ_const_addr	SP0→SQ	36	Constant address load / predicate 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) 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) 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) to the sequencer
SP3_SQ_valid	SP3→SQ	1	Data valid

~~27.2.14~~24.2.14 SQ to SPx: constant broadcast

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_const	SQ→SPx	128	Constant broadcast

~~27.2.15~~24.2.15 SP0 to SQ: Kill vector load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SP0_SQ_kill_vect	SP0→SQ	4	Kill vector load
SP1_SQ_kill_vect	SP1→SQ	4	Kill vector load
SP2_SQ_kill_vect	SP2→SQ	4	Kill vector load
SP3_SQ_kill_vect	SP3→SQ	4	Kill vector load

~~27.2.16~~24.2.16 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)

~~27.2.17~~24.2.17 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

~~27.2.18~~24.2.18 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	2	Vertex Shader Event ID
SQ_CP_ps_event	SQ→CP	1	Pixel Shader Event
SQ_CP_ps_eventid	SQ→CP	2	Pixel Shader Event ID

eventid = 0 => *sEndOfState (i.e. VsEndOfState)
eventid = 1 => *sDone (i.e. VsDone)

So, the CP will assume the Vs is done with a state whenever it gets a pulse on the SQ_CP_vs_event and the SQ_CP_vs_eventid = 0.

Formatted: Bullets and Numbering

24.3 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 Alu 0 Alu 0 Tex 0 Tex 0 Alu 1 Alu 0 Tex 0 Alu 0 Alu 1 Tex 0
Execute Alu 0
Alloc Position 1
Execute Alu 0 Tex 0
Alloc Param 3
Execute Alu 0 Tex 0 Alu 1 Alu 0 End
```

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 Alu 0 Alu 0 Tex 0 Tex 0 Alu 1 Alu 0 Tex 0 Alu 0 Alu 1 Tex 0
```

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 Alu 0
```

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 Alu 0 Tex 0
```

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 Alu 0 Tex 0 Alu 1 Alu 0 End
```

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.

~~28.~~25. 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?

Formatted: Bullets and Numbering