	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 2015~~16~~ ~~November 20017~~	GEN-CXXXXX-REVA	1 of 35

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SEQ

Version 1.2~~1~~

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\arch\doc\gfx\RE\R400_Sequencer.doc
Current Intranet Search Title: R400 Sequencer Specification

APPROVALS

Name/Dept	Signature/Date

Remarks:

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

Table Of Contents

Revision Changes:

Rev 0.1 (Laurent Lefebvre) Date: May 7, 2001	First draft.
Rev 0.2 (Laurent Lefebvre) Date : July 9, 2001	Changed the interfaces to reflect the changes in the SP. Added some details in the arbitration section.
Rev 0.3 (Laurent Lefebvre) Date : August 6, 2001	Reviewed the Sequencer spec after the meeting on August 3, 2001.
Rev 0.4 (Laurent Lefebvre) Date : August 24, 2001	Added the dynamic allocation method for register file and an example (written in part by Vic) of the

	flow of pixels/vertices in the sequencer.
Rev 0.5 (Laurent Lefebvre) Date : September 7, 2001	Added timing diagrams (Vic)
Rev 0.6 (Laurent Lefebvre) Date : September 24, 2001	Changed the spec to reflect the new R400 architecture. Added interfaces.
Rev 0.7 (Laurent Lefebvre) Date : October 5, 2001	Added constant store management, instruction store management, control flow management and data dependant predication.
Rev 0.8 (Laurent Lefebvre) Date : October 8, 2001	Changed the control flow method to be more flexible. Also updated the external interfaces.
Rev 0.9 (Laurent Lefebvre) Date : October 17, 2001	Incorporated changes made in the 10/18/01 control flow meeting. Added a NOP instruction, removed the conditional_execute_or_jump. Added debug registers.
Rev 1.0 (Laurent Lefebvre) Date : October 19, 2001	Refined interfaces to RB. Added state registers.
Rev 1.1 (Laurent Lefebvre) Date : October 26, 2001	Added SEQ→SP0 interfaces. Changed delta precision. Changed VGT→SP0 interface. Debug Methods added.
Rev 1.2 (Laurent Lefebvre) Date : November 16, 2001	Interfaces greatly refined. Cleaned up the spec.

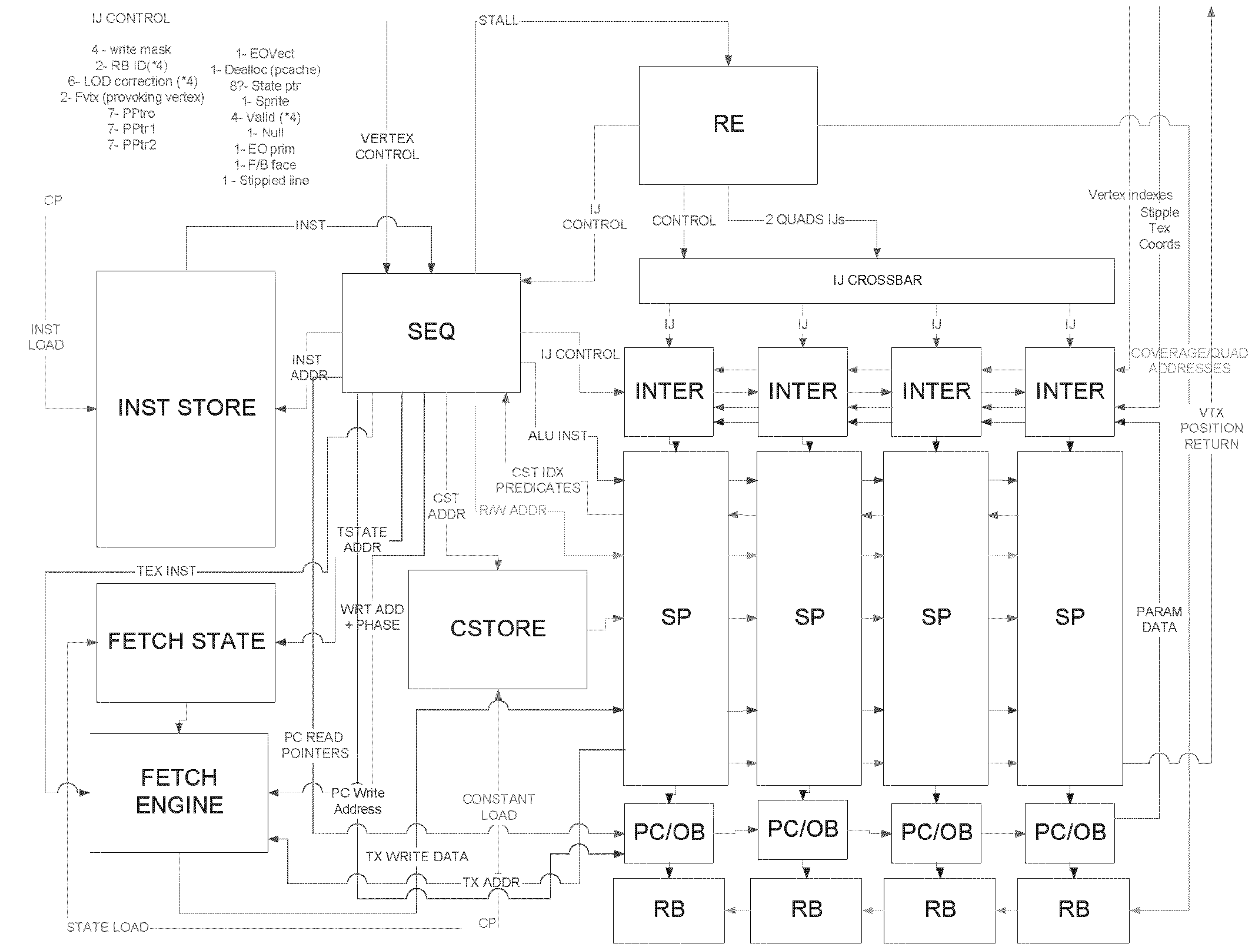
1. Overview

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 raster engine.

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

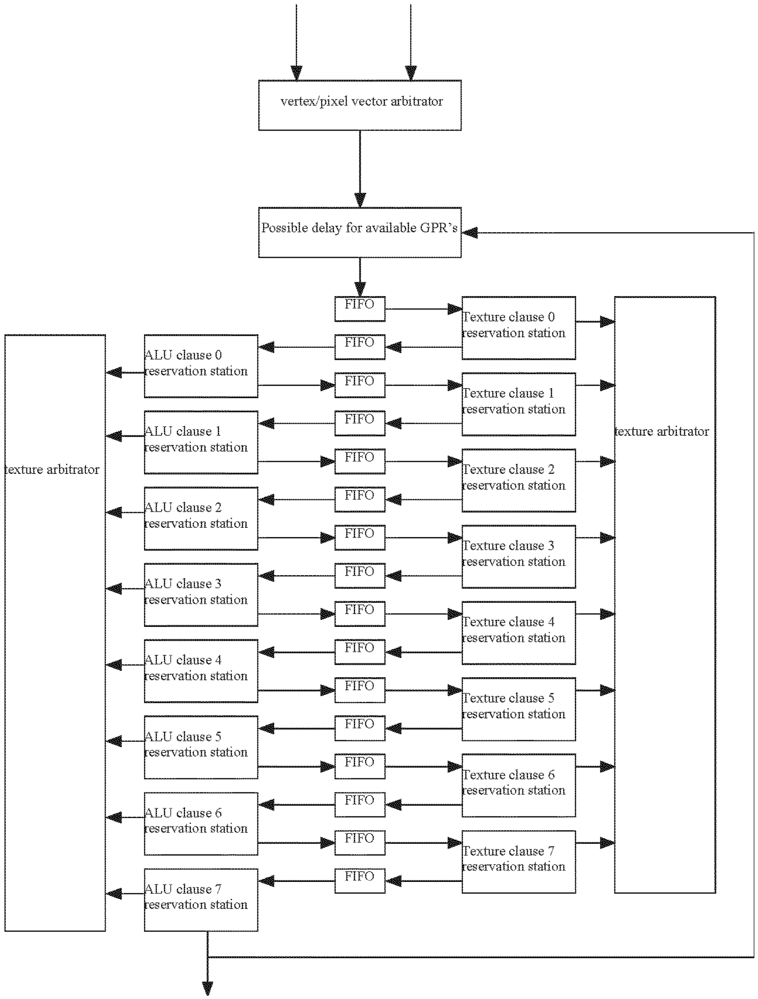
The sequencer is based on the R300 design. It chooses two ALU clauses and a fetch clause to execute, and executes all of the instructions in a clause before looking for a new clause of the same type. Two ALU clauses are executed interleaved to hide the ALU latency. Each vector will have eight fetch and 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. 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 raster engine also contains the shader instruction cache and constant store. There are only one constant store for the whole chip and one instruction store. These will be shared among the four shader pipes. The four shader pipes also execute the same ~~instuction~~instruction thus there is only one sequencer for the whole chip.



IJ CONTROL
4 - write mask
2- RB ID(*4)
6- LOD correction (*4)
2- Fvtx (provoking vertex)
7- PPtro
7- PPtr1
7- PPtr2
1- EOVect
1- Dealloc (pcache)
8?- State ptr
1- Sprite
4- Valid (*4)
1- Null
1- EO prim
1- F/B face
1 - Stippled line
VERTEX CONTROL
STALL
RE
IJ CONTROL
CONTROL
2 QUADS IJs
IJ CROSSBAR
Vertex indexes
Stipple
Tex
Coords
CP
INST LOAD
INST
SEQ
INST ADDR
INST STORE
IJ CONTROL
IJ
IJ
IJ
IJ
INTER
INTER
INTER
INTER
COVERAGE/QUAD ADDRESSES
VTX POSITION RETURN
ALU INST
CST IDX PREDICATES
R/W ADDR
CST ADDR
TSTATE ADDR
TEX INST
SP
SP
SP
SP
PARAM DATA
FETCH STATE
WRT ADD + PHASE
CSTORE
FETCH ENGINE
PC READ POINTERS
PC Write Address
CONSTANT LOAD
TX WRITE DATA
TX ADDR
PC/OB
PC/OB
PC/OB
PC/OB
RB
RB
RB
RB
STATE LOAD
CP

1.1 Top Level Block Diagram


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

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 2~~1~~3 bits of state, 6-7 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 registers to store the interpolated values and temporaries. Following this, the input state machine stacks the packet in the first FIFO.

On receipt of a command, the level 0 fetch machine issues a ~~texure~~texture request and corresponding register 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 register 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 recept 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 1 that the data is ready to be processed.

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

There will always be two active ALU clauses at any given time (and two ~~arbitrers~~arbiters). One ~~arbitrer~~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 ~~arbitrers~~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 ~~arbitrer~~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, the location where the vertex data is to be put is also sent (parameter data pointers).

{ISSUE: How do we handle parameter cache pointers (computed, semi-computed or not computed)?}

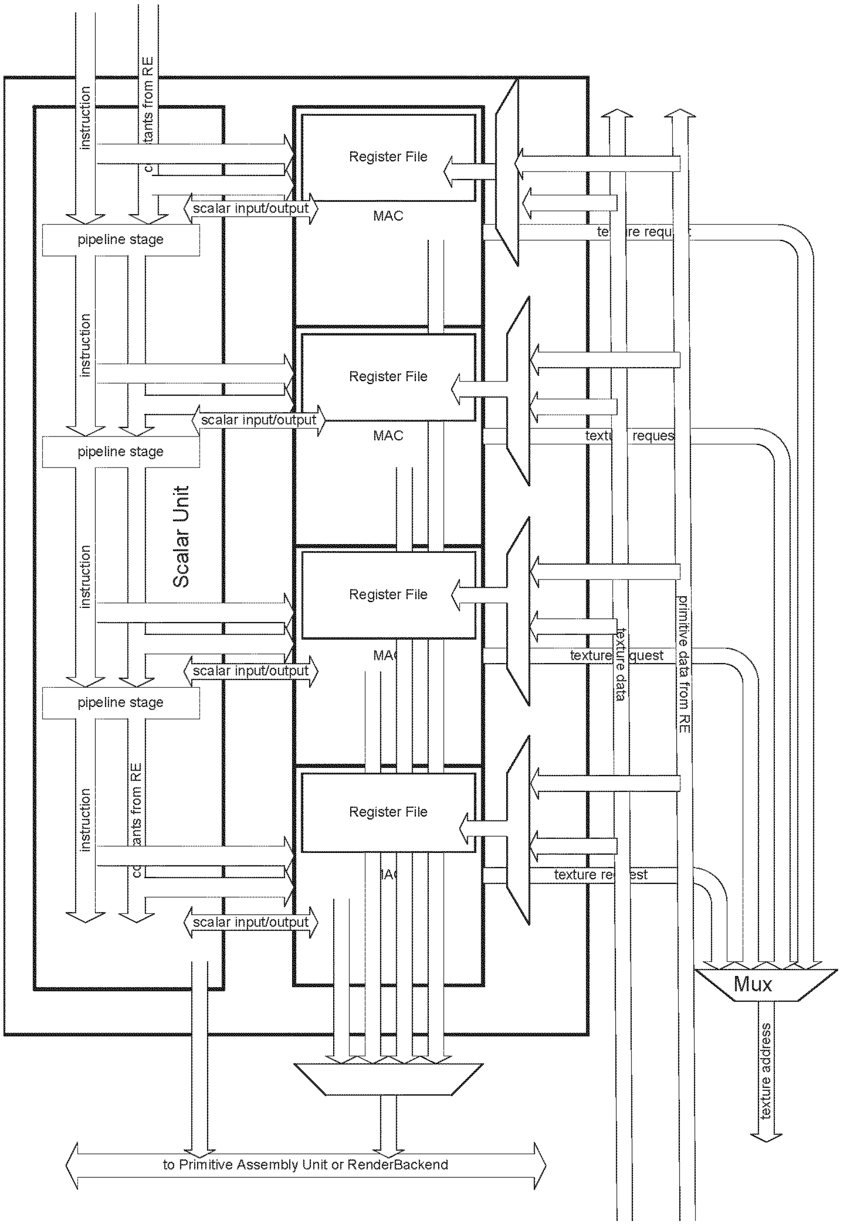
A special case is for HOS surfaces wich can export 12 parameters per last 6 clauses to the output buffer. If the output buffer is full or doesn't have enough space the sequencer will prevent such a vertex group to enter an exporting clause.

Regular pixel and vertex shaders can export 12 parameters to memory from the last clause only (7).

All other level process in the same way until the packet finally reaches the last ALU machine (7). On completion of the level 7 ALU clause, a valid bit is sent to the Render Backend which picks up the color data. This requires that the last instruction writes to the output register – a condition that is almost always true. If the packet was a vertex packet, instead of sending the valid bit to the RB, it is sent to the PA so it can know that the data present in the parameter store is valid.

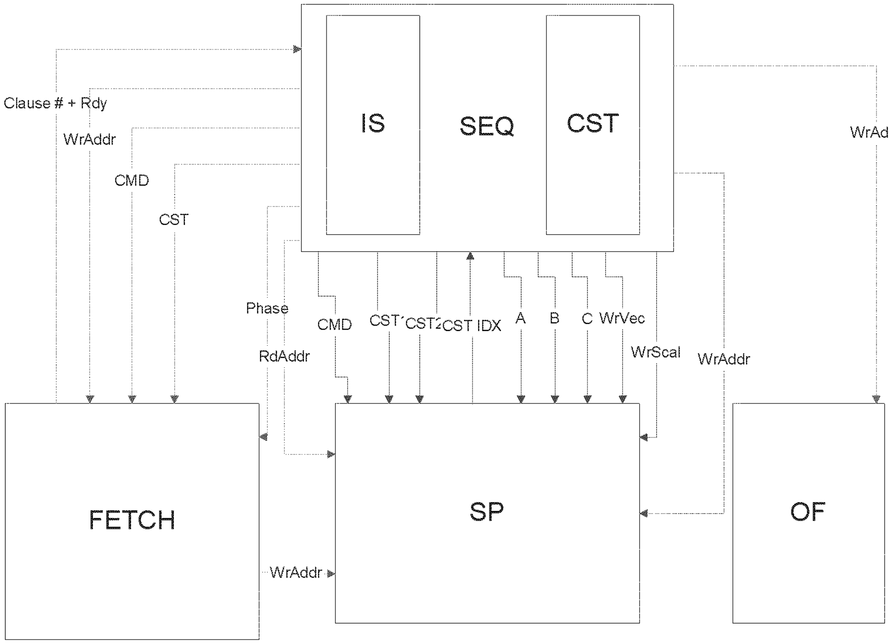
Only two ALU state machine may have access to the register file address bus or the instruction decode bus at one time. Similarly, only one fetch state machine may have access to the register file address bus at one time. Arbitration is performed by three ~~arbitrer~~arbiter blocks (two for the ALU state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

1.2 Data Flow graph



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

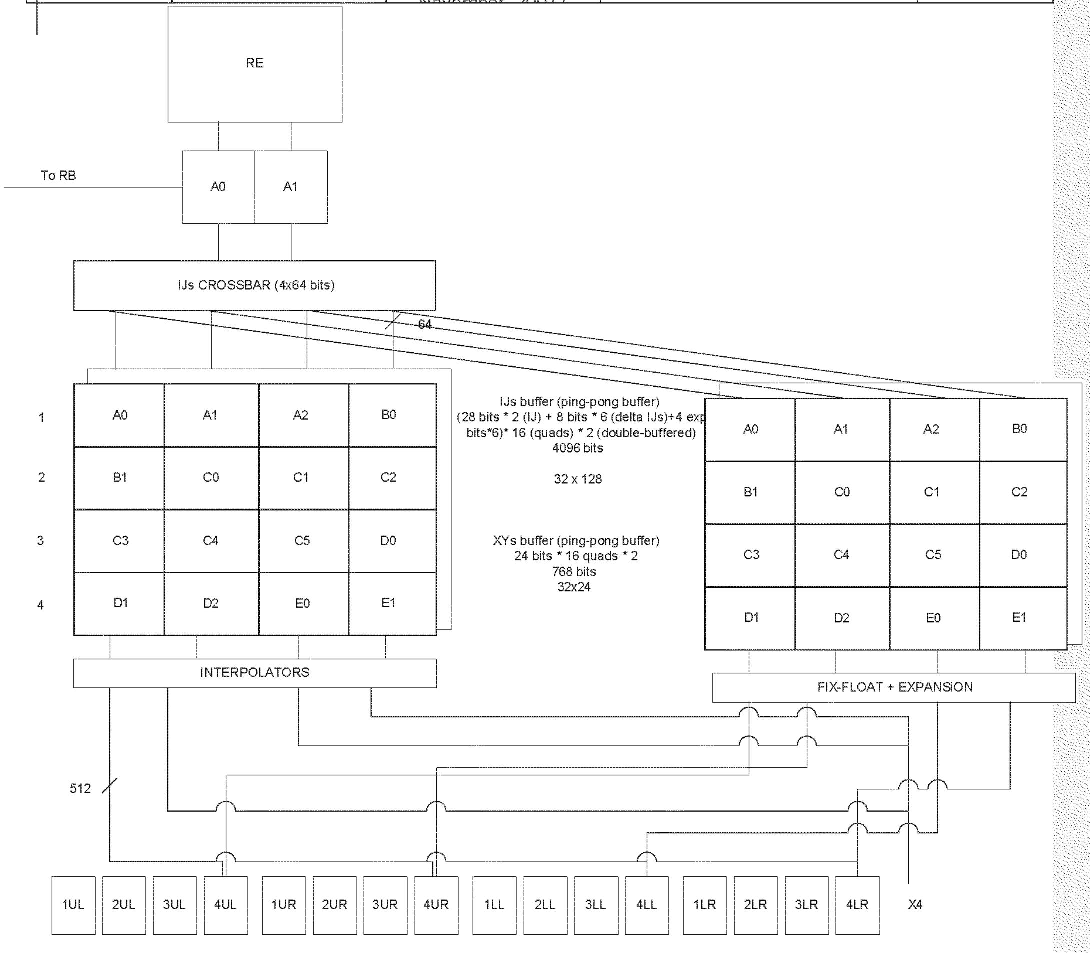
1.3 Control Graph



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.


RE
To RB
A0
A1
IJs CROSSBAR (4x64 bits)
64
1
2
3
4
A0
A1
A2
B0
B1
C0
C1
C2
C3
C4
C5
D0
D1
D2
E0
E1
IJs buffer (ping-pong buffer)
(28 bits * 2 (IJ) + 8 bits * 6 (delta IJs)+4 exp
bits*6)* 16 (quads) * 2 (double-buffered)
4096 bits
32 x 128
XYs buffer (ping-pong buffer)
24 bits * 16 quads * 2
768 bits
32x24
INTERPOLATORS
FIX-FLOAT + EXPANSION
512
1UL
2UL
3UL
4UL
1UR
2UR
3UR
4UR
1LL
2LL
3LL
4LL
1LR
2LR
3LR
4LR
X4

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19	T20	T21	T22	T23
SP 0	A0	B1	C3	D1					A0	B1	C3	D1					XY 0-3	XY 16-19	XY 32-35	XY 48-51	V 0-3	V 16-19	V 32-35	V 48-51
SP 1	A1		C4	D2		C0			A1		C4	D2		C0			XY 4-7	XY 20-23	XY 36-39	XY 52-55	V 4-7	V 20-23	V 36-39	V 52-55
SP 2	A2		C5			C1		E0	A2		C5			C1		E0	XY 8-11	XY 24-27	XY 40-43	XY 56-59	V 8-11	V 24-27	V 40-43	V 56-59
SP 3					B0	C2	D0	E1					B0	C2	D0	E1	XY 12-15	XY 28-31	XY 44-47	XY 60-63	V 12-15	V 28-31	V 44-47	V 60-63
	P0								P1								XY				VTX			

Above is an example of a tile we might receive. The IJ information is packed in the IJ buffer 2 quads at a time. 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 register 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. ~~There is also going to be a control instruction store of size 256(512?)x32.~~

~~{ISSUE : The instruction store is loaded by the sequencer using the memory hub ?}.~~

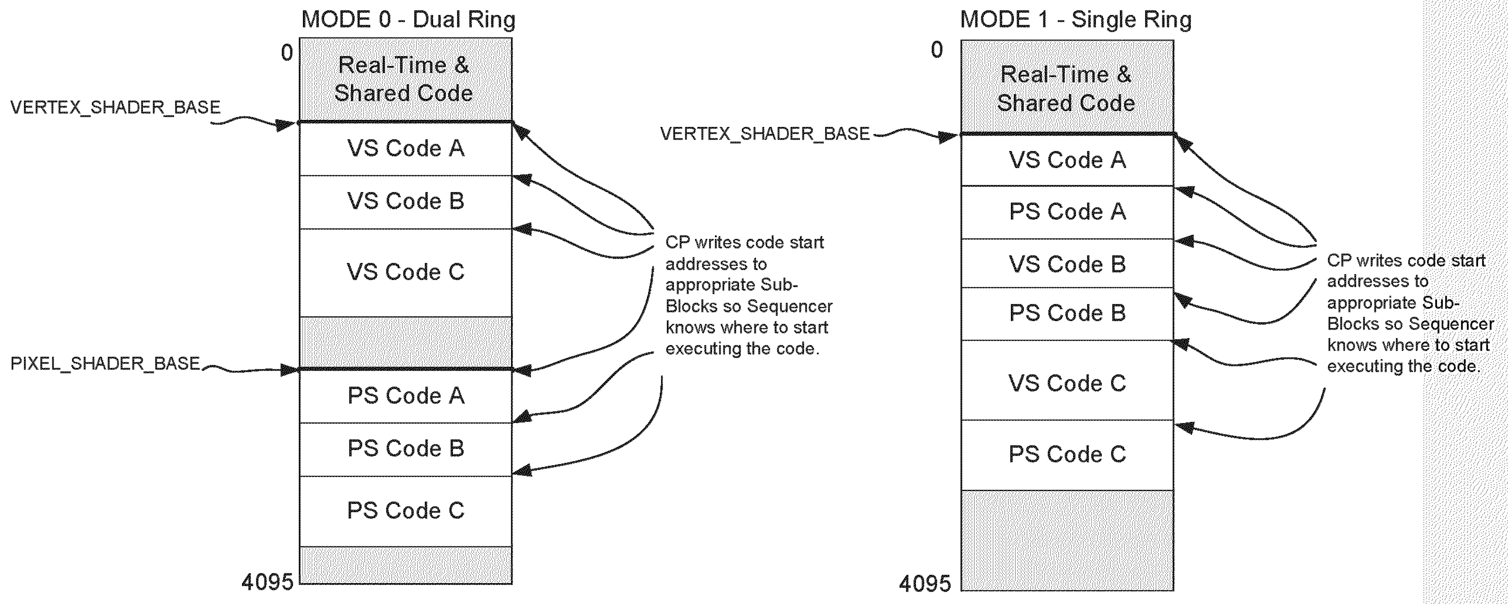
~~The read bandwith from this store is 96*2 bits/ 4 clocks (48 bits/clock).~~ It is likely to be a 1 port memory; we use 1 clock to load the ALU instruction, 1 clocks to load the Fetch instruction, 1 clock to load 2 control flow instructions and 1 clock to write instructions.

The instruction store is loaded by the CP thru the INSTRUCTION_DATA, INSTRUCTION_INDEX_PORT control registers. The INSTRUCTION_INDEX_PORT is auto-incremented on both reads and writes to the INSTRUCTION_DATA register.

The next picture shows the various modes the CP can load the memory. The Sequencer has to keep track of the loading modes in order to wrap around the correct boundaries. The MSB of the INSTRUCTION_INDEX_PORT register contains the packet type for the sequencer to know where it must wrap around. The wrap around points are arbitrary and they are specified in the VERTEX_SHADER_BASE and PIXEL_SHADER_BASE registers.

R400 CP's Views of Instruction Memory

Updated: 11/14/2001
John A. Carey



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).

5. Constant Stores

~~The constant store is managed by the CP.~~ The sequencer is aware of where the constants are using a remaping table~~ also managed by the CP~~. A likely size for the constant store is ~~512x128~~ 1024x128 bits. ~~The constant store is also planned to be shared.~~ The read BW from the constant store is 128 bits/clock and the write bandwith is 32/4 bits/clock.

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 convertion, 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.

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

The texture state is also kept in a similar memory. The size of this memory is 192x128. Which lets us load a texture state in ????

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 state change.

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.

As per Dx9 the following state is available for control flow:

Boolean[15:0]
loop_count[7:0][7:0]
In addition:
loop_start [7:0] [7:0]
loop_step [7:0] [7:0]
Exist to give more control to the controlling program.

We will extend that in the R400 to:
Boolean[255:0]
Loop_count[7:0][15:0]
Loop_Start[7:0] [15:0] times ~~2~~ 3 (one for constant,registert1, register2)
Loop_Step[7:0] [15:0] times ~~2~~ 3 (one for constant, registert1, register2~~register~~)
Loop_End[7:0] [15:0]

{ISSUE: How is the controlling state loaded and how many contexts do we have?}

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

6.2 The Control Flow Program

The R300 uses a match method for control flow: The shader is executed, and at every instruction its address is compared with addresses (or address?) in a control table. The "event" in the control table can redirect operations in the program.

The Method chosen for the R400 is a "control program". The control program has ten basic instructions:

Execute
Conditional_execute
Conditional_Execute_Predicates
Conditional_jump
Call
Return
Loop_start
Loop_end
End_of_clause
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.
Call jumps to an address and pushes the IP counter on the stack. On the return instruction, the IP is poped from the stack.
Conditional_execute_or_Jump executes a block of instructions or jumps to an address is the condition is not met.
Conditional_execute_Predicates executes a block of instructions if all bits in the predicate vectors meet the condition.
End_of_clause marks the end of a clause.
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. 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.

Execute				
47	46... 42	41 ... 24	23 ... 12	11 ... 0
Addressing	00001	RESERVED	Instruction _count	Exec Address

Execute up to 4k instructions at the specified address in the instruction memory.

NOP		
47	46 ... 42	41 ... 0
Addressing	00010	RESERVED

~~If the specified boolean (8 bits can address 256 booleans) meets the specified condition then execute the specified instructions (up to 512 instructions) or if the condition is not met jump to the jump address in the control flow program. This MUST be a forward jump.~~This is a regular NOP.

Conditionnal_Execute						
47	46 … 42	41 … 34	33	32 … 24	23 … 12	11 … 0
Addressing	00011	Boolean address	Condition	RESERVED	Instruction_count	Exec Address

If the specified boolean (8 bits can address 256 booleans) meets the specified condition then execute the specified instructions (up to 4k instructions)

Conditionnal_Execute_Predicates						
47	46 … 42	41 … 38	37	36 … 24	23 … 12	11 … 0
Addressing	00100	Predicate vector	Condition	RESERVED	Instruction_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.

Loop_Start				
47	46 … 42	41 … 16	15 … 4	3 … 0
Addressing	00101	RESERVED	Jump address	Loop ID

Loop Start. Compares the loop count with the end value. If loop condition not met jump to the address. Forward jump only. Also computes the index value.

Loop_End				
47	46 … 42	41 … 16	15 … 4	3 … 0
Addressing	00111	RESERVED	Start address	Loop ID

Loop end. Increments the counter by one and jumps BACK only 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.

Call			
47	46 … 42	41…12	11 … 0
Addressing	01000	RESERVED	Address

Jumps to the specified address and pushes the IP counter on the stack.

Return		
47	46 … 42	41 … 0
Addressing	01001	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 … 42	41 … 34	33	32 … 13	12	11 … 0
Addressing	01010	Boolean address	Condition	RESERVED	FW only	Address

If condition met, jumps to the address. FORWARD jump only allowed if bit 12 set. Bit 12 is only an optimization for the compiler and should NOT be exposed to the API.

End_of_Clause		
47	46 … 42	41 … 0
Addressing	01011	RESERVED

Marks the end of a clause.

To prevent infinite loops, we will keep 9 bits loop counters 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 th~~d~~e debug registers. ~~The sequencer will keep two loop indexes values:~~
~~IC index for constant indexing (9 bits)~~
~~IR index for register file indexing (7 bits)~~
~~This will be updated every time we loop and can only be used to index the constant store and the register file. The way to compute this value is:~~

~~Index = Loop_counter*Loop_iterator + Loop_init.~~

~~The IC for constant is going to return 0 if it is out of the constant range. The IR index is going to break the program if the index exceeds the number of requested registers.~~

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.

Formatted

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.

6.3 Data dependant predicate instructions

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

PRED_SETE_# - similar to SETE except that the result is 'exported' to the sequencer.
PRED_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 HW Detection of PV,PS

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

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

6.5 Register file indexing

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

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

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

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

Index = Loop_counter*Loop_iterator + Loop_init.

The index is going to return 0 if it is out of the range.

6.6 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.**

Formatted

6.7 Debugging the Shaders

In order to be able to debug the pixel/vertex shaders efficiently, we provide ~~3~~2 methods.

6.7.1 *Method 1: Debugging registers*

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

The sequencer will detect the following groups of errors:
- count overflow
- jump error
  relative jump address > size of the control flow program
  relative jump address > length of the shader program

- constant overflow
- register overflow
- call stack
  call with stack full
  return with stack empty

With two of the errors, a jump error or a register overflow will 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 the other errors, program can continue to run, potentially to worst-case limits.

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

{ISSUE : Interrupt to the driver or not?}

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

The sequencer will have a 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 :

1) Normal
2) Debug Kill
3) Debug Addr + Count

~~4)Debug Count~~

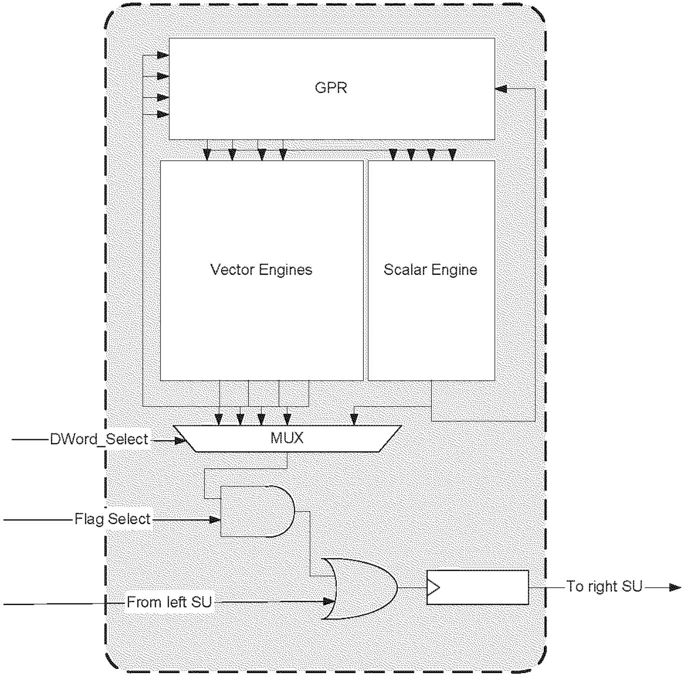
Formatted: Bullets and Numbering

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 ~~two~~ other mode~~s~~, normal execution is done until we reach an address specified by the address register ~~or~~ and instruction count (useful for loops) specified by the count register. ~~After we have hit the address or the count we change to the kill mode for the rest of the clause~~ After we have hit the instruction n times (n=count) we switch the clause to the kill mode.~~.~~

~~6.7.3 *Method 3: Selective export of a 32 bit Dword.*~~

Formatted: Bullets and Numbering

~~The third debug option will be mainly used for HW debug. For this mode, the sequencer will keep the following control debug registers: Shader_pipe (6 bits), Mode(1 bit), Dword_select (3 bits), clause_+count (16 bits?),address (12 bits) Vector_number (8 bits), Render_state (21 bits). The shader pipe register selects a shader pipe amongst the 64, the dword_select selects a channel (0...3 of vector or scalar), the clause_+count selects at which clause and which count we export, the Render_state specifies which render state is concerned, the Vector_number specifies which vector is concerned and the mode selects count export, or address export.~~



~~Flag Select is a combination of Shader_pipe, clause_+count, address, Vector_number and render_state. It is only active for 1 shader pipe at a time and for 1 vector of a given state. The driver is responsible to reset the output register to 0 before executing a given program.~~

7. Pixel Kill Mask

A vector of 64 bits is kept 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:

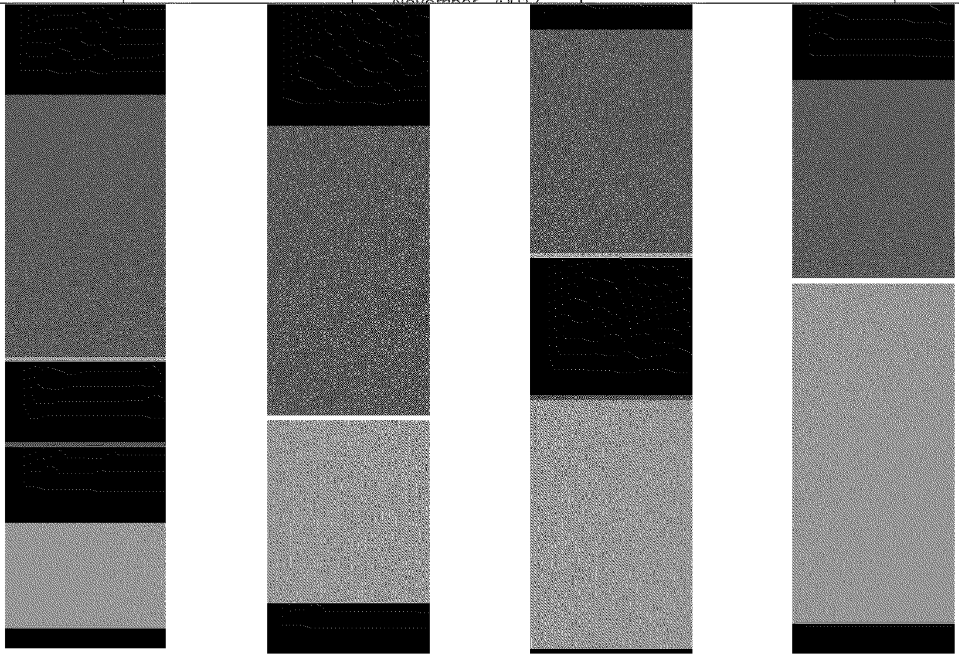
MASK_SETE
MASK_SETGT
MASK_SETGTE

8. HOS surfaces

HOS surfaces are able to export from the 6 last clauses but to memory ONLY. If they want to export to the parameter cache they have to do it in the last clause (7). They can also export position in clause 3.

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 VERTEX_REG_SIZE for vertices and 256-VERTEX_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.

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 arbitrers, one for the even clocks and one for the odd clocks. For exemple, 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.

12. Handling Stalls

When the output file is full, the sequencer prevents the ALU arbitration logic to select 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 to enter the exporting clause (~~4~~3?). The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU ~~arbitrer~~arbiter will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

13. Content of the reservation station FIFOs

21 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, quad address and 1 bit to specify if the vector is of pixels or vertices. Since pixels and vertices are kept in order in the shader pipe, we only need two fifos (one for vertices and one for pixels) deep enough to cover the shader pipe latency. This size will be determined later when we will know the size of the small fifos between the reservation stations.

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. ~~For this reason only ONE concurrent program can be of clause 8 (exporting clause) the other program MUST not.~~ 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 : 19*2 + 8*6 + 4*8 + 4*2 = 128

The Deltas have a leading 1, the Full precision IJs don't. This means that in the case of the deltas we MUST be able to shift 8 right (exponent value of 0 means number = 0, exponent value of 1 means shift right 8). This means that the maximum range for the IJs (Full precision) is +/- 64 and the range for the Deltas is +/- 128.

16. The parameter cache

The parameter cache is where the vertex shaders export their data. It consists of 16 128x128 memories (1R/1W). The reuse engine will make it so that all vertexes of a given primitive will hit different memories.

17. Vertex position exporting

On clause ~~4 (or 5)~~3 the vertex shader can export to the PA both the vertex position and the point sprite. It can also do so at clause ~~8~~ 7 if not done at clause ~~4~~3. ~~Along with the position is exported a pointer to the parameter cache where the data will be once the vertex shader exports.~~ 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.

18. Exporting Arbitration

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

1) Position exports and position exports cannot be co-issued.
2) Position exports and memory exports cannot be co-issued.
3) Position exports and Z/Color exports cannot be co-issued.
4) Memory exports and Z/Color exports cannot be co-issued.
5) Memory exports and memory exports cannot be co-issued.
6) Z/color exports and Z/color exports cannot be co-issued.
7) Parameter exports and Z/Color exports CAN be co-issued.
8) Parameter exports and parameter exports CAN be co-issued.
9) Parameter exports and memory exports CAN be co-issued.

19. 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 realtime 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~~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.

Formatted: Bullets and Numbering

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.

Formatted: Bullets and Numbering

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.

Formatted: Bullets and Numbering

~~20.~~22. Registers

~~20.1~~22.1 Control

DYNAMIC_REG	Dynamic allocation (pixel/vertex) of the register file on or off.
VERTEX_REG_SIZE	What portion of the register file is reserved for vertices (static allocation only)
PIXEL_MIN_SIZE	Minimal size of the register file's pixel portion (dynamic only)
VERTEX_MIN_SIZE	Minimal size of the register file's vertex portion (dynamic only)
ARBITRATION_policy	policy of the arbitration between vertexes and pixels
CST_SIZE_P	Size of the constant store for pixels
CST_SIZE_V	Size of the constant store for vertexes
INST_STOR_ALLOC	interleaved, separate, interleaved+shared,separate+shared
VERTEX_WRAP	start point for the vertex instruction store (RT always ends at vertex_wrap and Begins at 0)
PIXEL_WRAP	start point for the pixel shader instruction store ~~(vertex shader always starts at 0)~~
~~SHAREDWRAP~~	~~start point for the shared instruction store~~
~~RTWRAP~~	~~start point for the RT instruction store (RT always ends at the end of the IM)~~
NO_INTERLEAVE	debug state register. Only allows one program at a time into the GPRs
NO_INTERLEAVE_ALU	debug state register. Only allows one ALU program at a time to be executed (instead of 2)
NO_PRED_OPTIMIZE	turns off the predicate bit optimization (conditional_execute_predicates is always executed.
INSTRUCTION_INDEX_PORT	This is where the CP puts the base address of the instruction writes and type (auto-incremented on reads/writes)
INSTRUCTION_DATA	This is where the CP puts the actual data going to the instruction memory
CONSTANT_DATA	This is where the CP puts constant data

Formatted: Bullets and Numbering

~~20.2~~22.2 Context

Vshader_fetch[7:0][7:0]	eight 8 bit pointers to the location where each clauses control program is located
Vshader_alu[7:0][7:0]	eight 8 bit pointers to the location where each clauses control program is located
Pshader_fetch[7:0][7:0]	eight 8 bit pointers to the location where each clauses control program is located
Pshader_alu[7:0][7:0]	eight 8 bit pointers to the location where each clauses control program is located
PSHADER	base pointer for the pixel shader
VSHADER	base pointer for the vertex shader
Vshader_cntl_size	size of the vertex shader (# of instructions in control program/2)
Pshader_cntl_size	size of the pixel shader (# of instructions in control program/2)
Pshader_size	size of the pixel shader (cntl+instructions)
Vshader_size	size of the vertex shader (cntl+instructions)
REG_ALLOC_PIX	number of registers to allocate for pixel shader programs
REG_ALLOC_VERT	number of registers to allocate for vertex shader programs
FLAT_GOUR[0…15]	which parameters are to be gouraud shaded
CYL_WRAP[0…63]	for which parameters (and channels (xyzw)) do we do the cyl wrapping.
P_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.
vshader_export_mask	which of the last 6 ALU clauses is exporting
vshader_export_mode	0: position (1 vector), 1: position (2 vectors), 3:multipass
vshader_export_count[6]	# of interpolated parameters exported in clause 7 OR # of exported vectors to memory per clause in multipass mode (per clause)
Control_Flow	24 Dwords that contain the control flow constants.~~kill_vector_on use the mask kill vector to kill pixels and optimize texture pipe fetches OR use it as the fifth predicate vector which is the only predicate vector kept across clause boundaries.~~

Formatted: Bullets and Numbering

~~21.~~23. DEBUG registers

~~21.1 Control~~

~~Shader_pipe~~	~~# of the shader pipe for method 3 (0…64)~~
~~Count_+clause~~	~~instruction count and clause number for method 3~~
~~Dword_select~~	~~channel select for method 3~~
~~Mode~~	~~operating mode for method 3~~
~~Rstate~~	~~render state method 3 is operating on~~
~~Vector_count~~	~~vector number the method 3 will export~~

Formatted: Bullets and Numbering

~~21.2~~23.1 Context

PROB_ADDR	instruction address where the first problem occurred
PROB_COUNT	number of problems encountered during the execution of the program
Count	instruction counter for debug method 2
Addr	break address for method number 2
Clause_mode[3]	clause mode for debug method 2

Formatted: Bullets and Numbering

~~22.~~24. Interfaces

~~22.1~~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.

Formatted: Bullets and Numbering

~~22.1.1~~24.1.1 *~~PA/~~SC to SP~~0~~ : IJ bus*

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

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

~~22.1.2~~24.1.2 ~~PA/~~SC to SEQ : IJ Control bus

Formatted: Bullets and Numbering

This is the control information sent to the sequencer in order to control the IJ fifos and all other information needed to execute a shader program on the sent pixels. This information is sent over 2 clocks, if SENDXY is asserted the next control packet is going to be ignored and XY information is going to be sent on the IJ bus (for the quads that where just sent).

Name	Direction	Bits	Description
~~Write Mask~~SC_SQ_q_wr_mask	~~PA~~SC→SEQ~~(SP)~~	4	Quad Write mask left to right
~~LOD_CORRECT~~SC_SQ_lod_correct	SC→SQ~~PA→SEQ(SP)~~	24	LOD correction per quad (6 bits per quad)
~~FVTX~~SC_SQ_flat_vertex	SC→SQ~~PA→SEQ(SP)~~	2	Provoking vertex for flat shading
~~PPTR0~~SC_SQ_param_ptr0	SC→SQ~~PA→SEQ(SP)~~	11	P Store pointer for vertex 0
SC_SQ_param_ptr1~~PPRT1~~	SC→SQ~~PA→SEQ(SP)~~	11	P Store pointer for vertex 1
SC_SQ_param_ptr2~~PPTR2~~	SC→SQ~~PA→SEQ(SP)~~	11	P Store pointer for vertex 2
SC~~E_OFF_VECTOR~~_SQ_end_of_vect	SC→SQ~~PA→SEQ(SP)~~	1	End of the vector
~~DEALLOC~~SC_SQ_store_dealloc	SC→SQ~~PA→SEQ(SP)~~	1	Deallocation token for the P Store
~~STATE~~SC_SQ_state	SC→SQ~~PA→SEQ(SP)~~	~~21~~3	State/constant pointer (6*3+3)
~~VALID~~SC_SQ_valid_pixel	SC→SQ~~PA→SEQ(SP)~~	16	Valid bits for all pixels
~~NULL~~SC_SQ_null_prim	SC→SQ~~PA→SEQ(SP)~~	1	Null Primitive (for PC deallocation purposes)
SC_SQ_end_of_prim~~E_OFF_PRIM~~	SC→SQ~~PA→SEQ(SP)~~	1	End Of the primitive
~~FBFACE~~SC_SQ_fbface	SC→SQ~~PA→SEQ(SP)~~	1	Front face = 1, back face = 0
SC_SQ_send_xy	SC→SQ	1	Sending XY information [XY information is going to be sent on the next clock]
~~TYPE~~SC_SQ_prim_type	SC→SQ~~PA→SEQ(SP)~~	3	Stippled line and Real time command need to load tex cords from alternate buffer 000 : Normal ~~001 : Stippled line~~ 011 : Real Time 100 : Line AA 101 : Point AA 110 : Sprite
SC_SQ_RTRn	~~SEQ→PA~~SQ→SC	1	Stalls the PA in n clocks
SC_SQ_RTS	~~PA~~SC→~~SEQ(SP)~~SQ	1	~~PA~~SC ready to send data

~~22.1.3~~24.1.3 SEQ to SP~~0~~: Interpolator bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
~~TYPE~~SQ_SPx_interp_prim_type	SEQ→SPx~~0~~	3	Type of the primitive 000 : Normal ~~001 : Stippled line/Poly~~ 011 : Real Time 100 : Line AA 101 : Point AA 110 : Sprite
~~FVTX~~SQ_SPx_interp_flat_vtx	SEQ→SP~~0~~x	2	Provoking vertex for flat shading

~~FLAT_GOURAUD~~SQ_SPx_interp_flat_gouraud	S~~E~~Q→SP~~0~~x	1	Flat or gouraud shading
SQ_SPx_interp_cyl_wrap~~CYL_WRAP~~	S~~E~~Q→SP~~0~~x	4	Wich parameter needs to be cylindrical wrapped
SQ_SPx_interp_ijline~~IJ_Line number~~	S~~E~~Q→SPx~~0~~	2	Line in the IJ/XY buffer to use to interpolate
SQ_SPx_interp_buff_swap~~Swap_Buffers~~	S~~E~~Q→SP~~0~~x	1	Swap the IJ/XY buffers at the end of the interpolation
SQ_SPx_interp_ij_xy	SQ→SPx	1	Read from the IJ buffer or from the XY buffer
SQ_SPx_interp_param0~~Param_0~~	S~~E~~Q→SP~~0~~x	1	We are interpolating parameter 0

Formatted: Bullets and Numbering

~~22.1.4~~24.1.4 *S~~E~~Q to SP~~0~~: Parameter Cache Read control bus*

The four following interfaces (SQ→SP, SQ→SX,SP→SX and SX→Interpolators) are all SYNCHRONIZED together.

Name	Direction	Bits	Description
SQ_SPx_~~P~~ptr0~~1~~	S~~E~~Q→SPx~~0~~	~~7~~9	Pointer of PC ~~(7 LSBs of Pointer)~~
SQ_SPx_~~P~~ptr1~~2~~	S~~E~~Q→SPx~~P0~~	~~7~~9	Pointer of PC ~~(7 LSBs of Pointer)~~
SQ_SPx_~~P~~ptr~~3~~2	S~~E~~Q→SPx~~0~~	9~~7~~	Pointer of PC ~~(7 LSBs of Pointer)~~
SQ_SP0_read_ena	SQ→SP0	4	Read enables for the 4 memories in the SP0
SQ_SP1_read_ena	SQ→SP1	4	Read enables for the 4 memories in the SP1
SQ_SP2_read_ena	SQ→SP2	4	Read enables for the 4 memories in the SP2
SQ_SP3_read_ena	SQ→SP3	4	Read enables for the 4 memories in the SP3

Formatted: Bullets and Numbering

~~22.1.5~~24.1.5 *S~~E~~Q to SX~~0~~: Parameter Cache Mux control Bus*

Name	Direction	Bits	Description
SQ_SXx_~~M~~mux0~~1~~	S~~E~~Q→SXx~~0~~	4	Mux control for PC (4 MSbs of Pointer)
SQ_SXx_~~M~~mux~~2~~1	S~~E~~Q→SXx~~0~~	4	Mux control for PC (4 MSbs of Pointer)
SQ_SXx_~~M~~mux~~3~~2	S~~E~~Q→SXx~~0~~	4	Mux control for PC (4 MSbs of Pointer)

Formatted: Bullets and Numbering

24.1.6 *SP to SX: Parameter data*

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

Formatted: Bullets and Numbering

~~22.1.6~~24.1.7 *SX~~0~~ to ~~SP0~~ Interpolators: Parameter Cache Return bus*

Name	Direction	Bits	Description
SXx_SPx_~~V~~vtx_data_~~1~~0	SXx~~0~~→SPx~~0~~	128	Vertex data to interpolate
SXx_SPx_~~V~~vtx_data_~~2~~1	SXx~~0~~→SPx~~0~~	128	Vertex data to interpolate
SXx_SPx_~~V~~vtx_data_~~3~~2	SXx~~0~~→SPx~~0~~	128	Vertex data to interpolate

~~22.1.7~~24.1.8 VGT to SP0~~0~~/SEQ : Vertex Bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
VGT_SP0_vrtx_indexes~~Vertex indexes~~	VGT→SP0	128	Pointers of indexes or HOS surface information
VGT_SP0_end_of_vect~~EOF_vector~~	VGT→SP0~~/SEQ~~	1	End of the vector
VGT_SP0_vrtx_format ~~Inputs_vert~~	VGT→SP0~~/SEQ~~	1	0: Normal 128 bits per vert 1: double 256 bits per vert
VGT_SQ_end_of_vect	VGT→SQ	1	End of the vector
VGT_SQ_vrtx_format	VGT→SQ	1	0: Normal 128 bits per vert 1: double 256 bits per vert
VGT_SQ_stateSTATE	VGT→SEQ	213	Render State (6*3+3 for constants)

~~22.1.8 CP to SEQ : Constant store load~~

Formatted: Bullets and Numbering

~~22.1.9CP to SEQ : Fetch State store load~~

~~22.1.10CP to SEQ : Control State store load~~

Formatted: Bullets and Numbering

~~{ISSUE: How,Who and what is the size of this bus?}~~

~~22.1.11 MH to SEQ: Instruction store Load~~

Formatted: Bullets and Numbering

24.1.9 SEQ to CP: State report

Name	Direction	Bits	Description
SQ_CP_vrtx_state	SEQ→CP	3	Oldest vertex state still in the pipe
SQ_CP_pix_state	SEQ→CP	3	Oldest pixel state still in the pipe

~~{ISSUE: CP or MH?}~~

~~22.1.12~~24.1.10 SP~~0~~ to SX~~0~~ : Pixel/Vertex ~~read from RBs~~write to SX

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SP0_SX0_Export_data	SP0→SX0	~~64*16~~256	~~43~~2 pairs~~s~~ of 32 bits channel values
SP0_SX0_Shader_Dest	SP0→SX0	4	Specifies one of the of up to 12 export destinations
SP1_SX1_Export_data	SP1→SX1	256	4 pairs of 32 bits channel values
SP1_SX1_Shader_Dest	SP1→SX1	4	Specifies one of the of up to 12 export destinations
SP2_SX0_Export_data	SP2→SX0	256	4 pairs of 32 bits channel values
SP2_SX0_Shader_Dest	SP2→SX0	4	Specifies one of the of up to 12 export destinations
SP3_SX1_Export_data	SP3→SX1	256	4 pairs of 32 bits channel values
SP3_SX1_Shader_Dest	SP3→SX1	4	Specifies one of the of up to 12 export destinations
SPx_SXx_Shader_Count	SP0→SX0	3	Each set of four pixels or vectors is exported over eight clocks. This field specifies where the SP is in that sequence.
SPx_SXx_Shader_Last	SP0→SX0	1	~~The current export clause is over (true for one clock)~~ ~~The last export instruction creates *two* cycles to the RB. This needs to be set on or after the last RB cycle that is produced by the last export instruction, but before the first RB cycle of the first export instruction of the next clause.~~Asserted on the first shader count of the last export of the clause
SP0_SX0_Shader_PixelValid	SP0→SX0	4~~x4~~	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock

SP0_SX0_Shader_WordValid	SP0→SX0	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SP1_SX1_Shader_PixelValid	SP1→SX1	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SP1_SX1_Shader_WordValid	SP1→SX1	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SP2_SX0_Shader_PixelValid	SP2→SX0	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SP2_SX0_Shader_WordValid	SP2→SX0	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SP3_SX1_Shader_PixelValid	SP3→SX1	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SP3_SX1_Shader_WordValid	SP3→SX1	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors

~~22.1.13~~24.1.11 *S~~E~~Q to SX~~X0~~ : Control bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SXx_Export_Pixel	S~~E~~Q→SXx~~0~~	1	1: Pixel 0: Vertex
SQ_SXx_Export_SEND	S~~E~~Q→SXx0	1	Raised to indicate that the SQ is starting an export
SQ_SXx_Export_Clause	S~~E~~Q→SXx~~0~~	3	Clause number, which is needed for vertex clauses
SQ_SXx_Export_State	S~~E~~Q→SXx0	~~21?~~3	State ID, which is needed for vertex clauses

These fields are sent synchronously with SP export data, described in SP0→SX0 interface
{ISSUE: Where are the PC pointers}

~~22.1.14~~24.1.12 *SX~~0~~ to S~~E~~Q : Output file control*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SXx_SQ_Export_~~RTS~~count_rdy	~~SX0~~SXx→S~~E~~Q	1	Raised by SX0 to indicate that the following two fields reflect the result of the most recent export
SXx_SQ_Export_Position	SXx~~0~~→S~~E~~Q	1	Specifies whether there is room for another position.
SXx_SQ_Export_Buffer	SX~~0~~x→S~~E~~Q	7	Specifies the space ~~availble~~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

~~22.1.15 SP0 to SX0 : Position return bus~~

Formatted: Bullets and Numbering

~~22.1.16~~24.1.13 Shader Engine to Fetch Unit Bus ~~(Fast Bus)~~

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

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

Name	Direction	Bits	Description
~~Tex_RegFile_Read_Data~~SP0_TP0_fetch_addr	SP0->T~~EX~~P0	~~2048~~512	~~16~~ 4 Fetch Addresses read from the Register file
~~Tex_RegFile_Write_Data~~TP0_SP0_data	TP0~~EX~~→SP0	~~2048~~512	~~16~~ 4 texture results
SP1_TP1_fetch_addr	SP1->TP1	512	4 Fetch Addresses read from the Register file
TP1_SP1_data	TP1→SP1	512	4 texture results
SP2_TP2_fetch_addr	SP2->TP2	512	4 Fetch Addresses read from the Register file
TP2_SP2_data	TP2→SP2	512	4 texture results
SP3_TP3_fetch_addr	SP3->TP3	512	4 Fetch Addresses read from the Register file
TP3_SP3_data	TP3→SP3	512	4 texture results
TPx_SPx_gpr_dst	TPx→SPx	7	Write address into the gprs
TPx_SPx_gpr_cmask	TPx→SPx	4	Channel mask

~~22.1.17~~24.1.14 Sequencer to Fetch Unit bus ~~(Slow Bus)~~

Formatted: Bullets and Numbering

Once every ~~four~~ clock, the fetch unit sends to the sequencer on ~~wich~~which clause it is now working and if the data in the registers is ready or not. This way the sequencer can update the fetch counters for the reservation station fifos. The sequencer also provides the ~~intruction~~instruction and constants for the fetch to execute and the address in the register file where to write the fetch return data.

Name	Direction	Bits	Description
~~Tex_Ready~~TPx_SQ_data_rdy	T~~EX~~Px→ S~~E~~Q	1	Data ready
TPx_SQ_clause_num~~Tex_Clause_Num~~	T~~EX~~Px→ S~~E~~Q	3	Clause number
SQ_TPx_const~~Tex_cst~~	S~~E~~Q→T~~EX~~Px	~~10~~64	Fetch state ~~address 10 bits~~ sent over 4 clocks
~~Tex_Inst~~SQ_TPx_instuct	S~~E~~Q→T~~EX~~Px	~~12~~24	Fetch instruction ~~address 12 bits~~ sent over 4 clocks
SQ_TPx_end_of_clause~~EO_CLAUSE~~	S~~E~~Q→T~~EX~~Px	1	Last instruction of the clause
~~PHASE~~SQ_TPx_phase	S~~E~~Q→T~~EX~~Px	~~1~~2	Write phase signal
SQ_TP0_lod_correct~~LOD CORRECT~~	S~~E~~Q→T~~EX~~P0	6~~96~~	LOD correct 3 bits per comp 2 components per quad ~~* 16~~ quads
~~Mask~~SQ_TP0_pmask	S~~E~~Q→T~~EX~~P0	~~64~~4	Pixel mask 1 bit per pixel
SQ_TP1_lod_correct	SQ→TP1	6	LOD correct 3 bits per comp 2 components per quad quads
SQ_TP1_pmask	SQ→TP1	4	Pixel mask 1 bit per pixel
SQ_TP2_lod_correct	SQ→TP2	6	LOD correct 3 bits per comp 2 components per quad quads
SQ_TP2_pmask	SQ→TP2	4	Pixel mask 1 bit per pixel
SQ_TP3_lod_correct	SQ→TP3	6	LOD correct 3 bits per comp 2 components per quad quads
SQ_TP3_pmask	SQ→TP3	4	Pixel mask 1 bit per pixel
~~Tex_Clause_Num~~SQ_TPx_clause_num	S~~E~~Q→T~~EX~~Px	3	Clause number
~~Tex_Write_Register_Index~~SQ_TPx_write_gpr_index	S~~E~~Q->T~~EX~~Px	7	Index into Register file for write of

			returned Fetch Data

24.1.15 Sequencer to SP: GPR control

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_re_addr	SQ→SPx	1	Read Enable
SQ_SPx_gpr_we_addr	SQ→SPx	1	Write Enable
SQ_SPx_gpr_phase_mux	SQ→SPx	2	The phase mux
SQ_SPx_gpr_channel_mask	SQ→SPx	4	The channel mask
SQ_SP0_gpr_pixel_mask	SQ→SP0	4	The pixel mask
SQ_SP1_gpr_pixel_mask	SQ→SP1	4	The pixel mask
SQ_SP2_gpr_pixel_mask	SQ→SP2	4	The pixel mask
SQ_SP3_gpr_pixel_mask	SQ→SP3	4	The pixel mask

24.1.16 Sequencer to SPx: Parameter cache write control

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_pc_wr_addr	SQ→SPx	7	Write address
SQ_SPx_pc_we_addr	SQ→SPx	1	Write Enable
SQ_SPx_pc_phase_mux	SQ→SPx	1	The output selector mux (gpr vs parameter cache)

24.1.17 Sequencer to SPx: Instructions

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_instruct_start	SQ→SPx	1	Instruction start
SQ_SP_instruct	SQ→SPx	20	Instruction sent over 4 clocks
SQ_SPx_stall	SQ→SPx	1	Stall signal
SQ_SPx_Shader_Count	SQ→SPx	3	Each set of four pixels or vectors is exported over eight clocks. This field specifies where the SP is in that sequence.
SQ_SPx_Shader_Last	SQ→SPx	1	Asserted on the first shader count of the last export of the clause
SQ_SP0_Shader_PixelValid	SQ→SP0	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP0_Shader_WordValid	SQ→SP0	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SQ_SP1_Shader_PixelValid	SQ→SP1	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP1_Shader_WordValid	SQ→SP1	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SQ_SP2_Shader_PixelValid	SQ→SP2	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP2_Shader_WordValid	SQ→SP2	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SQ_SP3_Shader_PixelValid	SQ→SP3	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP3_Shader_WordValid	SQ→SP3	2	Specifies whether to write low and/or high 32-bit

			word of the 64-bit export data from each of the 16 pixels or vectors

24.1.18 *SP to Sequencer: Constant address load*

Formatted: Bullets and Numbering

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

Formatted

Formatted

Formatted

Formatted

24.1.19 *Sequencer to SPx: constant broadcast*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_constant	SQ→SPx	128	Constant broadcast

Formatted

Formatted

Formatted

Formatted

24.1.20 *SP0 to Sequencer: 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

24.1.21 *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)

24.1.22 *CP to SQ: RBBM bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
rbbm_we	CP→SQ	1	Write Enable
rbbm_a	CP→SQ	18	Address -- Upper Extent is TBD
rbbm_wd	CP→SQ	32	Data
rbbm_be	CP→SQ	4	Byte Enables
rbbm_re	CP→SQ	1	Read Enable
rbb_rs0	CP→SQ	1	Read Return Strobe 0
rbb_rs1	CP→SQ	1	Read Return Strobe 1
rbb_rd0	CP→SQ	32	Read Data 0
rbb_rd1	CP→SQ	32	Read Data 0
RBBM_SQ_soft_reset	CP→SQ	1	Soft Reset

~~23.~~25. Examples of program executions

Formatted: Bullets and Numbering

~~23.1.1~~25.1.1 *Sequencer Control of a Vector of Vertices*

1. PA sends a vector of 64 vertices (actually vertex indices – 32 bits/index for 2048 bit total) to the RE's Vertex FIFO
   - state pointer as well as tag into position cache is sent along with vertices
   - space was allocated in the position cache for transformed position before the vector was sent

   - **also before the vector is sent to the RE, the CP has loaded the global instruction store with the vertex shader program (using the MH?)**
   - The vertex program is assumed to be loaded when we receive the vertex vector.
     - the SEQ then accesses the IS base for this shader using the local state pointer (provided to all sequencers by the RBBM when the CP is done loading the program)

2. SEQ arbitrates between the Pixel FIFO and the Vertex FIFO – basically the Vertex FIFO always has priority
   - at this point the vector is removed from the Vertex FIFO
   - the ~~arbitrer~~arbiter is not going to select a vector to be transformed if the parameter cache is full unless the pipe as nothing else to do (ie no pixels are in the pixel fifo).
3. SEQ allocates space in the SP register file for index data plus GPRs used by the program
   - the number of GPRs required by the program is stored in a local state register, which is accessed using the state pointer that came down with the vertices
   - SEQ will not send vertex data until space in the register file has been allocated
4. SEQ sends the vector to the SP register file over the RE_SP interface (which has a bandwidth of 2048 bits/cycle)
   - the 64 vertex indices are sent to the 64 register files over 4 cycles
     - RF0 of SU0, SU1, SU2, and SU3 is written the first cycle
     - RF1 of SU0, SU1, SU2, and SU3 is written the second cycle
     - RF2 of SU0, SU1, SU2, and SU3 is written the third cycle
     - RF3 of SU0, SU1, SU2, and SU3 is written the fourth cycle
   - the index is written to the least significant 32 bits **(floating point format?) (what about compound indices)** of the 128-bit location within the register file (w); the remaining data bits are set to zero (x, y, z)
5. SEQ constructs a control packet for the vector and sends it to the first reservation station (the FIFO in front of fetch state machine 0, or TSM0 FIFO)
   - the control packet contains the state pointer, the tag to the position cache and a register file base pointer.
6. TSM0 accepts the control packet and fetches the instructions for fetch clause 0 from the global instruction store
   - TSM0 was first selected by the TSM arbiter before it could start
7. all instructions of fetch clause 0 are issued by TSM0
8. the control packet is passed to the next reservation station (the FIFO in front of ALU state machine 0, or ASM0 FIFO)
   - TSM0 does not wait for requests made to the Fetch Unit to complete; it passes the register file write index for the fetch data to the TU, which will write the data to the RF as it is received
   - once the TU has written all the data to the register files, it increments a counter that is associated with ASM0 FIFO; a count greater than zero indicates that the ALU state machine can go ahead start to execute the ALU clause
9. ASM0 accepts the control packet (after being selected by the ASM arbiter) and gets the instructions for ALU clause 0 from the global instruction store
10. all instructions of ALU clause 0 are issued by ASM0, then the control packet is passed to the next reservation station (the FIFO in front of fetch state machine 1, or TSM1 FIFO)
11. the control packet continues to travel down the path of reservation stations until all clauses have been executed
    - position can be exported in ALU clause 3 (or 4?); the data (and the tag) is sent over a position bus (which is shared with all four shader pipes) back to the PA's position cache
    - A parameter cache pointer is also sent along with the position data. This tells to the PA where the data is going to be in the parameter cache.
      - there is a position export FIFO in the SP that buffers position data before it gets sent back to the PA
      - the ASM arbiter will prevent a packet from starting an exporting clause if the position export FIFO is full
    - parameter data is exported in clause 7 (as well as position data if it was not exported earlier)
      - parameter data is sent to the Parameter Cache over a dedicated bus
      - the SEQ allocates storage in the Parameter Cache, and the SEQ deallocates that space when there is no longer a need for the parameters (it is told by the PA when using a token).
      - the ASM arbiter will prevent a packet from starting on ASM7 if the parameter cache (or the position buffer if position is being exported) is full

12. after the shader program has completed, the SEQ will free up the GPRs so that they can be used by another shader program

~~23.1.2~~25.1.2 *Sequencer Control of a Vector of Pixels*

Formatted: Bullets and Numbering

1. **As with vertex shader programs, pixel shaders are loaded into the global instruction store by the CP**
    - At this point it is assumed that the pixel program is loaded into the instruction store and thus ready to be read.
2. the RE's Pixel FIFO is loaded with the barycentric coordinates for pixel quads by the detailed walker
    - the state pointer and the LOD correction bits are also placed in the Pixel FIFO
    - the Pixel FIFO is wide enough to source four quad's worth of barycentrics per cycle
3. SEQ arbitrates between Pixel FIFO and Vertex FIFO – when there are no vertices pending OR there is no space left in the register files for vertices, the Pixel FIFO is selected
4. SEQ allocates space in the SP register file for all the GPRs used by the program
    - the number of GPRs required by the program is stored in a local state register, which is accessed using the state pointer
    - SEQ will not allow interpolated data to be sent to the shader until space in the register file has been allocated
5. SEQ controls the transfer of interpolated data to the SP register file over the RE_SP interface (which has a bandwidth of 2048 bits/cycle). See interpolated data bus diagrams for details.
6. SEQ constructs a control packet for the vector and sends it to the first reservation station (the FIFO in front of fetch state machine 0, or TSM0 FIFO)
    - note that there is a separate set of reservation stations/arbiters/state machines for vertices and for pixels
    - the control packet contains the state pointer, the register file base pointer, and the LOD correction bits
    - all other ~~informations~~information (such as quad address for example) travels in a separate FIFO
7. TSM0 accepts the control packet and fetches the instructions for fetch clause 0 from the global instruction store
    - TSM0 was first selected by the TSM arbiter before it could start
8. all instructions of fetch clause 0 are issued by TSM0
9. the control packet is passed to the next reservation station (the FIFO in front of ALU state machine 0, or ASM0 FIFO)
    - TSM0 does not wait for fetch requests made to the Fetch Unit to complete; it passes the register file write index for the fetch data to the TU, which will write the data to the RF as it is received
    - once the TU has written all the data for a particular clause to the register files, it increments a counter that is associated with the ASM0 FIFO; a count greater than zero indicates that the ALU state machine can go ahead and pop the FIFO and start to execute the ALU clause
10. ASM0 accepts the control packet (after being selected by the ASM arbiter) and gets the instructions for ALU clause 0 from the global instruction store
11. all instructions of ALU clause 0 are issued by ASM0, then the control packet is passed to the next reservation station (the FIFO in front of fetch state machine 1, or TSM1 FIFO)
12. the control packet continues to travel down the path of reservation stations until all clauses have been executed
    - pixel data is exported in the last ALU clause (clause 7)
        - it is sent to an output FIFO where it will be picked up by the render backend
        - the ASM arbiter will prevent a packet from starting on ASM7 if the output FIFO is full
13. after the shader program has completed, the SEQ will free up the GPRs so that they can be used by another shader program

~~23.1.3~~25.1.3 *Notes*

Formatted: Bullets and Numbering

14. The state machines and ~~arbitrers~~arbiters will operate ahead of time so that they will be able to immediately start the real threads or stall.

15. The register file base pointer for a vector needs to travel with the vector through the reservation stations, but the instruction store base pointer does not – this is because the RF pointer is different for all threads, but the IS pointer is only different for each state and thus can be accessed via the state pointer
16. Waterfalling still needs to be specked out.

Formatted: Bullets and Numbering

~~24.~~26. Open issues

There is currently an issue with constants. If the constants are not the same for the whole vector of vertices, we don't have the ~~bandwith~~bandwidth from the fetch store to feed the ALUs. Two solutions exists for this problem:

1) Let the compiler handle the case and put those instructions in a fetch clause so we can use the ~~bandwith~~bandwidth there to operate. This requires a significant amount of temporary storage in the register store.
2) Waterfall down the pipe allowing only at a given time the vertices having the same constants to operate in ~~parrallel~~parallel. This might in the worst case slow us down by a factor of 16.

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

Saving power?

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 20155 ~~December, 200126~~	GEN-CXXXXX-REVA	1 of 43

Author: Laurent Lefebvre

Issue To:

Copy No:

R400 Sequencer Specification

S~~E~~Q

Version 1.4~~3~~

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~~C:\perforce\r400\doc_lib\parts\sq\R400_Sequencer.docC:\perforce\r400\arch\doc\gfx\RE\R400_Sequencer.doc~~

APPROVALS	
Name/Dept	Signature/Date

Remarks:

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

ATI 2020
LG v. ATI
IPR2015-00325

Table Of Contents

Revision Changes:

Rev 0.1 (Laurent Lefebvre) Date: May 7, 2001	First draft.
Rev 0.2 (Laurent Lefebvre) Date : July 9, 2001	Changed the interfaces to reflect the changes in the SP. Added some details in the arbitration section.
Rev 0.3 (Laurent Lefebvre) Date : August 6, 2001	Reviewed the Sequencer spec after the meeting on August 3, 2001.
Rev 0.4 (Laurent Lefebvre) Date : August 24, 2001	Added the dynamic allocation method for register file and an example (written in part by Vic) of the flow of pixels/vertices in the sequencer.
Rev 0.5 (Laurent Lefebvre) Date : September 7, 2001	Added timing diagrams (Vic)
Rev 0.6 (Laurent Lefebvre) Date : September 24, 2001	Changed the spec to reflect the new R400 architecture. Added interfaces.
Rev 0.7 (Laurent Lefebvre) Date : October 5, 2001	Added constant store management, instruction store management, control flow management and data dependant predication.
Rev 0.8 (Laurent Lefebvre) Date : October 8, 2001	Changed the control flow method to be more flexible. Also updated the external interfaces.
Rev 0.9 (Laurent Lefebvre) Date : October 17, 2001	Incorporated changes made in the 10/18/01 control flow meeting. Added a NOP instruction, removed the conditional_execute_or_jump. Added debug registers.
Rev 1.0 (Laurent Lefebvre) Date : October 19, 2001	Refined interfaces to RB. Added state registers.
Rev 1.1 (Laurent Lefebvre) Date : October 26, 2001	Added SEQ→SP0 interfaces. Changed delta precision. Changed VGT→SP0 interface. Debug Methods added.
Rev 1.2 (Laurent Lefebvre) Date : November 16, 2001	Interfaces greatly refined. Cleaned up the spec.
Rev 1.3 (Laurent Lefebvre) Date : November 26, 2001	Added the different interpolation modes.
Rev 1.4 (Laurent Lefebvre) Date : December 6, 2001	Added the auto incrementing counters. Changed the VGT→SQ interface. Added content on constant management. Updated registers.

1. Overview

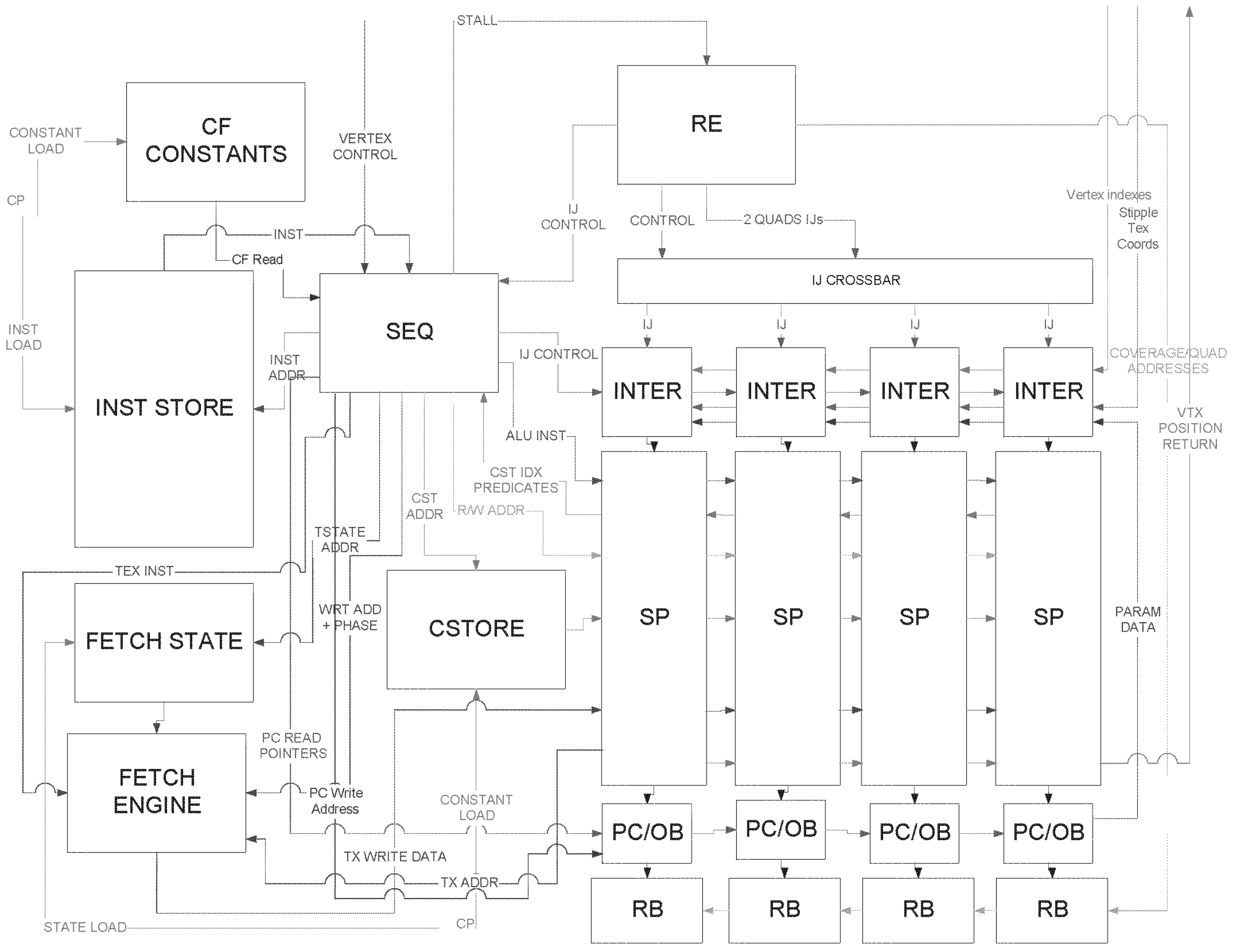
The sequencer is based on the R300 design. It chooses two ALU clauses and a fetch clause to execute, and executes all of the instructions in a clause before looking for a new clause of the same type. Two ALU clauses are executed interleaved to hide the ALU latency. Each vector will have eight fetch and 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. 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.

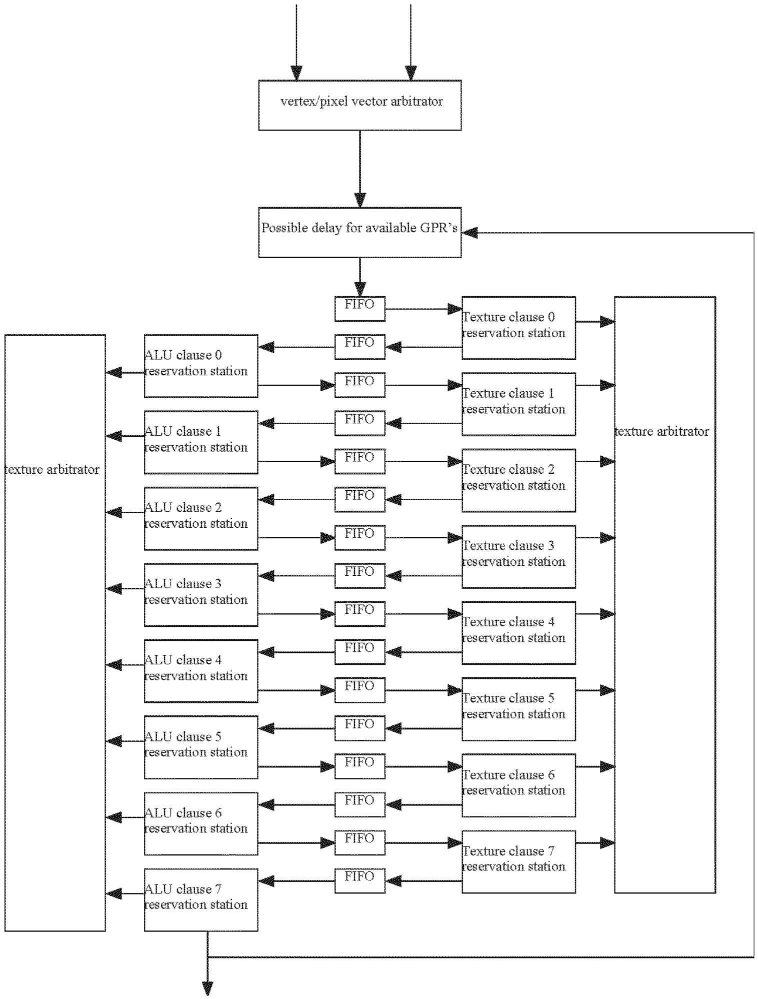
The sequencer first arbitrates between vectors of 64 vertices that arrive directly from primitive assembly and vectors of 16 quads (64 pixels) that are generated in the scan converter.

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.

PROTECTIVE ORDER MATERIAL



1.1 Top Level Block Diagram


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

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 registers to store the interpolated values and temporaries. Following this, the barycentric coordinates (and XY screen position if needed) are sent to the interpolator buffers which are going to use these barycentric coordinates to interpolate the parameters and place the interpolated values 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 texture request and corresponding register 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 register 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 1 that the data is ready to be processed.

On receipt of a command, the level 0 ALU machine first decrements the input FIFO counter and then issues a complete set of level 0 shader instructions. For each instruction, the state machine generates 3 source addresses, one destination address (3 cycles later) and an instruction. Once the last instruction as 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.

{ISSUE: How do we handle parameter cache pointers (computed, semi-computed or not computed)?}

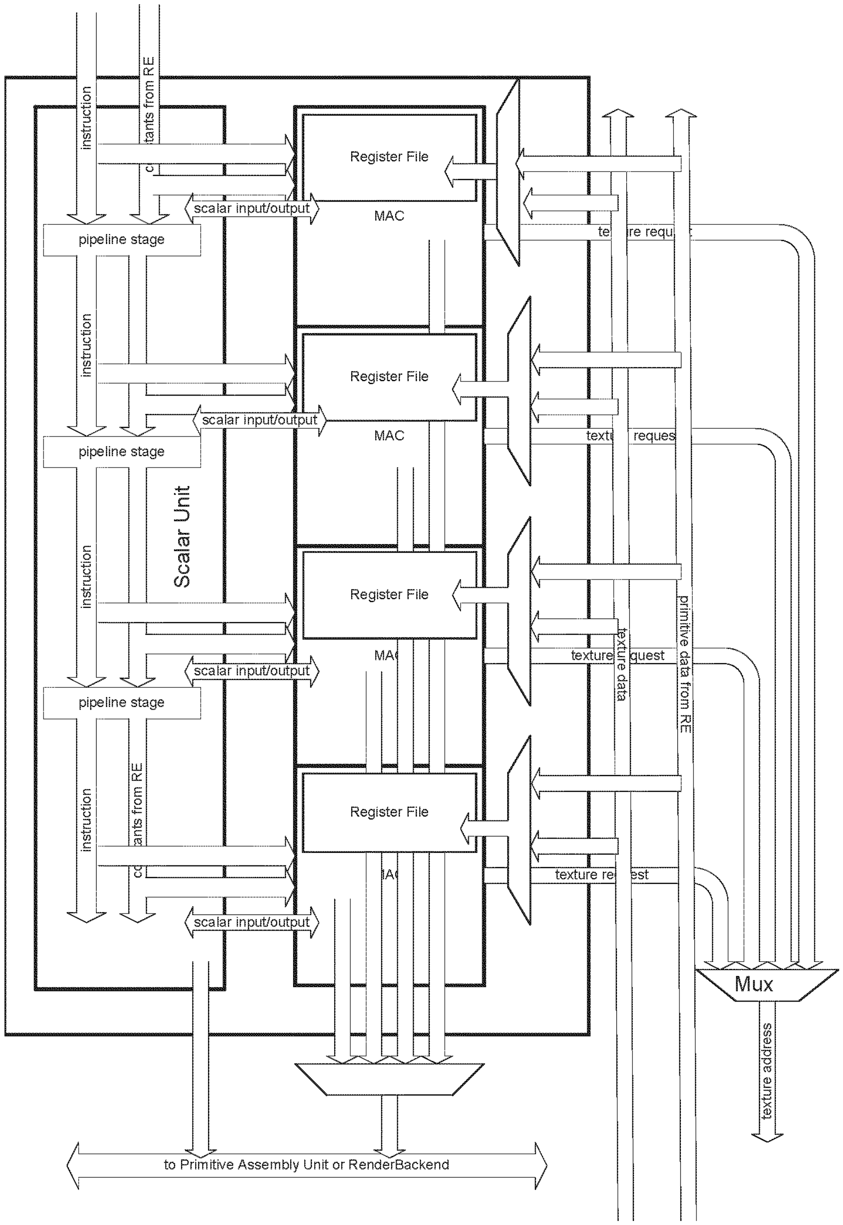
A special case is for multipass vertex ~~shaders which~~shaders, which can export 12 parameters per last 6 clauses to the output buffer. If the output buffer is full or doesn't have enough space the sequencer will prevent such a vertex group to enter an exporting clause.

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

All other level process in the same way until the packet finally reaches the last ALU machine (7). ~~Upon completion of a vertex shader, a bit is sent to the SC to let it know that it can begin sending pixels of this group to the sequencer.~~

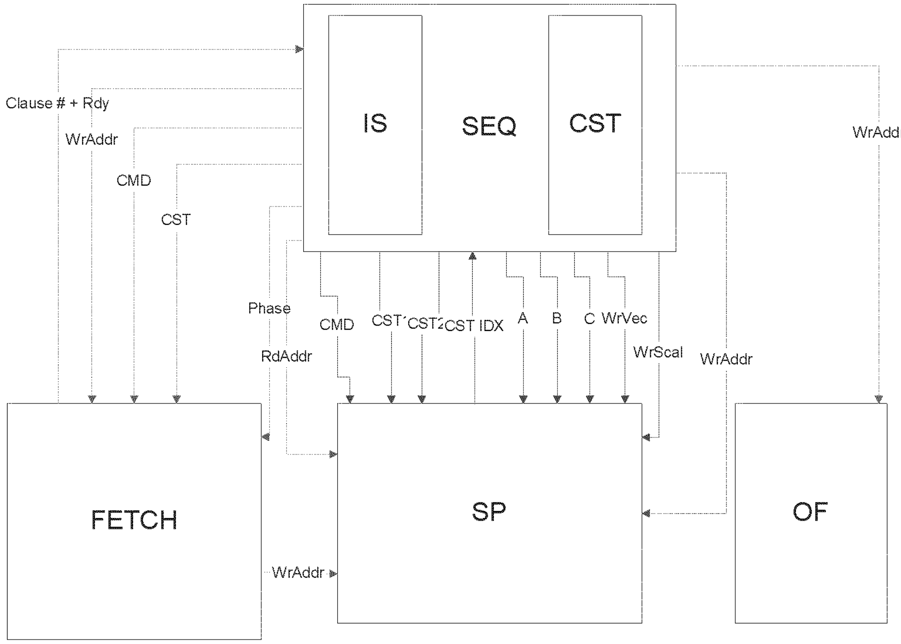
Only two ALU state machine may have access to the register file address bus or the instruction decode bus at one time. Similarly, only one fetch state machine may have access to the register file address bus at one time. Arbitration is performed by three arbiter blocks (two for the ALU state machines 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.

1.2 Data Flow graph



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

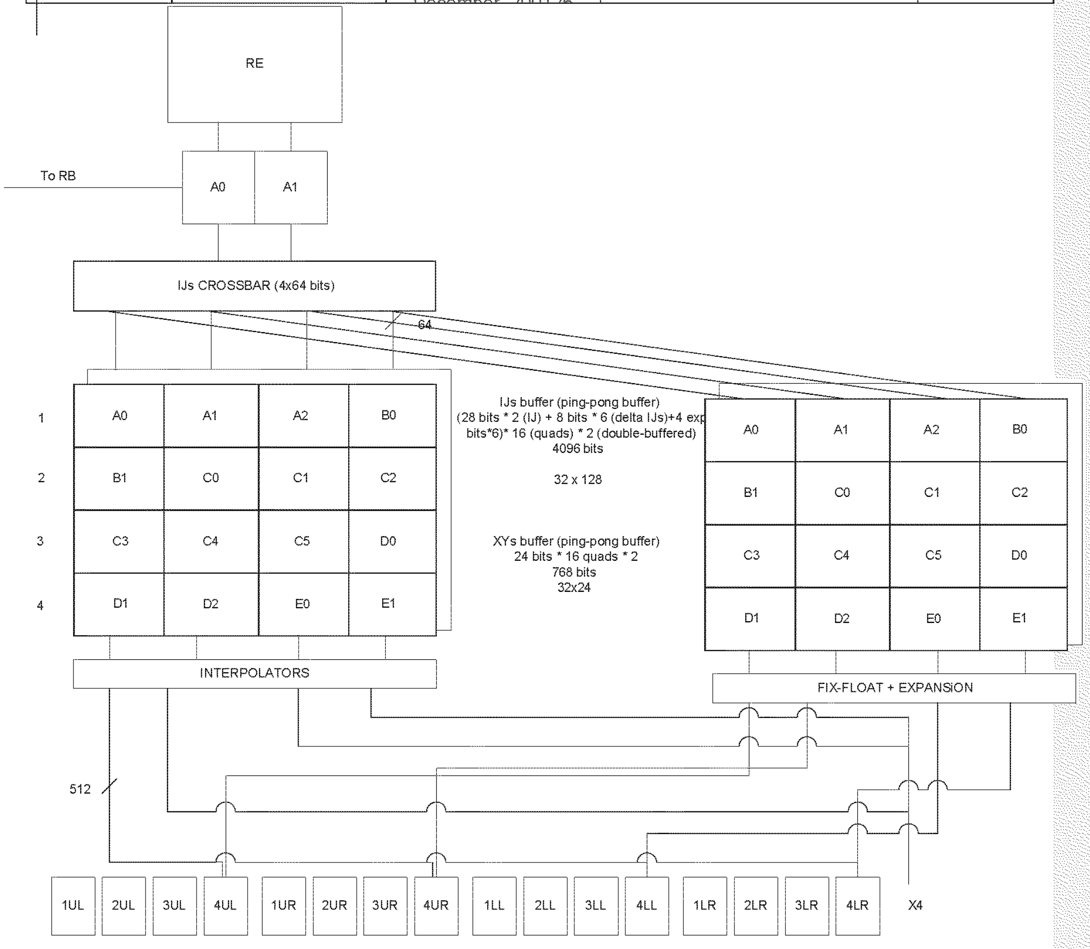
1.3 Control Graph



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.



IJs buffer (ping-pong buffer)
(28 bits * 2 (IJ) + 8 bits * 6 (delta IJs)+4 exp
bits*6)* 16 (quads) * 2 (double-buffered)
4096 bits

32 x 128

XYs buffer (ping-pong buffer)
24 bits * 16 quads * 2
768 bits
32x24

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 20155 December, 200126	GEN-CXXXXX-REVA	13 of 43

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19	T20	T21	T22	T23
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			

Above is an example of a tile we might receive. The IJ information is packed in the IJ buffer 2 quads at a time. 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 register 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.

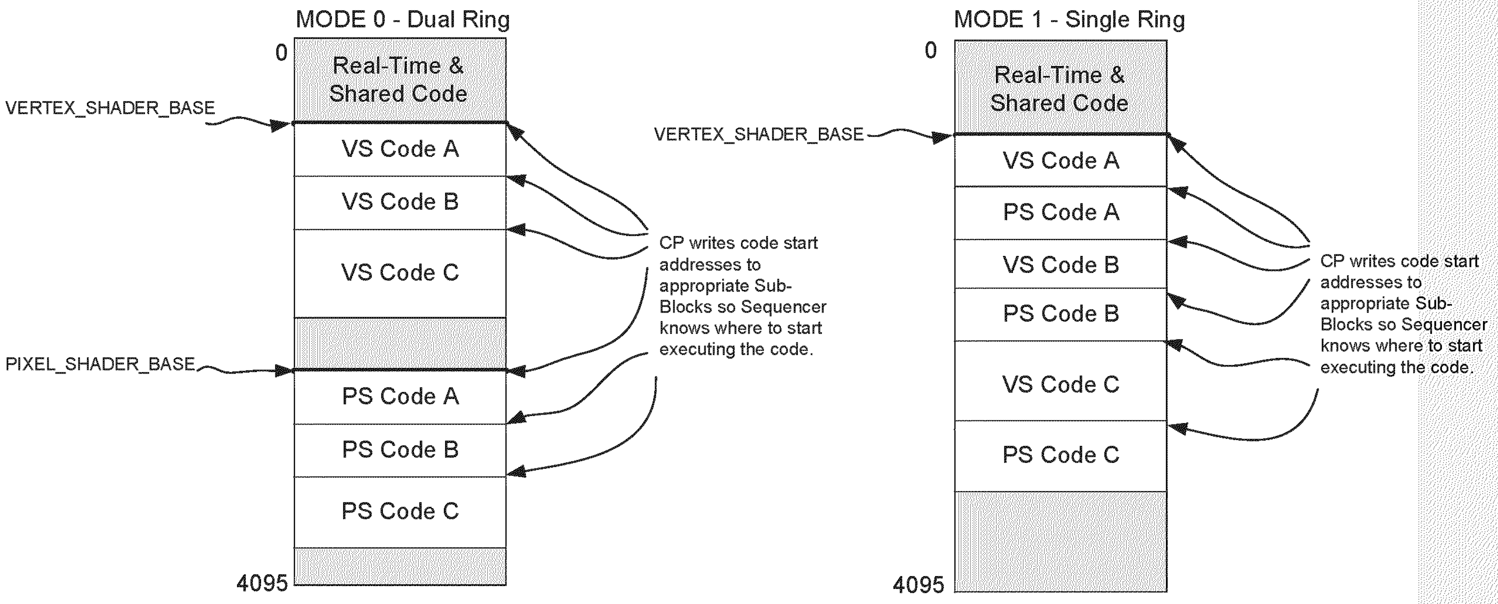
The instruction store is loaded by the CP thru the INST_DATA~~INSTRUCTION_DATA~~, INST_INDEX_PORT ~~INSTRUCTION_INDEX_PORT~~ control registers. The INST_INDEX_PORT~~INSTRUCTION_INDEX_PORT~~ is auto-incremented on both reads and writes to the INST_DATA~~INSTRUCTION_DATA~~ register.

The next picture shows the various modes the CP can load the memory. The Sequencer has to keep track of the loading modes in order to wrap around the correct boundaries. The MSB of the INST_INDEX_PORT ~~INSTRUCTION_INDEX_PORT~~ register contains the packet type for the sequencer to know where it must wrap around. The wrap around points are arbitrary and they are specified in the ~~VERTEX_SHADER~~VS_BASE and PI~~XEL_SHADER~~X_BASE registers.

For the Real time commands the story is quite the same but for some small differences. The CP will use the INST_INDEX_PORT_RT and INST_DATA_RT register pair instead of the regular ones and 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.

R400 CP's Views of Instruction Memory

Updated: 11/14/2001
John A. Carey



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).

5. Constant Stores

5.1 Memory organizations

Formatted: Bullets and Numbering

~~The sequencer is aware of where the constants are using a remaping table.~~ 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)~~32/4 bits/clock~~.

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 remaping 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.

The texture state is also kept in a similar memory. The size of this memory is 192x128. The memory thus holds 128 texture states (192 bits per state). The logical size exposed 32 different states total, which are going to be shared between the pixel and the vertex shader. The size of the remaping table to for the texture state memory is 16 lines (each line addresses 2 texture state lines in the real memory). The write granularity is 2 texture state lines (or 384 bits). The driver sends 512 bits but the CP ignores the top 128 bits. It thus takes 12 clocks to write the two texture states.

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 state change. Its size is 256*32 because it must hold 8 copies of the 32 dwords of control flow constants.

The CP is loading the constant store using the CONST_DATA and CONST_ADDR registers. It does so by writing to the CONST_ADDR register the logical address for the constant block it wants to update and then writes 16 times to the CONST_DATA register. The CONST_ADDR is auto-incremented on both reads and writes to the CONST_DATA register.

5.2 Management of the remaping tables

Formatted: Bullets and Numbering

The sequencer is responsible to manage two remaping 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 remaping tables to a new one. We have 8 different remaping tables we can use concurrently. More details and a diagram to come….

~~5.1~~5.3 Constant Store Indexing

Formatted: Bullets and Numbering

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.

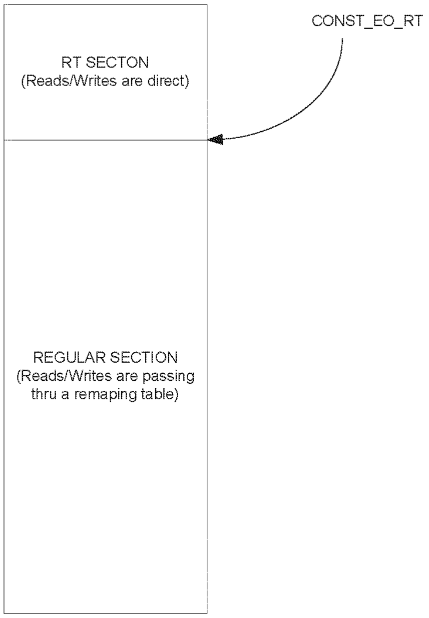
The storage needed in the sequencer in order to support this feature is 2*64*9 bits = 1152 bits.

Formatted: Bullets and Numbering

5.4

5.5 Real Time Commands

The real time commands constants are written by the CP using the CONST_DATA_RT and CONST_ADDR_RT registers. 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 remaping table but are directly read from the memory. The boundary between the two zones is defined by the CONST_EO_RT control register.



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

~~The texture state is also kept in a similar memory. The size of this memory is 192x128. Which lets us load a texture state in 2 clocks. The memory thus holds 96 texture states (2*128 bits per state)~~

~~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 state change. Its size is 192*32 because it must hold 8 copies of the 24 dwords of control flow constants.~~

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:
~~As per Dx9 the following state is available for control flow:~~

~~Boolean[15:0]~~
~~loop_count[7:0][7:0]~~
~~In addition:~~

~~loop_start [7:0] [7:0]~~
~~loop_step [7:0] [7:0]~~
~~Exist to give more control to the controlling program.~~

~~We will extend that in the R400 to:~~
Boolean[~~255~~256:0]
Loop_count[7:0][~~31:0~~31:0]
Loop_Start[7:0]~~[~~[~~31:0~~31:0]
Loop_Step[7:0]~~[31:~~31:0]
~~Loop_End[7:0] [31:0]~~

That is 256 Booleans and 32 loops.

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

6.2 The Control Flow Program

~~The R300 uses a match method for control flow: The shader is executed, and at every instruction its address is compared with addresses (or address?) in a control table. The "event" in the control table can redirect operations in the program.~~
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 Method chosen for the R400 is a "control program".~~ The control program has ~~ten~~ eleven basic instructions:

Execute
Conditional_execute
Conditional_Execute_Predicates
Conditional_jump
Call
Return
Loop_start
Loop_end
End_of_clause
Conditional_End_of_clause
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.
Call jumps to an address and pushes the IP counter on the stack. On the return instruction, the IP is popped from the stack.
Conditional_execute_or_Jump executes a block of instructions or jumps to an address is the condition is not met.
Conditional_execute_Predicates executes a block of instructions if all bits in the predicate vectors meet the condition.
End_of_clause marks the end of a clause.

Conditional_End_of_clause marks the end of a clause if the condition is met.
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. 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.

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

Formatted

Execute				
47	46… 42	41 … 24	23 … 12	11 … 0
Addressing	00001	RESERVED	Instruction _count	Exec Address

Execute up to 4k instructions at the specified address in the instruction memory.

NOP		
47	46 … 42	41 … 0
Addressing	00010	RESERVED

This is a regular NOP.

Conditionnal_Execute							
47	46 … 42	41~~41 … 34~~	40 … 33	~~33~~32	~~32~~31 … 24	23 … 12	11 … 0
Addressing	00011	RESERVED ~~Boolean address~~	Boolean address	Condition	RESERVED	Instruction _count	Exec Address

If the specified boolean (8 bits can address 256 booleans) meets the specified condition then execute the specified instructions (up to 4k instructions)

Conditionnal_Execute_Predicates							
47	46 … 42	41 … 35	~~41~~ 34 … ~~38~~33	~~37~~32	~~36~~ 31 … 24	23 … 12	11 … 0
Addressing	00100	RESERVED	Predicate vector	Condition	RESERVED	Instruction_ 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.

Loop_Start				
47	46 … 42	41 … 17	~~16 … 5~~16 … 12	~~4 … 0~~11 … 0
Addressing	00101	RESERVED	~~Jump address~~loop ID	~~Loop ID~~Jump address

Loop Start. Compares the loop count with the end value. If loop condition not met jump to the address. Forward jump only. Also computes the index value.

Loop_End				
47	46 … 42	41 … 17	16 … ~~5~~12	~~4 … 0~~11 … 0

Addressing	00111	RESERVED	~~Start address~~loop ID	~~Loop ID~~start address

Loop end. Increments the counter by one and jumps BACK only 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.

Call			
47	46 … 42	41…12	11 … 0
Addressing	01000	RESERVED	Jump address~~Address~~

Jumps to the specified address and pushes the IP counter on the stack.

Return		
47	46 … 42	41 … 0
Addressing	01001	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 … 42	41~~41 … 34~~	40 … 33	~~33~~32	~~32 … 13~~31	~~12~~30 … 12	11 … 0
Addressing	01010	RESERVED ~~Boolean address~~	Boolean address	Condition	FW only~~RESERVED~~	RESERVED~~FW only~~	Jump address~~Address~~

If condition met, jumps to the address. FORWARD jump only allowed if bit ~~12~~ 31 set. Bit ~~12~~ 31 is only an optimization for the compiler and should NOT be exposed to the API.

Conditional_End_of_Clause					
47	46 … 42	41~~41 … 34~~	40 … 33	~~33~~32	~~32~~ 31 … 0
Addressing	01011	RESERVED ~~Boolean address~~	Boolean address	Condition	RESERVED

This is an optimization in the case of very short shaders (where the control flow instruction can't be hidden anymore and thus are not free. In this case, if the condition is met, the clause is ended, else we continue the execution of the clause.

End_of_Clause		
47	46 … 42	41 … 0
Addressing	01011	RESERVED

Marks the end of a clause.

To prevent infinite loops, we will keep 9 bits loop counters 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 registers.

~~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.~~

6.3 Data dependant predicate instructions

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

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

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

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

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

P0_ADD_# R0,R1,R2

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

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

6.4 HW Detection of PV,PS

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

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

6.5 Register file indexing

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

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

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

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

Index = Loop_counter*Loop_iterator + Loop_init.

The index is going to return 0 if it is out of the range.

6.6 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.**

6.7 Debugging the Shaders

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

6.7.1 *Method 1: Debugging registers*

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

The sequencer will detect the following groups of errors:
- count overflow
- jump error
  relative jump address > size of the control flow program
  relative jump address > length of the shader program
- constant overflow
- register overflow
- call stack
  call with stack full
  return with stack empty

With two of the errors, a jump error or a register overflow will 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 the other errors, program can continue to run, potentially to worst-case limits.

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

{ISSUE : Interrupt to the driver or not?}

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

The sequencer will have a 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 :

1) Normal
2) Debug Kill
3) Debug Addr + Count

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 clause 7 is always exporting 12 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:

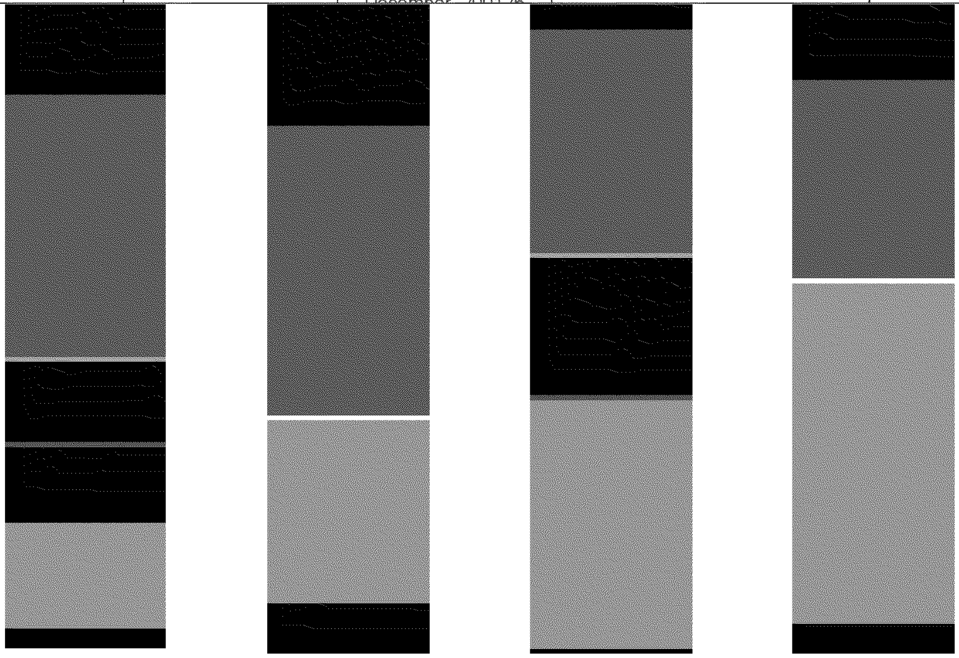
MASK_SETE
MASK_SETNE
MASK_SETGT
MASK_SETGTE

8. ~~HOS surfaces~~Multipass vertex shaders (HOS)

~~HOS~~Multipass vertex shaders ~~surfaces~~ are able to export from the 6 last clauses but to memory ONLY. ~~If they want to export to the parameter cache they have to do it in the last clause (7). They can also export position in clause 3.~~

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 VERTEX_REG_SIZE for vertices and 256-VERTEX_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.

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 arbitrers, one for the even clocks and one for the odd clocks. For exemple, 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.

12. Handling Stalls

When the output file is full, the sequencer prevents the ALU arbitration logic to select 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 to enter 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

~~21~~ 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, quad address and 1 bit to specify if the vector is of pixels or vertices. Since pixels and vertices are kept in order in the shader pipe, we only need two fifos (one for vertices and one for pixels) deep enough to cover the shader pipe latency. This size will be determined later when we will know the size of the small fifos between the reservation stations.

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).

$$\begin{aligned}
\Delta 01I &= I(1) - I(0)\\
\Delta 01J &= J(1) - J(0)\\
\Delta 02I &= I(2) - I(0)\\
\Delta 02J &= J(2) - J(0)\\
\Delta 03I &= I(3) - I(0)\\
\Delta 03J &= J(3) - J(0)
\end{aligned}$$

P0	P1
P2	P3

$$\begin{aligned}
P0 &= C + I(0)*(A-C) + J(0)*(B-C)\\
P1 &= P0 + \Delta 01I*(A-C) + \Delta 01J*(B-C)\\
P2 &= P0 + \Delta 02I*(A-C) + \Delta 02J*(B-C)\\
P3 &= P0 + \Delta 03I*(A-C) + \Delta 03J*(B-C)
\end{aligned}$$

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 : ~~19~~20*2 + ~~8*6 + 4*8 + 4*2~~8*6 + 4*8 + 4*2 = 128

The Deltas have a leading 1, the Full precision IJs don't. This means that in the case of the deltas we MUST be able to shift 8 right (exponent value of 0 means number = 0, exponent value of 1 means shift right 8~~8~~). This means that the maximum range for the IJs (Full precision) is +/- 63~~4~~ and the range for the Deltas is +/- 12~~8~~7.

15.1 Interpolation of constant attributes

Formatted: Bullets and Numbering

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. 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.

17. 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.

18. Exporting Arbitration

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

1) Position exports and position exports cannot be co-issued.
2) Position exports and memory exports cannot be co-issued.
3) Position exports and Z/Color exports cannot be co-issued.
4) Memory exports and Z/Color exports cannot be co-issued.
5) Memory exports and memory exports cannot be co-issued.
6) Z/color exports and Z/color exports cannot be co-issued.
7) Parameter exports and Z/Color exports CAN be co-issued.
8) Parameter exports and parameter exports CAN be co-issued.
9) Parameter exports and memory exports CAN be co-issued.

Formatted: Bullets and Numbering

19. 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:

Formatted: Bullets and Numbering

19.1 Vertex Shading

0:15	- 16 parameter cache
16:31	- Empty (Reserved?)
32:43	- 12 vertex exports to the frame buffer and index
44:47	- Empty
48:59	- 12 debug export (interpret as normal vertex export)
60	- export addressing mode
61	- Empty
62	- sprite size export that goes with position export (point_h,point_w,edgeflag,misc)
63	- position

Formatted: Bullets and Numbering

19.2 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?)
32:43	- 12 exports for multipass pixel shaders.
44:47	- Empty
48:59	- 12 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

~~19.~~20. Special Interpolation modes

~~19.1~~20.1 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 realtime 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.

~~19.2~~20.2 Sprites/ XY screen coordinates/ FB information

Formatted: Bullets and Numbering

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_I0 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 registers. 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 ~~be read from the GEN_S and GEN_T state registers instead of being read from the parameter cache~~generated between 0 and 1.

Param_Gen_I0 disable, snd_xy disable, no gen_st – I0 = No modification
Param_Gen_I0 disable, snd_xy disable, gen_st – I0 = No modification
Param_Gen_I0 disable, snd_xy enable, no gen_st – I0 = No modification
Param_Gen_I0 disable, snd_xy enable, gen_st – I0 = No modification
Param_Gen_I0 enable, snd_xy disable, no gen_st – I0 = garbage, garbage, garbage, faceness
Param_Gen_I0 enable, snd_xy disable, gen_st – I0 = garbage, garbage, s, t
Param_Gen_I0 enable, snd_xy enable, no gen_st – I0 = screen x, screen y, garbage, faceness
Param_Gen_I0 enable, snd_xy enable, gen_st – I0 = screen x, screen y, s, t

20.3 Auto generated counters

Formatted: Bullets and Numbering

In the cases we are dealing with multipass shaders, the sequencer is going to generate a vector count to be able to both use this count to write the 1st pass data to memory and then use the count to retrieve the data on the 2nd pass. The count is always generated in the same way but it is passed to the shader in a slightly different way depending on the shader type (pixel or vertex). This is toggled on and off using the GEN_INDEX register. While there is only one count broadcast to the registers, the LSB are hardwired to specific values making the index different for all elements in the vector.

Formatted

Formatted

20.3.1 *Vertex shaders*

Formatted: Bullets and Numbering

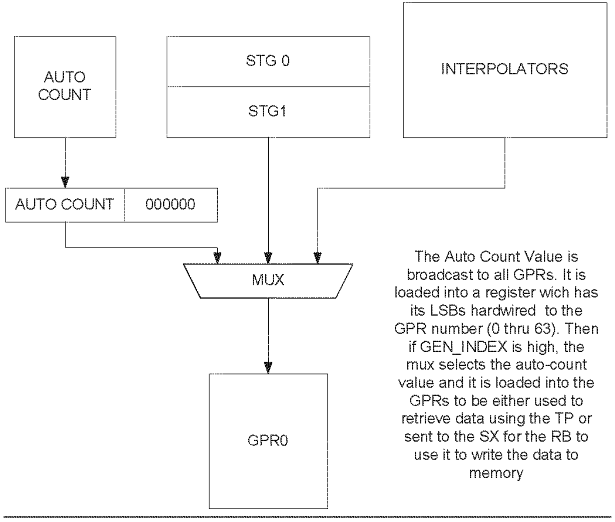
In the case of vertex shaders, if GEN_INDEX is set, the data will be put into the x field of the third register (it means that the compiler must allocate 3 GPRs in all multipass vertex shader modes).

20.3.2 *Pixel shaders*

Formatted: Bullets and Numbering

In the case of pixel shaders, if GEN_INDEX is set, the data will be put in the x field of the 2nd register (I1.x).

Formatted



The Auto Count Value is broadcast to all GPRs. It is loaded into a register wich has its LSBs hardwired to the GPR number (0 thru 63). Then if GEN_INDEX is high, the mux selects the auto-count value and it is loaded into the GPRs to be either used to retrieve data using the TP or sent to the SX for the RB to use it to write the data to memory

Formatted: Bullets and Numbering

~~20.~~21. State management

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

Formatted: Bullets and Numbering

~~21.~~22. 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 20.2~~19.2~~ for details on how to control the interpolation in this mode.

Formatted: Bullets and Numbering

22.1 Vertex indexes imports

In order to import vertex indexes, we have 64x2x96 staging registers. These are loaded one at a time by the VGT block. They are loaded in floating point format and can be transferred in 4 or 8 clocks to the GPRs.

Formatted: Bullets and Numbering

~~22.~~23. Registers

~~22.1~~23.1 Control

~~DYNAMIC_~~REG_DYNAMIC Dynamic allocation (pixel/vertex) of the register file on or off.
~~VERTEX_REG_SIZE What portion of the register file is reserved for vertices (static allocation only)~~
REG_SIZE_PIX~~PIXEL_MIN_SIZE~~ S~~Minimal s~~ize of the register file's pixel portion (minimal size when dynamic allocation turned on) ~~(dynamic only)~~
REG_SIZE_VTX ~~VERTEX_MIN_SIZE Minimal s~~Size of the register file's vertex portion (minimal size when dynamic allocation turned on) ~~(dynamic only)~~
ARBITRATION_~~policy~~POLICY policy of the arbitration between vertexes and pixels
INST_STORE_ALLOC interleaved, separate
~~VERTEX_BASE~~INST_BASE_VTX start point for the vertex instruction store (RT always ends at vertex_base and

Begins at 0)

~~PIXEL_BASE~~INST_BASE_PIX start point for the pixel shader instruction store

~~NO_INTERLEAVE~~ONE_THREAD debug state register. Only allows one program at a time into the GPRs

~~NO_INTERLEAVE_ALU~~ONE_ALU debug state register. Only allows one ALU program at a time to be executed (instead of 2)

INSTRUCTION_INDEX_PORT This is where the CP puts the base address of the instruction writes and type (auto-incremented on reads/writes)

INSTRUCTION_DATA This is where the CP puts the actual data going to the instruction memory

CONSTANT_DATA This is where the CP puts constant data (32 bits)

CONSTANT_ADDR This is where the CP puts the logical constant address (9 bits)

INSTRUCTION_INDEX_PORT_RT This is where the CP puts the base address of the instruction writes and type for Real Time (auto-incremented on reads/writes)

Formatted

INSTRUCTION_DATA_RT This is where the CP puts the actual data going to the instruction memory for Real Time

CONSTANT_DATA_RT This is where the CP puts constant data for Real Time (32 bits)

CONSTANT_ADDR_RT This is where the CP puts the logical constant address for Real Time (9 bits)

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 remaping table operates on the rest of the memory

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

~~22.2~~23.2 Context

Formatted: Bullets and Numbering

~~Vshader_fetch[7:0][7:0]~~VS_FETCH_{0…7} eight 8 bit pointers to the location where each clauses control program is located

~~Vshader_alu[7:0][7:0]~~VS_ALU_{0…7} eight 8 bit pointers to the location where each clauses control program is located

PS_FETCH_{0…7} ~~Pshader_fetch[7:0][7:0]~~ eight 8 bit pointers to the location where each clauses control program is located

PS_ALU_{0…7} ~~Pshader_alu[7:0][7:0]~~ eight 8 bit pointers to the location where each clauses control program is located

~~PSHADER~~PS_BASE base pointer for the pixel shader in the instruction store

~~VSHADER~~VS_BASE base pointer for the vertex shader in the instruction store

~~Vshader_cntl_size~~VS_CF_SIZE size of the vertex shader (# of instructions in control program/2)

~~Pshader_cntl_size~~PS_CF_SIZE size of the pixel shader (# of instructions in control program/2)

~~Pshader_size~~PS_SIZE size of the pixel shader (cntl+instructions)

~~Vshader_size~~VS_SIZE size of the vertex shader (cntl+instructions)

~~REG_ALLOC_PIX~~PS_NUM_REG number of registers to allocate for pixel shader programs

~~REG_ALLOC_VERT~~VS_NUM_REG number of registers to allocate for vertex shader programs

~~FLAT_GOUR[0…15]~~PARAM_SHADE One 16 bit register specifying which parameters are to be gouraud shaded (0 = flat, 1 = gouraud)

~~CYL_WRAP[0…63]~~ PARAM_WRAP 64 bits: for which parameters (and channels (xyzw)) do we do the cyl wrapping (0=linear, 1=cylindrical).

~~P_export_mode~~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.

~~vshader_export_mask~~VS_EXPORT_MASK which of the last 6 ALU clauses is exporting (multipass only)

~~vshader_export_mode~~VS_EXPORT_MODE 0: position (1 vector), 1: position (2 vectors), 3:multipass

~~vshader_export_count[6]~~VS_EXPORT_COUNT_{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)

~~Control_Flow~~ ~~24 Dwords that contain the control flow constants.~~

PARAM_~~GEN_T~~ ~~Max Value interpolated in the T coordinate field (sprites)~~

~~GEN_S~~ ~~Max Value interpolated in the S coordinate field (sprites)~~

GEN_I0 Do we overwrite or not the parameter 0 with XY data and generated T and S values
GEN_INDEX Auto generates an address from 0 to XX. Puts the results into R1 for pixel shaders and R3 for vertex shaders
CONST_BASE_VTX~~VTX_CST_BASE~~ (9 bits) Logical Base address for the constants of the Vertex shader
~~PIX_CST_BASE~~CONST_BASE_PIX (9 bits) Logical Base address for the constants of the Pixel shader
~~PIX_CST_SIZE~~CONST_SIZE_PIX (8 bits) Size of the logical constant store for pixel shaders
~~VTX_CST_SIZE~~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)

Formatted: Bullets and Numbering

~~23.~~24. DEBUG registers

~~23.1~~24.1 Context

DB_PROB_ADDR instruction address where the first problem occurred
DB_PROB_COUNT number of problems encountered during the execution of the program
~~Count~~DB_INST_COUNT instruction counter for debug method 2
~~Addr~~DB_BREAK_ADDR break address for method number 2
DB_CLAUSE_MODE_ALU_{0…7}~~Clause_mode[3]~~ 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)

~~NO_PRED_OPTIMIZE turns off the predicate bit optimization (conditional_execute_predicates is always executed.~~

Formatted: Bullets and Numbering

~~24.~~25. Interfaces

~~24.1~~25.1 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.

Formatted: Bullets and Numbering

~~24.1.1~~25.1.1 SC to SP : IJ bus

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

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

~~24.1.2~~25.1.2 SC to S~~E~~Q : IJ Control bus

Formatted: Bullets and Numbering

This is the control information sent to the sequencer in order to control the IJ fifos and all other information needed to execute a shader program on the sent pixels. This information is sent over 2 clocks, if SENDXY is asserted the next control packet is going to be ignored and XY information is going to be sent on the IJ bus (for the quads that where just sent).

Name	Direction	Bits	Description
SC_SQ_q_wr_mask	SC→SQ	4	Quad Write mask left to right
SC_SQ_lod_correct	SC→SQ	24	LOD correction per quad (6 bits per quad)
SC_SQ_flat_vertex	SC→SQ	2	Provoking vertex for flat shading
SC_SQ_param_ptr0	SC→SQ	11	P Store pointer for vertex 0
SC_SQ_param_ptr1	SC→SQ	11	P Store pointer for vertex 1
SC_SQ_param_ptr2	SC→SQ	11	P Store pointer for vertex 2
SC_SQ_end_of_vect	SC→SQ	1	End of the vector
SC_SQ_store_dealloc	SC→SQ	1	Deallocation token for the P Store
SC_SQ_state	SC→SQ	3	State/constant pointer (6*3+3)
SC_SQ_valid_pixel	SC→SQ	16	Valid bits for all pixels
SC_SQ_null_prim	SC→SQ	1	Null Primitive (for PC deallocation purposes)
SC_SQ_end_of_prim	SC→SQ	1	End Of the primitive
SC_SQ_fbface	SC→SQ	1	Front face = 1, back face = 0
SC_SQ_send_xy	SC→SQ	1	Sending XY information [XY information is going to be sent on the next clock]
SC_SQ_prim_type	SC→SQ	3	Real time command need to load tex cords from alternate buffer. Line AA, Point AA and Sprite reads their parameters from GEN_T and GEN_S registers. 000 : Normal 011 : Real Time 100 : Line AA 101 : Point AA 110 : Sprite
SC_SQ_new_vector	SC→SQ	1	This primitive comes from a new vector of vertices. Make sure that the corresponding vertex shader has finished before starting the group of pixels.
SC_SQ_RTRn	SQ→SC	1	Stalls the PA in n clocks
SC_SQ_RTS	SC→SQ	1	SC ready to send data

~~24.1.3~~25.1.3 SQ to SP: Interpolator bus

Formatted: Bullets and Numbering

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

25.1.4 SQ to SP: GPR Input Mux select

This interface is synchronized with the Interpolator bus. This controls the input mux to the GPRs. The three types of data are: generated index, Interpolated data, vertex index data (coming from the staging registers).

Name	Direction	Bits	Description
SQ_SPx_data_type	SQ→SPx	2	00: Interpolated data 01: Staging register data 1x: Count
SQ_SPx_index_count	SQ→SPx	12?	Index count, common for all shader pipes
SQ_SPx_stage_addr	SQ→SPx	1	Staging register address 0: First staging register 1: second staging register

25.1.5 SQ to SPx: Parameter cache write control

Name	Direction	Bits	Description
SQ_SPx_pc_wr_addr	SQ→SPx	7	Write address
SQ_SPx_pc_we_addr	SQ→SPx	1	Write Enable
SQ_SPx_pc_phase_mux	SQ→SPx	1	The output selector mux (gpr vs parameter cache)

~~24.1.4~~25.1.6 SQ to SP: Parameter Cache Read control bus

The four following interfaces (SQ→SP, SQ→SX,SP→SX and SX→Interpolators) are all SYNCHRONIZED together.

Name	Direction	Bits	Description
SQ_SPx_ptr0	SQ→SPx	9	Pointer of PC
SQ_SPx_ptr1	SQ→SPx	9	Pointer of PC
SQ_SPx_ptr2	SQ→SPx	9	Pointer of PC
SQ_SP0_read_ena	SQ→SP0	4	Read enables for the 4 memories in the SP0
SQ_SP1_read_ena	SQ→SP1	4	Read enables for the 4 memories in the SP1
SQ_SP2_read_ena	SQ→SP2	4	Read enables for the 4 memories in the SP2
SQ_SP3_read_ena	SQ→SP3	4	Read enables for the 4 memories in the SP3

~~24.1.5~~25.1.7 SQ to SX: Parameter Cache Mux control Bus

Name	Direction	Bits	Description
SQ_SXx_mux0	SQ→SXx	4	Mux control for PC (4 MSbs of Pointer)
SQ_SXx_mux1	SQ→SXx	4	Mux control for PC (4 MSbs of Pointer)
SQ_SXx_mux2	SQ→SXx	4	Mux control for PC (4 MSbs of Pointer)

~~24.1.6~~25.1.8 SP to SX: Parameter data

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

Formatted: Bullets and Numbering

Formatted

Formatted

Formatted

Formatted

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

SP3_SX1_data3	SP3→SX1	128	Parameter data 3

~~24.1.7~~25.1.9 *SX to Interpolators: Parameter Cache Return bus*

Formatted: Bullets and Numbering

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

25.1.10 *SQ to SP0: Staging Register Data*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SP0_vgt_vsisr_data	SQ→SP0	96	Pointers of indexes or HOS surface information
SQ_SP0_vgt_vsisr_double	SQ→SP0	1	0: Normal 96 bits per vert 1: double 192 bits per vert
SQ_SP0_data_valid	SQ→SP0	1	Data is valid

25.1.11 *PA to SQ : Vertex interface*

Formatted: Bullets and Numbering

25.1.11.1 Interface Signal Table

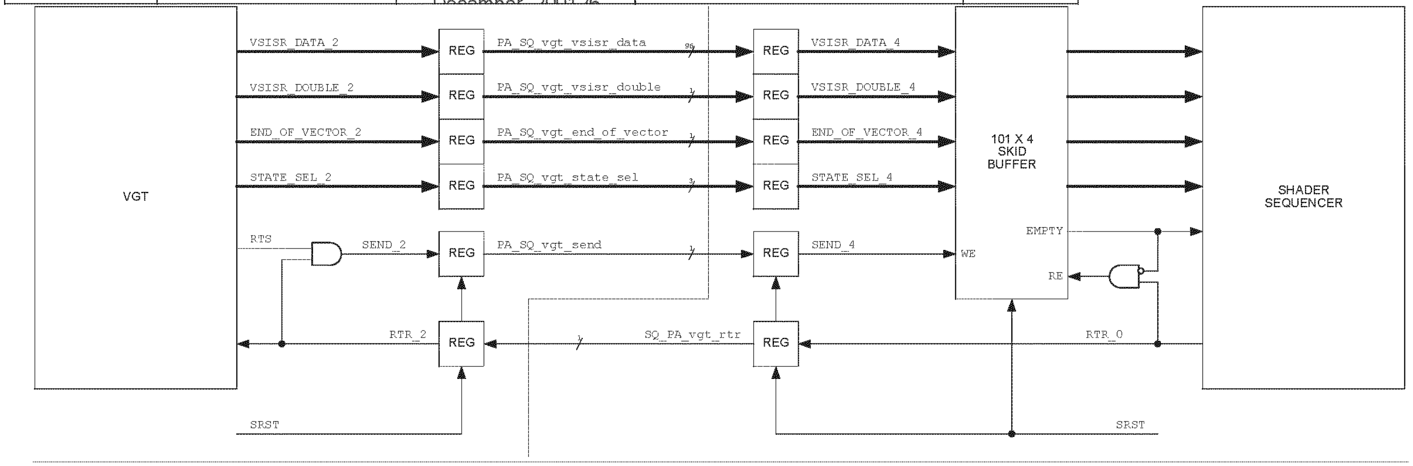
The area difference between the two methods is not sufficient to warrant complicating the interface or the state requirements of the VSISRs. **Therefore, the POR for this interface is that the VGT will transmit the data to the VSISRs (via the Shader Sequencer) in full, 32-bit floating-point format.** 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
PA_SQ_vgt_vsisr_data	96	Pointers of indexes or HOS surface information
PA_SQ_vgt_vsisr_double	1	0: Normal 96 bits per vert 1: double 192 bits per vert
PA_SQ_vgt_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 second vector)
PA_SQ_vgt_state	3	Render State (6*3+3 for constants). This signal is guaranteed to be correct when "PA_SQ_vgt_end_of_vector" is high.
PA_SQ_vgt_send	1	Data on the VGT_SQ is valid receive (see write-up for standard R400 SEND/RTR interface handshaking)
SQ_PA_vgt_rtr	1	Ready to receive (see write-up for standard R400 SEND/RTR interface handshaking)

25.1.11.2 Interface Diagrams

Formatted: Bullets and Numbering

PROTECTIVE ORDER MATERIAL



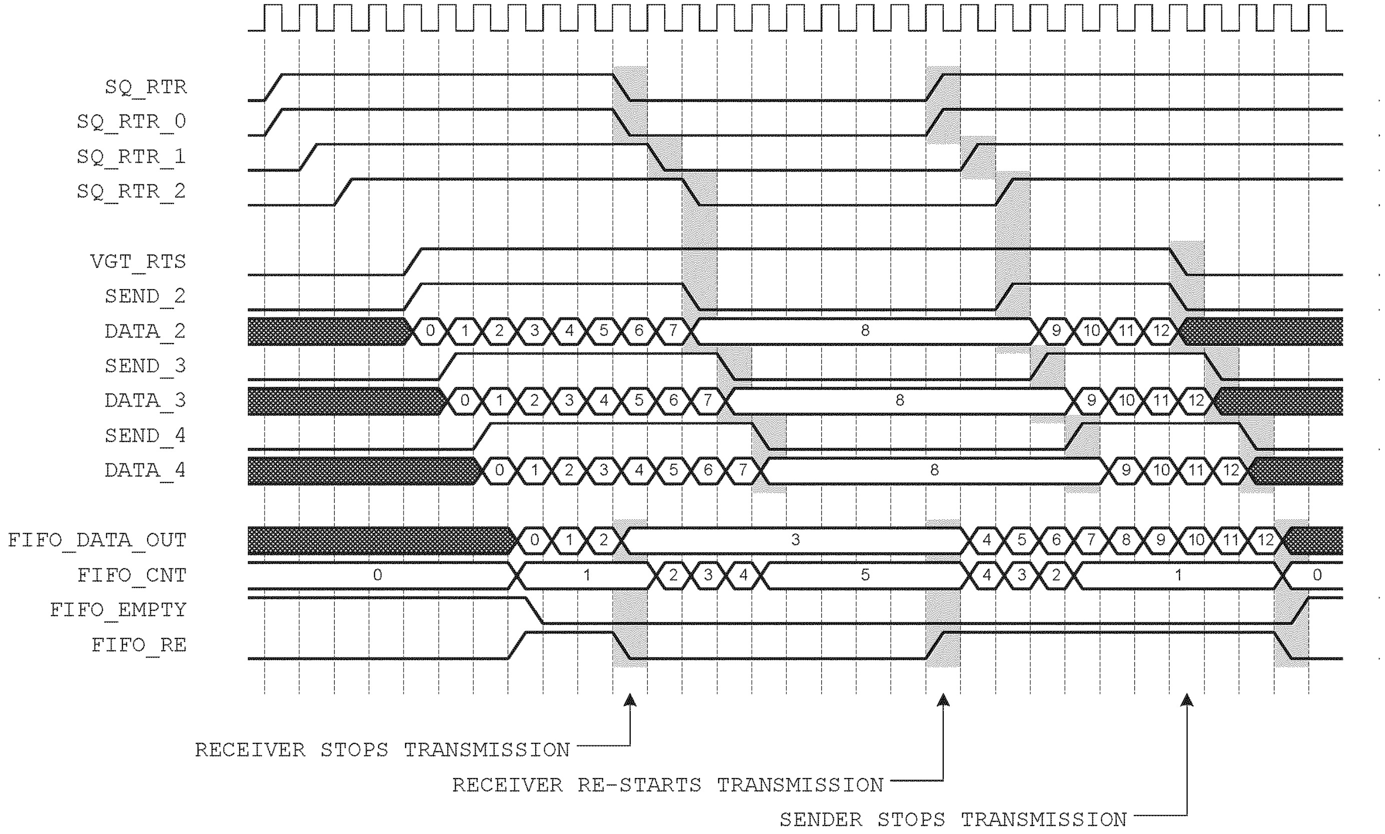


Figure 1. Detailed Logical Diagram for PA_SQ_vgt Interface.

~~24.1.8 VGT to SP0/SQ : Vertex Bus~~

Formatted: Bullets and Numbering

~~24.1.9~~

~~24.1.10~~25.1.12 SEQ to CP: State report

Name	Direction	Bits	Description
SQ_CP_vrtx_ state	SEQ→CP	3	Oldest vertex state still in the pipe
SQ_CP_pix_state	SEQ→CP	3	Oldest pixel state still in the pipe

~~24.1.10~~25.1.13 SP to SX : Pixel/Vertex write to SX

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SP0_SX0_Export_data	SP0→SX0	256	4 pairs of 32 bits channel values
SP0_SX0_Shader_Dest	SP0→SX0	4	Specifies one of the of up to 12 export destinations
SP1_SX1_Export_data	SP1→SX1	256	4 pairs of 32 bits channel values
SP1_SX1_Shader_Dest	SP1→SX1	4	Specifies one of the of up to 12 export destinations
SP2_SX0_Export_data	SP2→SX0	256	4 pairs of 32 bits channel values
SP2_SX0_Shader_Dest	SP2→SX0	4	Specifies one of the of up to 12 export destinations
SP3_SX1_Export_data	SP3→SX1	256	4 pairs of 32 bits channel values
SP3_SX1_Shader_Dest	SP3→SX1	4	Specifies one of the of up to 12 export destinations
SPx_SXx_Shader_Count	SP0→SX0	3	Each set of four pixels or vectors is exported over eight clocks. This field specifies where the SP is in that sequence.
SPx_SXx_Shader_Last	SP0→SX0	1	Asserted on the first shader count of the last export of the clause
SP0_SX0_Shader_PixelValid	SP0→SX0	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SP0_SX0_Shader_WordValid	SP0→SX0	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SP1_SX1_Shader_PixelValid	SP1→SX1	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SP1_SX1_Shader_WordValid	SP1→SX1	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SP2_SX0_Shader_PixelValid	SP2→SX0	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SP2_SX0_Shader_WordValid	SP2→SX0	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SP3_SX1_Shader_PixelValid	SP3→SX1	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SP3_SX1_Shader_WordValid	SP3→SX1	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors

~~24.1.11~~25.1.14 SQ to SX: Control bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description

SQ_SXx_exp_Pixel	SQ→SXx	1	1: Pixel 0: Vertex
SQ_SXx_exp_start	SQ→SXx	1	Raised to indicate that the SQ is starting an export
SQ_SXx_exp_Clause	SQ→SXx	3	Clause number, which is needed for vertex clauses
SQ_SXx_exp_State	SQ→SXx	3	State ID, which is needed for vertex clauses

These fields are sent synchronously with SP export data, described in SP0→SX0 interface
{ISSUE: Where are the PC pointers}

~~24.1.12~~25.1.15 *SX to SQ : Output file control*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SXx_SQ_Export_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_Export_Position	SXx→SQ	1	Specifies whether there is room for another position.
SXx_SQ_Export_Buffer	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

~~24.1.13~~25.1.16 *Shader Engine to Fetch Unit Bus*

Formatted: Bullets and Numbering

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

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

Name	Direction	Bits	Description
SP0_TP0_fetch_addr	SP0->TP0	512	4 Fetch Addresses read from the Register file
TP0_SP0_data	TP0→SP0	512	4 texture results
SP1_TP1_fetch_addr	SP1->TP1	512	4 Fetch Addresses read from the Register file
TP1_SP1_data	TP1→SP1	512	4 texture results
SP2_TP2_fetch_addr	SP2->TP2	512	4 Fetch Addresses read from the Register file
TP2_SP2_data	TP2→SP2	512	4 texture results
SP3_TP3_fetch_addr	SP3->TP3	512	4 Fetch Addresses read from the Register file
TP3_SP3_data	TP3→SP3	512	4 texture results
TPx_SPx_gpr_dst	TPx→SPx	7	Write address into the gprs
TPx_SPx_gpr_cmask	TPx→SPx	4	Channel mask

~~24.1.14~~25.1.17 *Sequencer to Fetch Unit bus*

Formatted: Bullets and Numbering

Once every clock, the fetch unit sends to the sequencer on which clause it is now working and if the data in the registers is ready or not. This way the sequencer can update the fetch counters 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	Direction	Bits	Description
TPx_SQ_data_rdy	TPx→ SQ	1	Data ready
TPx_SQ_clause_num	TPx→ SQ	3	Clause number
SQ_TPx_const	SQ→TPx	64	Fetch state sent over 4 clocks
SQ_TPx_instuct	SQ→TPx	24	Fetch instruction sent over 4 clocks
SQ_TPx_end_of_clause	SQ→TPx	1	Last instruction of the clause

SQ_TPx_phase	SQ→TPx	2	Write phase signal
SQ_TP0_lod_correct	SQ→TP0	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP0_pmask	SQ→TP0	4	Pixel mask 1 bit per pixel
SQ_TP1_lod_correct	SQ→TP1	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP1_pmask	SQ→TP1	4	Pixel mask 1 bit per pixel
SQ_TP2_lod_correct	SQ→TP2	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP2_pmask	SQ→TP2	4	Pixel mask 1 bit per pixel
SQ_TP3_lod_correct	SQ→TP3	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP3_pmask	SQ→TP3	4	Pixel mask 1 bit per pixel
SQ_TPx_clause_num	SQ→TPx	3	Clause number
SQ_TPx_write_gpr_index	SQ->TPx	7	Index into Register file for write of returned Fetch Data

~~24.1.15~~25.1.18 *Sequencer to SP: GPR control*

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_re_addr	SQ→SPx	1	Read Enable
SQ_SPx_gpr_we_addr	SQ→SPx	1	Write Enable
SQ_SPx_gpr_phase_mux	SQ→SPx	2	The phase mux
SQ_SPx_gpr_channel_mask	SQ→SPx	4	The channel mask
SQ_SP0_gpr_pixel_mask	SQ→SP0	4	The pixel mask
SQ_SP1_gpr_pixel_mask	SQ→SP1	4	The pixel mask
SQ_SP2_gpr_pixel_mask	SQ→SP2	4	The pixel mask
SQ_SP3_gpr_pixel_mask	SQ→SP3	4	The pixel mask

~~24.1.16 *Sequencer to SPx: Parameter cache write control*~~

Formatted: Bullets and Numbering

~~24.1.17~~25.1.19 *Sequencer to SPx: Instructions*

Name	Direction	Bits	Description
SQ_SPx_instruct_start	SQ→SPx	1	Instruction start
SQ_SP_instruct	SQ→SPx	20	Instruction sent over 4 clocks
SQ_SPx_stall	SQ→SPx	1	Stall signal
SQ_SPx_Shader_Count	SQ→SPx	3	Each set of four pixels or vectors is exported over eight clocks. This field specifies where the SP is in that sequence.
SQ_SPx_Shader_Last	SQ→SPx	1	Asserted on the first shader count of the last export of the clause
SQ_SP0_Shader_PixelValid	SQ→SP0	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP0_Shader_WordValid	SQ→SP0	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SQ_SP1_Shader_PixelValid	SQ→SP1	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP1_Shader_WordValid	SQ→SP1	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SQ_SP2_Shader_PixelValid	SQ→SP2	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP2_Shader_WordValid	SQ→SP2	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16

			pixels or vectors
SQ_SP3_Shader_PixelValid	SQ→SP3	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP3_Shader_WordValid	SQ→SP3	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors

~~24.1.18~~25.1.20 *SP to Sequencer: Constant address load*

Formatted: Bullets and Numbering

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

~~24.1.19~~25.1.21 *Sequencer to SPx: constant broadcast*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_constant	SQ→SPx	128	Constant broadcast

~~24.1.20~~25.1.22 *SP0 to Sequencer: 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

~~24.1.21~~25.1.23 *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)

~~24.1.22~~25.1.24 *CP to SQ: RBBM bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
rbbm_we	CP→SQ	1	Write Enable
rbbm_a	CP→SQ	18	Address -- Upper Extent is TBD
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

Formatted: Bullets and Numbering

~~25.~~26. Examples of program executions

~~25.1.1~~26.1.1 *Sequencer Control of a Vector of Vertices*

1. PA sends a vector of 64 vertices (actually vertex indices – 32 bits/index for 2048 bit total) to the RE's Vertex FIFO
   - state pointer as well as tag into position cache is sent along with vertices
   - space was allocated in the position cache for transformed position before the vector was sent
   - **also before the vector is sent to the RE, the CP has loaded the global instruction store with the vertex shader program (using the MH?)**
   - The vertex program is assumed to be loaded when we receive the vertex vector.
     - the SEQ then accesses the IS base for this shader using the local state pointer (provided to all sequencers by the RBBM when the CP is done loading the program)
2. SEQ arbitrates between the Pixel FIFO and the Vertex FIFO – basically the Vertex FIFO always has priority
   - at this point the vector is removed from the Vertex FIFO
   - the arbiter is not going to select a vector to be transformed if the parameter cache is full unless the pipe as nothing else to do (ie no pixels are in the pixel fifo).
3. SEQ allocates space in the SP register file for index data plus GPRs used by the program
   - the number of GPRs required by the program is stored in a local state register, which is accessed using the state pointer that came down with the vertices
   - SEQ will not send vertex data until space in the register file has been allocated
4. SEQ sends the vector to the SP register file over the RE_SP interface (which has a bandwidth of 2048 bits/cycle)
   - the 64 vertex indices are sent to the 64 register files over 4 cycles
     - RF0 of SU0, SU1, SU2, and SU3 is written the first cycle
     - RF1 of SU0, SU1, SU2, and SU3 is written the second cycle
     - RF2 of SU0, SU1, SU2, and SU3 is written the third cycle
     - RF3 of SU0, SU1, SU2, and SU3 is written the fourth cycle
   - the index is written to the least significant 32 bits **(floating point format?) (what about compound indices)** of the 128-bit location within the register file (w); the remaining data bits are set to zero (x, y, z)
5. SEQ constructs a control packet for the vector and sends it to the first reservation station (the FIFO in front of fetch state machine 0, or TSM0 FIFO)
   - the control packet contains the state pointer, the tag to the position cache and a register file base pointer.
6. TSM0 accepts the control packet and fetches the instructions for fetch clause 0 from the global instruction store
   - TSM0 was first selected by the TSM arbiter before it could start
7. all instructions of fetch clause 0 are issued by TSM0
8. the control packet is passed to the next reservation station (the FIFO in front of ALU state machine 0, or ASM0 FIFO)
   - TSM0 does not wait for requests made to the Fetch Unit to complete; it passes the register file write index for the fetch data to the TU, which will write the data to the RF as it is received
   - once the TU has written all the data to the register files, it increments a counter that is associated with ASM0 FIFO; a count greater than zero indicates that the ALU state machine can go ahead start to execute the ALU clause
9. ASM0 accepts the control packet (after being selected by the ASM arbiter) and gets the instructions for ALU clause 0 from the global instruction store
10. all instructions of ALU clause 0 are issued by ASM0, then the control packet is passed to the next reservation station (the FIFO in front of fetch state machine 1, or TSM1 FIFO)
11. the control packet continues to travel down the path of reservation stations until all clauses have been executed
    - position can be exported in ALU clause 3 (or 4?); the data (and the tag) is sent over a position bus (which is shared with all four shader pipes) back to the PA's position cache
    - A parameter cache pointer is also sent along with the position data. This tells to the PA where the data is going to be in the parameter cache.
      - there is a position export FIFO in the SP that buffers position data before it gets sent back to the PA

    - the ASM arbiter will prevent a packet from starting an exporting clause if the position export FIFO is full
- parameter data is exported in clause 7 (as well as position data if it was not exported earlier)
    - parameter data is sent to the Parameter Cache over a dedicated bus
    - the SEQ allocates storage in the Parameter Cache, and the SEQ deallocates that space when there is no longer a need for the parameters (it is told by the PA when using a token).
    - the ASM arbiter will prevent a packet from starting on ASM7 if the parameter cache (or the position buffer if position is being exported) is full

12. after the shader program has completed, the SEQ will free up the GPRs so that they can be used by another shader program

~~25.1.2~~26.1.2 *Sequencer Control of a Vector of Pixels*

Formatted: Bullets and Numbering

1. **As with vertex shader programs, pixel shaders are loaded into the global instruction store by the CP**
    - At this point it is assumed that the pixel program is loaded into the instruction store and thus ready to be read.
2. the RE's Pixel FIFO is loaded with the barycentric coordinates for pixel quads by the detailed walker
    - the state pointer and the LOD correction bits are also placed in the Pixel FIFO
    - the Pixel FIFO is wide enough to source four quad's worth of barycentrics per cycle
3. SEQ arbitrates between Pixel FIFO and Vertex FIFO – when there are no vertices pending OR there is no space left in the register files for vertices, the Pixel FIFO is selected
4. SEQ allocates space in the SP register file for all the GPRs used by the program
    - the number of GPRs required by the program is stored in a local state register, which is accessed using the state pointer
    - SEQ will not allow interpolated data to be sent to the shader until space in the register file has been allocated
5. SEQ controls the transfer of interpolated data to the SP register file over the RE_SP interface (which has a bandwidth of 2048 bits/cycle). See interpolated data bus diagrams for details.
6. SEQ constructs a control packet for the vector and sends it to the first reservation station (the FIFO in front of fetch state machine 0, or TSM0 FIFO)
    - note that there is a separate set of reservation stations/arbiters/state machines for vertices and for pixels
    - the control packet contains the state pointer, the register file base pointer, and the LOD correction bits
    - all other information (such as quad address for example) travels in a separate FIFO
7. TSM0 accepts the control packet and fetches the instructions for fetch clause 0 from the global instruction store
    - TSM0 was first selected by the TSM arbiter before it could start
8. all instructions of fetch clause 0 are issued by TSM0
9. the control packet is passed to the next reservation station (the FIFO in front of ALU state machine 0, or ASM0 FIFO)
    - TSM0 does not wait for fetch requests made to the Fetch Unit to complete; it passes the register file write index for the fetch data to the TU, which will write the data to the RF as it is received
    - once the TU has written all the data for a particular clause to the register files, it increments a counter that is associated with the ASM0 FIFO; a count greater than zero indicates that the ALU state machine can go ahead and pop the FIFO and start to execute the ALU clause
10. ASM0 accepts the control packet (after being selected by the ASM arbiter) and gets the instructions for ALU clause 0 from the global instruction store
11. all instructions of ALU clause 0 are issued by ASM0, then the control packet is passed to the next reservation station (the FIFO in front of fetch state machine 1, or TSM1 FIFO)
12. the control packet continues to travel down the path of reservation stations until all clauses have been executed
    - pixel data is exported in the last ALU clause (clause 7)
        - it is sent to an output FIFO where it will be picked up by the render backend
        - the ASM arbiter will prevent a packet from starting on ASM7 if the output FIFO is full
13. after the shader program has completed, the SEQ will free up the GPRs so that they can be used by another shader program

~~25.1.3~~26.1.3 *Notes*

Formatted: Bullets and Numbering

14. The state machines and arbiters will operate ahead of time so that they will be able to immediately start the real threads or stall.
15. The register file base pointer for a vector needs to travel with the vector through the reservation stations, but the instruction store base pointer does not – this is because the RF pointer is different for all threads, but the IS pointer is only different for each state and thus can be accessed via the state pointer
16. Waterfalling still needs to be specked out.

~~26.~~27. Open issues

Formatted: Bullets and Numbering

There is currently an issue with constants. If the constants are not the same for the whole vector of vertices, we don't have the bandwidth from the fetch store to feed the ALUs. Two solutions exists for this problem:

1) Let the compiler handle the case and put those instructions in a fetch clause so we can use the bandwidth there to operate. This requires a significant amount of temporary storage in the register store.
2) Waterfall down the pipe allowing only at a given time the vertices having the same constants to operate in parallel. This might in the worst case slow us down by a factor of 16.

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

Saving power?

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 2015~~11~~ ~~December, 20016~~	GEN-CXXXXX-REVA	1 of 48

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SQ

Version 1.5~~4~~

Overview: This is an architectural specification for the R400 Sequencer block (SEQ). It provides an overview of the required capabilities and expected uses of the block. It also describes the block interfaces, internal sub-blocks, and provides internal state diagrams.

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

APPROVALS	
Name/Dept	Signature/Date

Remarks:

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

Table Of Contents

Revision Changes:

Rev 0.1 (Laurent Lefebvre) Date: May 7, 2001	First draft.
Rev 0.2 (Laurent Lefebvre) Date : July 9, 2001	Changed the interfaces to reflect the changes in the SP. Added some details in the arbitration section.
Rev 0.3 (Laurent Lefebvre) Date : August 6, 2001	Reviewed the Sequencer spec after the meeting on August 3, 2001.
Rev 0.4 (Laurent Lefebvre) Date : August 24, 2001	Added the dynamic allocation method for register file and an example (written in part by Vic) of the flow of pixels/vertices in the sequencer.
Rev 0.5 (Laurent Lefebvre) Date : September 7, 2001	Added timing diagrams (Vic)
Rev 0.6 (Laurent Lefebvre) Date : September 24, 2001	Changed the spec to reflect the new R400 architecture. Added interfaces.
Rev 0.7 (Laurent Lefebvre) Date : October 5, 2001	Added constant store management, instruction store management, control flow management and data dependant predication.
Rev 0.8 (Laurent Lefebvre) Date : October 8, 2001	Changed the control flow method to be more flexible. Also updated the external interfaces.
Rev 0.9 (Laurent Lefebvre) Date : October 17, 2001	Incorporated changes made in the 10/18/01 control flow meeting. Added a NOP instruction, removed the conditional_execute_or_jump. Added debug ~~registers~~registers.
Rev 1.0 (Laurent Lefebvre) Date : October 19, 2001	Refined interfaces to RB. Added state ~~registers~~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 ~~registers~~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.

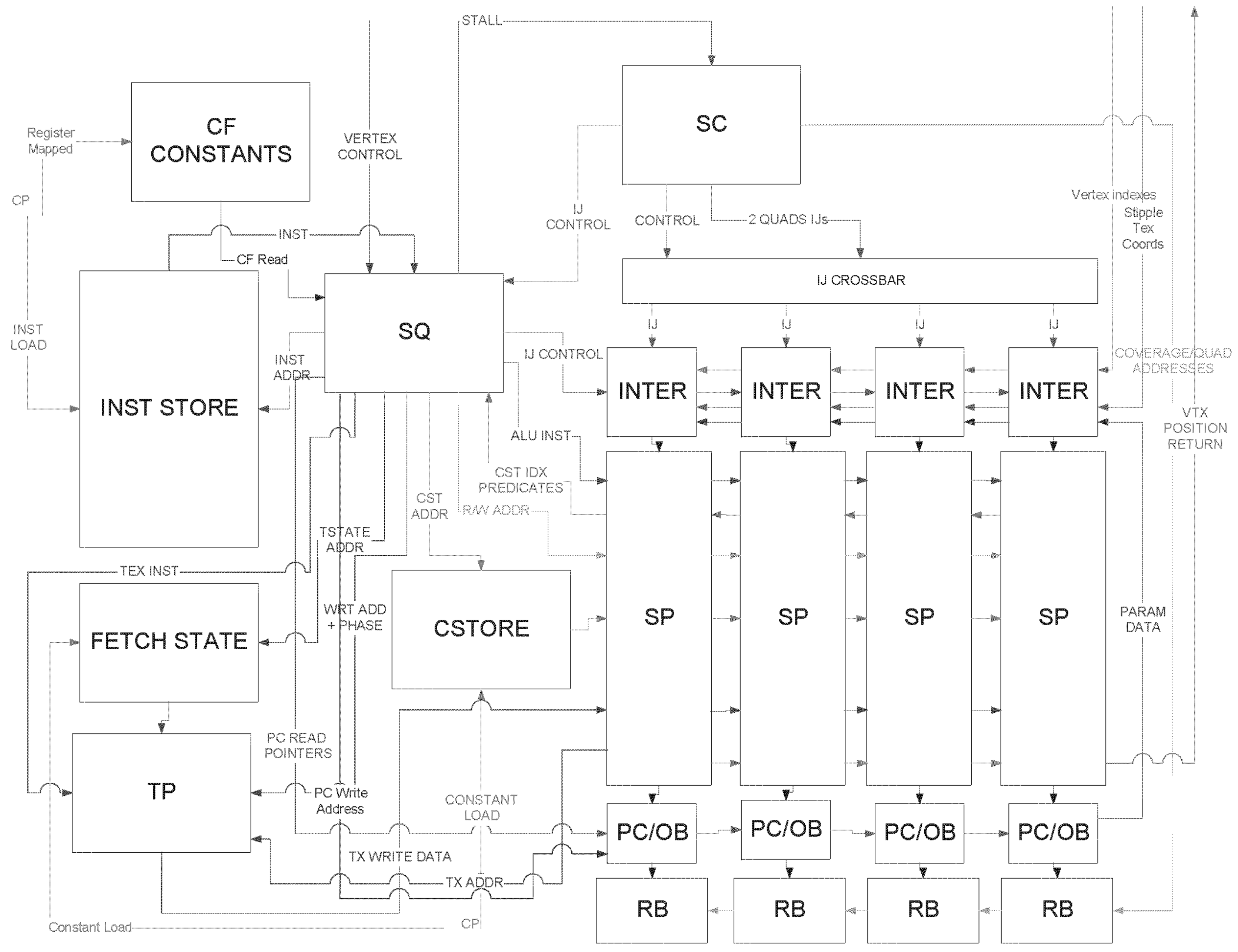
1. Overview

The sequencer is based on the R300 design. It chooses two ALU clauses and a fetch clause to execute, and executes all of the instructions in a clause before looking for a new clause of the same type. Two ALU clauses 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. 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.

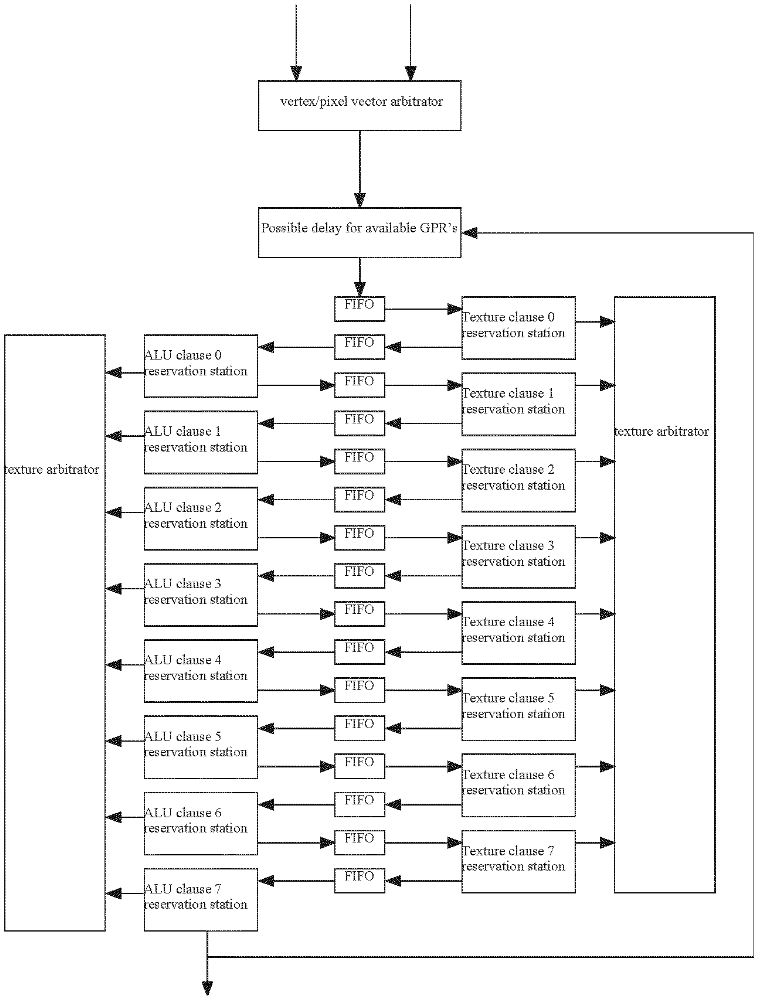
To support the shader pipe the sequencer also contains the shader instruction cache, constant store, control flow constants and texture state. The four shader pipes also execute the same instruction thus there is only one sequencer for the whole chip.

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.


STALL
SC
CF CONSTANTS
Register Mapped
VERTEX CONTROL
CP
INST
CF Read
IJ CONTROL
CONTROL
2 QUADS IJs
Vertex indexes
Stipple Tex Coords
IJ CROSSBAR
SQ
INST LOAD
INST STORE
INST ADDR
IJ
IJ
IJ
IJ
IJ CONTROL
INTER
INTER
INTER
INTER
COVERAGE/QUAD ADDRESSES
VTX POSITION RETURN
ALU INST
CST IDX PREDICATES
R/W ADDR
CST ADDR
TSTATE ADDR
TEX INST
WRT ADD + PHASE
CSTORE
FETCH STATE
SP
SP
SP
SP
PARAM DATA
PC READ POINTERS
TP
PC Write Address
CONSTANT LOAD
TX WRITE DATA
TX ADDR
PC/OB
PC/OB
PC/OB
PC/OB
RB
RB
RB
RB
Constant Load
CP

1.1 Top Level Block Diagram


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

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 ~~registers~~GPRs to store the interpolated values and temporaries. Following this, the barycentric coordinates (and

XY screen position if needed) are sent to the interpolator ~~buffers~~ which ~~are going to~~will use ~~these barycentric coordinates~~them to interpolate the parameters and place the ~~interpolated value~~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 ~~texture~~ fetch request to the TP and corresponding ~~register~~ 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 ~~register~~ 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 ~~1~~ 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 ~~(3 cycles later)~~ 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.

{ISSUE: How do we handle parameter cache pointers (computed, semi-computed or not computed)?}

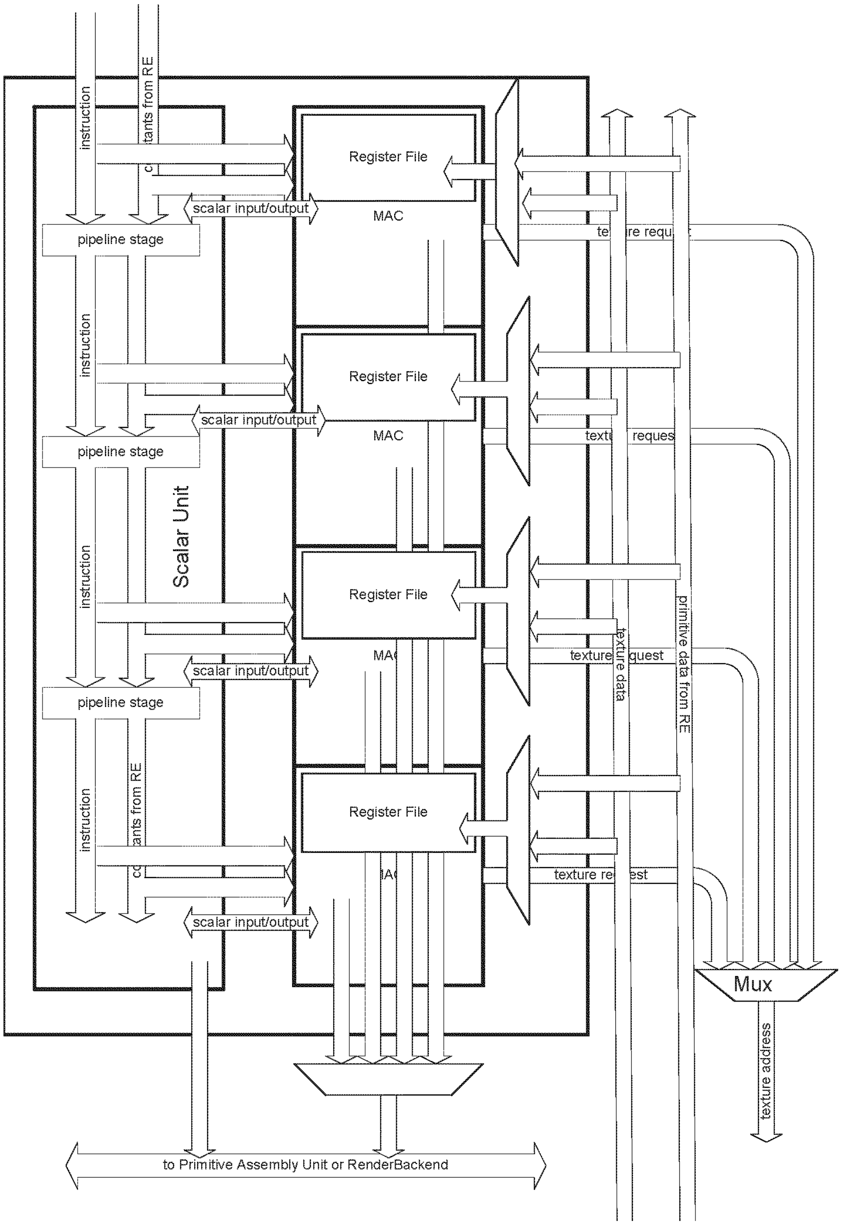
A special case is for multipass vertex shaders, which can export 12 parameters per last 6 clauses to the output buffer. If the output buffer is full or doesn't have enough space the sequencer will prevent such a vertex group to enter an exporting clause.

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

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

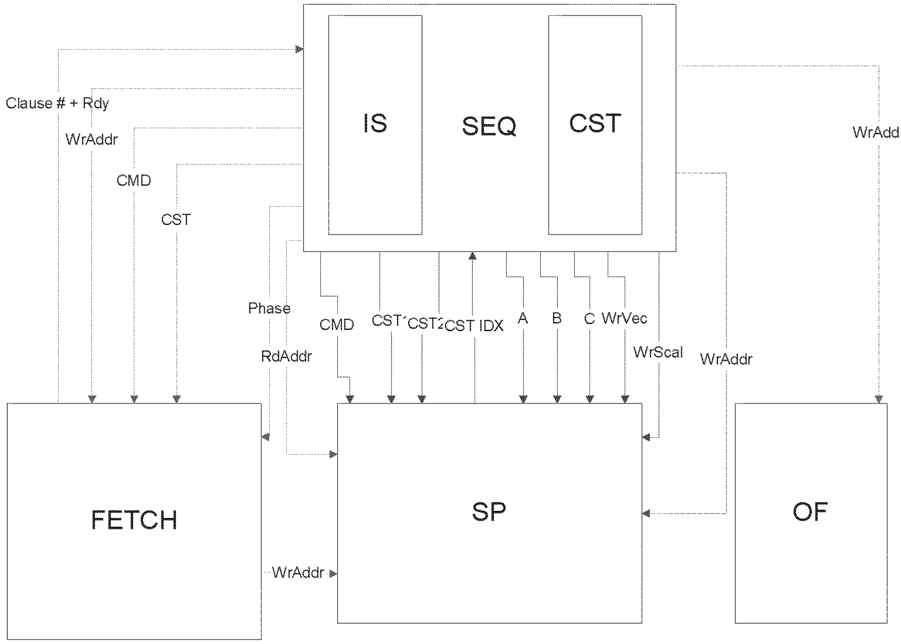
Only ~~two~~ one pair of interleaved ALU state machines may have access to the register file address bus or the instruction decode bus at one time. Similarly, only one fetch state machine may have access to the register file address bus at one time. Arbitration is performed by three arbiter blocks (two 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.

1.2 Data Flow graph



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

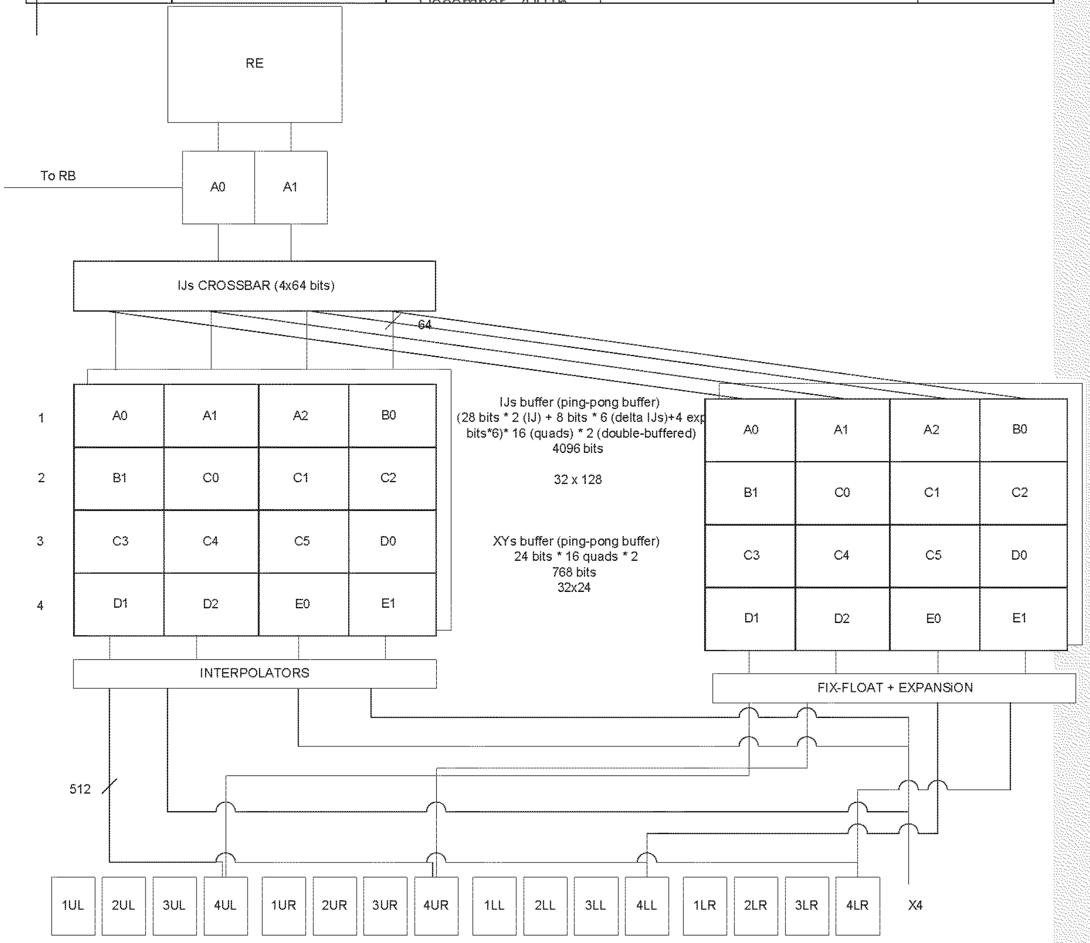
1.3 Control Graph



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.


RE
To RB
A0
A1
IJs CROSSBAR (4x64 bits)
64
1
2
3
4
A0 A1 A2 B0
B1 C0 C1 C2
C3 C4 C5 D0
D1 D2 E0 E1
IJs buffer (ping-pong buffer)
(28 bits * 2 (IJ) + 8 bits * 6 (delta IJs)+4 exp bits*6)* 16 (quads) * 2 (double-buffered)
4096 bits
32 x 128
XYs buffer (ping-pong buffer)
24 bits * 16 quads * 2
768 bits
32x24
A0 A1 A2 B0
B1 C0 C1 C2
C3 C4 C5 D0
D1 D2 E0 E1
INTERPOLATORS
FIX-FLOAT + EXPANSION
512
1UL
2UL
3UL
4UL
1UR
2UR
3UR
4UR
1LL
2LL
3LL
4LL
1LR
2LR
3LR
4LR
X4

	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		

Above is an example of a tile ~~we~~ 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 ~~2~~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 ~~register~~ 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.

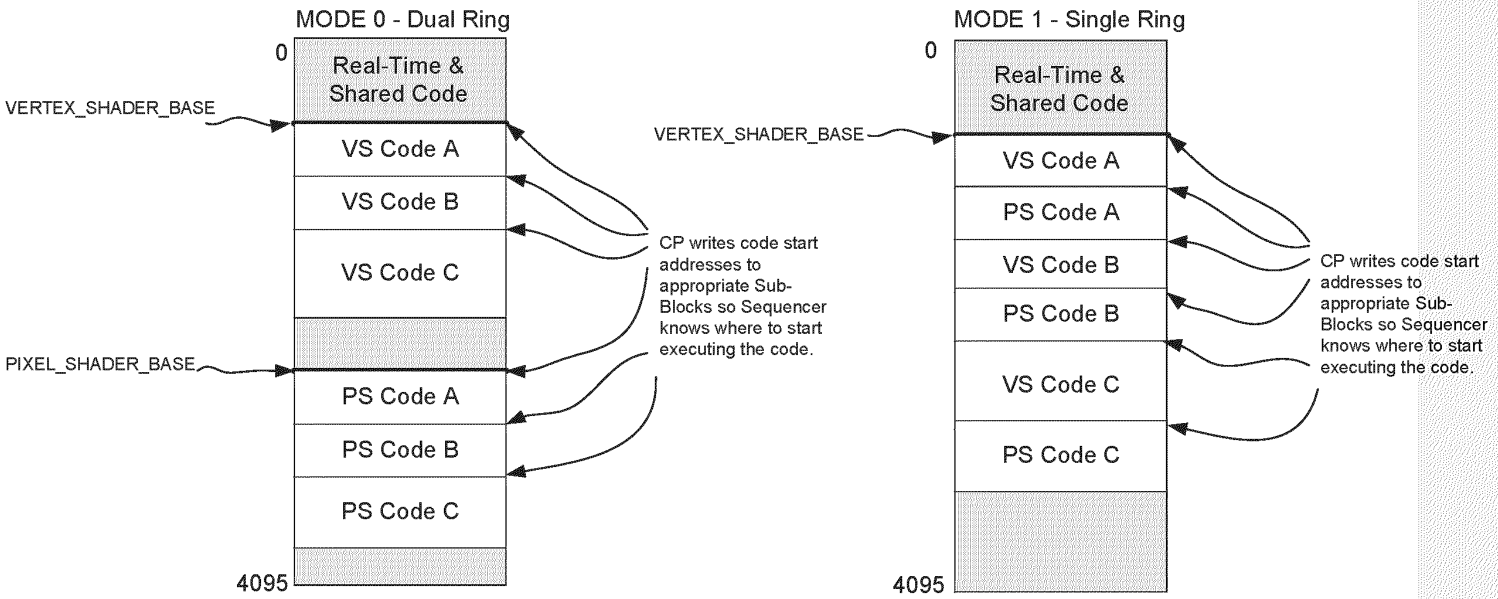
The instruction store is loaded by the CP thru the INST_DATA, INST_INDEX_PORT control ~~registers~~register. The INST_INDEX_PORT is auto-incremented on both reads and writes to the INST_DATA register.

The next picture shows the various modes the CP can load the memory. The Sequencer has to keep track of the loading modes in order to wrap around the correct boundaries. The MSB of the INST_INDEX_PORT register contains the packet type for the sequencer to know where it must wrap around. The wrap around points are arbitrary and they are specified in the VS_BASE and PIX_BASE ~~registers~~registers.

For the Real time commands the story is quite the same but for some small differences. The CP will use the INST_INDEX_PORT_RT and INST_DATA_RT register pair instead of the regular ones and 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.

R400 CP's Views of Instruction Memory

Updated: 11/14/2001
John A. Carey



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).

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 ~~remaping~~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.

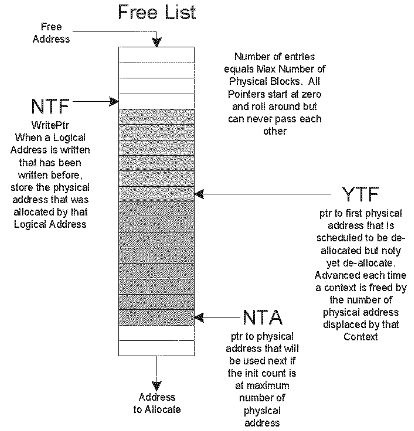
The texture state is also kept in a similar memory. The size of this memory is 192x128. The memory thus holds 128 texture states (192 bits per state). The logical size ~~exposed~~ exposes 32 different states total, which are going to be shared between the pixel and the vertex shader. The size of the ~~remaping~~re-mapping table to for the texture state memory is 16 lines (each line addresses 2 texture state lines in the real memory). The write granularity is 2 texture state lines (or 384 bits). The driver sends 512 bits but the CP ignores the top 128 bits. It thus takes 12 clocks to write the two texture states.

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 state change. Its size is ~~256~~320*32 because it must hold 8 copies of the 32 dwords of control flow constants and the loop construct constants must be aligned.

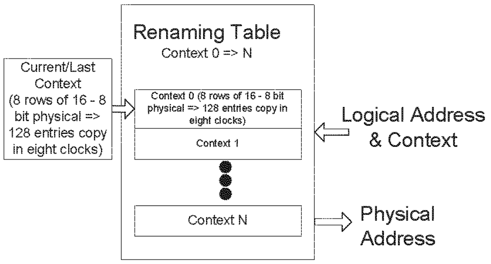
The CP is loading the constant store using the CONST_DATA and CONST_ADDR ~~registers~~registers. It does so by writing to the CONST_ADDR register the logical address for the constant block it wants to update and then writes 16 times to the CONST_DATA register. The CONST_ADDR is auto-incremented on both reads and writes to the CONST_DATA register.

5.2 Management of the ~~remaping~~re-mapping tables

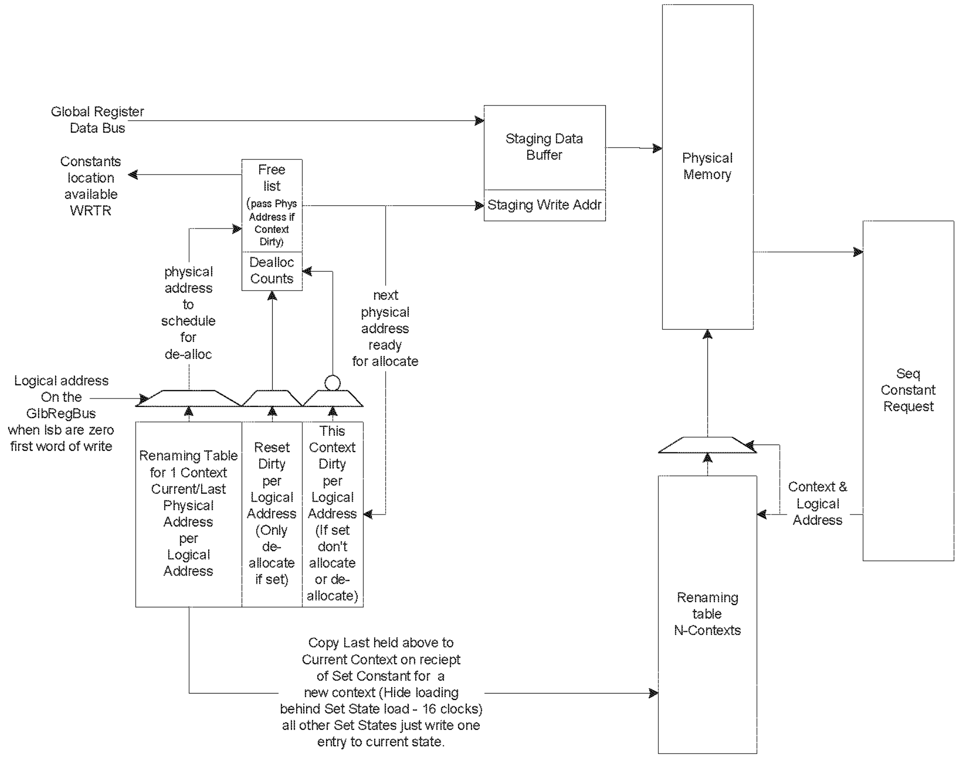
The sequencer is responsible to manage two ~~remaping~~re-mapping tables (one for the constant store and one for the texture state). On a state change (by the driver), the sequencer will broadside copy the contents of its ~~remaping~~re-mapping tables to a new one. We have 8 different ~~remaping~~re-mapping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



Free List
Free Address
Number of entries equals Max Number of Physical Blocks. All Pointers start at zero and roll around but can never pass each other
NTF
WritePtr
When a Logical Address is written that has been written before, store the physical address that was allocated by that Logical Address
YTF
ptr to first physical address that is scheduled to be de-allocated but noty yet de-allocate. Advanced each time a context is freed by the number of physical address displaced by that Context
NTA
ptr to physical address that will be used next if the init count is at maximum number of physical address
Address to Allocate



Renaming Table
Context 0 => N
Current/Last Context (8 rows of 16 - 8 bit physical => 128 entries copy in eight clocks)
Context 0 (8 rows of 16 - 8 bit physical => 128 entries copy in eight clocks)
Context 1
Context N
Logical Address & Context
Physical Address



Global Register Data Bus
Staging Data Buffer
Staging Write Addr
Physical Memory
Constants location available WRTR
Free list (pass Phys Address if Context Dirty)
Dealloc Counts
physical address to schedule for de-alloc
next physical address ready for allocate
Seq Constant Request
Logical address On the GlbRegBus when lsb are zero first word of write
Renaming Table for 1 Context Current/Last Physical Address per Logical Address
Reset Dirty per Logical Address (Only de-allocate if set)
This Context Dirty per Logical Address (If set don't allocate or de-allocate)
Context & Logical Address
Renaming table N-Contexts
Copy Last held above to Current Context on reciept of Set Constant for a new context (Hide loading behind Set State load - 16 clocks) all other Set States just write one entry to current state.

5.2.1 *Dirty bits*

Formatted: Bullets and Numbering

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 when ever 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.

5.2.2 *Free List Block*

Formatted: Bullets and Numbering

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 NTF (Next To Free). 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 YTF (Yet To Free). The YTF pointer will be advanced by the number of address chunks de-allocates when a context finishes. The address between the YTF and NTF cannot be reused because they are still in use. But as soon as the context using then is dismissed the YTF will be advanced.
The third pointer will be called NTA (Next To Allocate). This pointer will point will point to the next address that can be used for allocation as long as the NTA does not equal the YTF and the IFC is at its maximum count.

5.2.3 *De-allocate Block*

Formatted: Bullets and Numbering

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 NTF 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 NTF 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.

5.2.4 *Operation of Incremental model*

Formatted: Bullets and Numbering

The basic operation of the model would start with the NTF, YTF, NTA 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 a set constant comes with a different than last context, 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 NTF pointer location on the free list and the NTF will be incremented. The de-allocation counter for the previous context (zero) 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 NTA pointer if NTA != to YTF .

Formatted: Bullets and Numbering

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 NTF 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 back pressure to the CP when ever he has not free list entries available (counter at max and YTF == NTA). The command stream will keep a count of contexts of constants in use and prevent more than max constants contexts from being sent.
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 YTF pointer. This will make all the physical addresses used by this context available to the NTA 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.

5.3 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
```

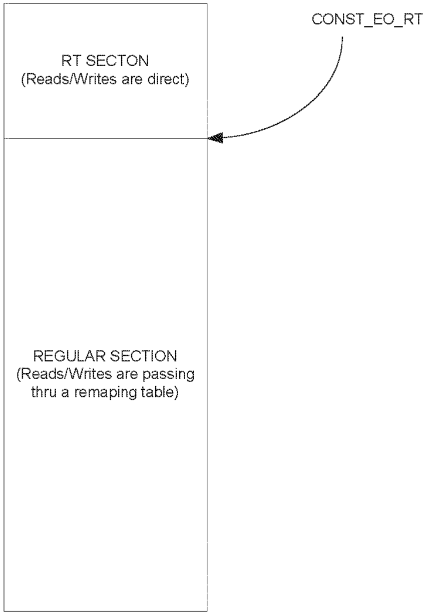
Note that we don't really care about what is in the brackets because we use the state from the MOVA instruction. R2.X is just written again for the sake of simplicity and coherency.

~~5.4~~The storage needed in the sequencer in order to support this feature is 2*64*9 bits = 1152 bits.

Formatted: Bullets and Numbering

~~5.5~~5.4 Real Time Commands

The real time commands constants are written by the CP using the CONST_DATA_RT and CONST_ADDR_RT ~~registers~~GPRs. 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 ~~remaping~~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.


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

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

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 eleven basic instructions:

Execute
Conditional_execute
Conditional_Execute_Predicates
Conditional_jump
Call
Return
Loop_start
Loop_end
End_of_clause
Conditional_End_of_clause
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.
Call jumps to an address and pushes the IP counter on the stack. On the return instruction, the IP is popped from the stack.
Conditional_execute_or_Jump executes a block of instructions or jumps to an address is the condition is not met.
Conditional_execute_Predicates executes a block of instructions if all bits in the predicate vectors meet the condition.
End_of_clause marks the end of a clause.
Conditional_End_of_clause marks the end of a clause if the condition is met.
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~~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~~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.

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… 42	41 … 24	23 … 12	11 … 0
Addressing	00001	RESERVED	Instruction count	Exec Address

Execute up to 4k instructions at the specified address in the instruction memory.

NOP		
47	46 … 42	41 … 0

Addressing	00010	RESERVED

This is a regular NOP.

Conditio~~n~~nal_Execute							
47	46 … 42	41	40 … 33	32	31 … 24	23 … 12	11 … 0
Addressing	00011	RESERVED	Boolean address	Condition	RESERVED	Instruction count	Exec Address

If the specified ~~boolean~~Boolean (8 bits can address 256 ~~booleans~~Booleans) meets the specified condition then execute the specified instructions (up to 4k instructions)

Conditio~~n~~nal_Execute_Predicates							
47	46 … 42	41 … 35	34 … 33	32	31 … 24	23 … 12	11 … 0
Addressing	00100	RESERVED	Predicate vector	Condition	RESERVED	Instruction 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.

Loop_Start				
47	46 … 42	41 … 17	16 … 12	11 … 0
Addressing	00101	RESERVED	loop ID	Jump address

Loop Start. Compares the loop ~~count~~iterator with the end value. If loop condition not met jump to the address. Forward jump only. Also computes the index ~~value.~~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 … 42	41 … 17	16 … 12	11 … 0
Addressing	00111	RESERVED	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, ~~and~~ jump~~s~~ BACK ~~only~~ 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.

Call			
47	46 … 42	41…12	11 … 0
Addressing	01000	RESERVED	Jump address

Jumps to the specified address and pushes the ~~IP~~ control flow program counter on the stack.

Return		
47	46 … 42	41 … 0
Addressing	01001	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 … 42	41	40 … 33	32	31	30 … 12	11 … 0
Addressing	01010	RESERVED	Boolean address	Condition	FW only	RESERVED	Jump address

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.

Conditional_End_of_Clause					
47	46 … 42	41	40 … 33	32	31 … 0
Addressing	01011	RESERVED	Boolean address	Condition	RESERVED

This is an optimization in the case of very short shaders (where the control flow instruction can't be hidden anymore and thus are not free. In this case, if the condition is met, the clause is ended, else we continue the execution of the clause.

End_of_Clause		
47	46 … 42	41 … 0
Addressing	01011	RESERVED

Marks the end of a clause.

To prevent infinite loops, we will keep 9 bits loop counters 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 ~~registers~~GPRs.

6.3 Data dependant predicate instructions

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

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

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

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

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

P0_ADD_# R0,R1,R2

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

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

6.4 HW Detection of PV,PS

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

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

6.5 Register file indexing

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

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

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

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

Index = Loop_~~counter~~iterator*Loop_~~iterator~~step + Loop_~~init~~start.

The index is going to return 0 if it is out of the range. We loop until loop_iterator = loop_count. Loop_step is a signed value [-128…127].

6.6 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.**

6.7 Debugging the Shaders

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

6.7.1 *Method 1: Debugging ~~registers~~registers*

Current plans are to expose 2 debugging, or error notification, ~~registers~~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
- jump error
  relative jump address > size of the control flow program
  relative jump address > length of the shader program
- constant overflow
- register overflow
- call stack
  call with stack full
  return with stack empty

With two of the errors, a jump error or a register overflow will 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 the other errors, program can continue to run, potentially to worst-case limits.

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

{ISSUE : Interrupt to the driver or not?}

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

The sequencer will have a 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 :

1) Normal
2) Debug Kill
3) Debug Addr + Count

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 clause 7 is always exporting 12 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:

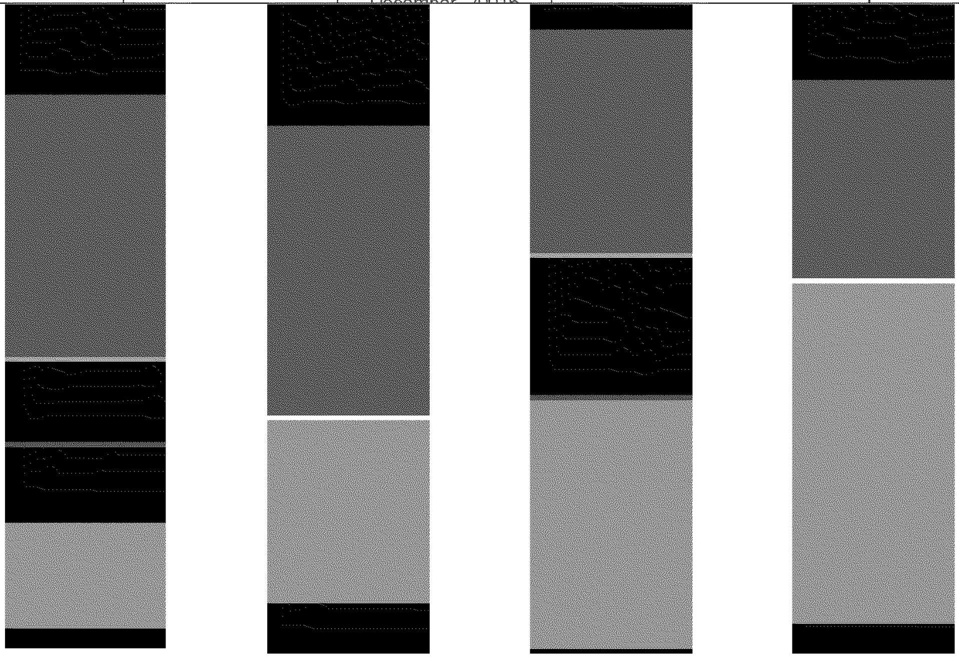
MASK_SETE
MASK_SETNE
MASK_SETGT
MASK_SETGTE

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 VERTEX_REG_SIZE for vertices and 256-VERTEX_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.

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 ~~arbitrers~~arbiters, one for the even clocks and one for the odd clocks. For exa~~e~~mple, 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 ~~to select~~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 ~~to enter~~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~~ and 1 bit to specify if the vector is of pixels or vertices~~. ~~Since pixels and vertices are kept in order in the shader pipe, we only need two fifos (one for vertices and one for pixels) deep enough to cover the shader pipe latency. This size will be determined later when we will know the size of the small fifos between the reservation stations.~~

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~~registers with write BW 512 bits/clock and read BW 256 bits/clock. The staging ~~registers~~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

~~The Deltas have a leading 1, the Full precision IJs don't. This means that in the case of the deltas we MUST be able to shift 8 right (exponent value of 0 means number = 0, exponent value of 1 means shift right 8).~~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. ~~This means that t~~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. 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.

17. 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.

18. Exporting Arbitration

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

1) Position exports and position exports cannot be co-issued.
2) Position exports and memory exports cannot be co-issued.
3) Position exports and Z/Color exports cannot be co-issued.
4) Memory exports and Z/Color exports cannot be co-issued.
5) Memory exports and memory exports cannot be co-issued.
6) Z/color exports and Z/color exports cannot be co-issued.
7) Parameter exports and Z/Color exports CAN be co-issued.
8) Parameter exports and parameter exports CAN be co-issued.
9) Parameter exports and memory exports CAN be co-issued.

19. 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:

19.1 Vertex Shading

0:15	- 16 parameter cache
16:31	- Empty (Reserved?)
32:43	- 12 vertex exports to the frame buffer and index
44:47	- Empty
48:59	- 12 debug export (interpret as normal vertex export)
60	- export addressing mode
61	- Empty
62	- sprite size export that goes with position export (point_h,point_w,edgeflag,misc)
63	- position

19.2 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?)
32:43	- 12 exports for multipass pixel shaders.
44:47	- Empty
48:59	- 12 debug exports (interpret as normal pixel export)
60	- export addressing mode
61:62	- Empty
63	- Z for primary buffer (Z exported to 'alpha' component)

20. Special Interpolation modes

20.1 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 realtime 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.

20.2 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_I0 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 ~~registers~~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_I0 disable, snd_xy disable, no gen_st – I0 = No modification
Param_Gen_I0 disable, snd_xy disable, gen_st – I0 = No modification
Param_Gen_I0 disable, snd_xy enable, no gen_st – I0 = No modification
Param_Gen_I0 disable, snd_xy enable, gen_st – I0 = No modification
Param_Gen_I0 enable, snd_xy disable, no gen_st – I0 = garbage, garbage, garbage, faceness
Param_Gen_I0 enable, snd_xy disable, gen_st – I0 = garbage, garbage, s, t
Param_Gen_I0 enable, snd_xy enable, no gen_st – I0 = screen x, screen y, garbage, faceness
Param_Gen_I0 enable, snd_xy enable, gen_st – I0 = screen x, screen y, s, t

20.3 Auto generated counters

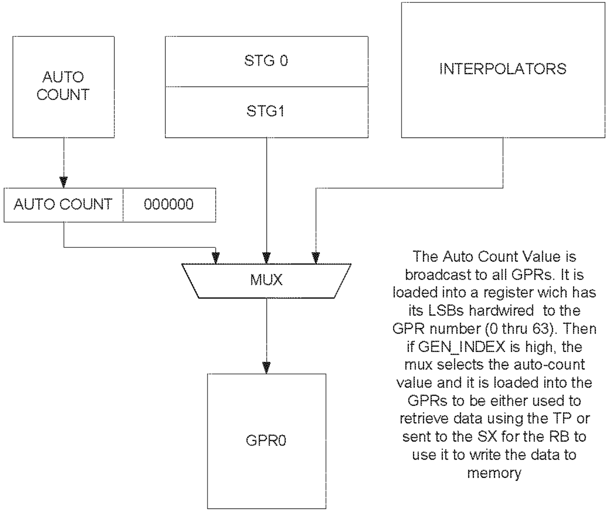
In the cases we are dealing with multipass shaders, the sequencer is going to generate a vector count to be able to both use this count to write the 1st pass data to memory and then use the count to retrieve the data on the 2nd pass. The count is always generated in the same way but it is passed to the 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. While there is only one count broadcast to the ~~registers~~GPRs, the LSB are hardwired to specific values making the index different for all elements in the vector.

20.3.1 *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).

20.3.2 *Pixel shaders*

In the case of pixel shaders, if GEN_INDEX is set, the data will be put in the x field of the 2nd register (I1.x).



The Auto Count Value is broadcast to all GPRs. It is loaded into a register wich has its LSBs hardwired to the GPR number (0 thru 63). Then if GEN_INDEX is high, the mux selects the auto-count value and it is loaded into the GPRs to be either used to retrieve data using the TP or sent to the SX for the RB to use it to write the data to memory

21. 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.

21.1 Parameter cache synchronization

Formatted: Bullets and Numbering

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.

22. 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 20.2 for details on how to control the interpolation in this mode.

22.1 Vertex indexes imports

In order to import vertex indexes, we have 64x2x96 staging ~~registers~~registers. These are loaded one at a time by the VGT block. They are loaded in floating point format and can be transferred in 4 or 8 clocks to the GPRs.

23. ~~Registers~~Registers

23.1 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_STORE_ALLOC	interleaved, separate
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_~~INDEX~~ ~~_PORT~~ADDR	This is where the CP puts the base address of the instruction writes and type (auto-incremented on reads/writes)
INSTRUCTION_DATA	This is where the CP puts the actual data going to the instruction memory
CONSTANT_DATA	This is where the CP puts constant data (32 bits)
CONSTANT_ADDR	This is where the CP puts the logical constant address (9 bits)
INSTRUCTION_~~INDEX~~ ~~PORT~~ADDR_RT	This is where the CP puts the base address of the instruction writes and type for Real Time (auto-incremented on reads/writes)
INSTRUCTION_DATA_RT	This is where the CP puts the actual data going to the instruction memory for Real Time
CONSTANT_DATA_RT	This is where the CP puts constant data for Real Time (32 bits)
CONSTANT_ADDR_RT	This is where the CP puts the logical constant address for Real Time (9 bits)
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 ~~remaping~~re-mapping table operates on the rest of the memory
EXPORT_LATE	Controls whether or not we are exporting position from clause 3. If set, position exports occur at clause 7.

23.2 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 ~~registers~~GPRs to allocate for pixel shader programs
VS_NUM_REG	number of ~~registers~~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)
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_MASK	which of the last 6 ALU clauses is exporting (multipass only)
VS_EXPORT_MODE	0: position (1 vector), 1: position (2 vectors), 3:multipass
VS_EXPORT _COUNT_{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
GEN_INDEX	Auto generates an address from 0 to XX. Puts the results into R1 for pixel shaders and R3 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)

24. DEBUG ~~registers~~Registers

24.1 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_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)

25. Interfaces

25.1 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.

~~25.1.1 SC to SP : IJ bus~~

Formatted: Bullets and Numbering

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

~~25.1.2~~25.1.1 *SC to SQ : IJ Control bus*

Formatted: Bullets and Numbering

This is the control information sent to the sequencer in order to control the IJ fifos and all other information needed to execute a shader program on the sent pixels. This information is sent over 2 clocks, if SENDXY is asserted the next control packet is going to be ignored and XY information is going to be sent on the IJ bus (for the quads that where just sent). All pixels from the group of quads are from the same primitive, all quads of a vector are from the same render state.

Name	Direction	Bits	Description
SC_SQ_q_wr_mask	SC→SQ	4	Quad Write mask left to right
SC_SQ_lod_correct	SC→SQ	24	LOD correction per quad (6 bits per quad)
SC_SQ_flat_vertex	SC→SQ	2	Provoking vertex for flat shading
SC_SQ_param_ptr0	SC→SQ	11	P Store pointer for vertex 0
SC_SQ_param_ptr1	SC→SQ	11	P Store pointer for vertex 1
SC_SQ_param_ptr2	SC→SQ	11	P Store pointer for vertex 2
SC_SQ_end_of_vect	SC→SQ	1	End of the vector
SC_SQ_store_dealloc	SC→SQ	1	Deallocation token for the P Store
SC_SQ_state	SC→SQ	3	State/constant pointer ~~(6*3+3)~~
SC_SQ_valid_pixel	SC→SQ	16	Valid bits for all pixels
SC_SQ_null_prim	SC→SQ	1	Null Primitive (for PC deallocation purposes)
SC_SQ_end_of_prim	SC→SQ	1	End Of the primitive
SC_SQ_fbface	SC→SQ	1	Front face = 1, back face = 0
SC_SQ_send_xy	SC→SQ	1	Sending XY information [XY information is going to be sent on the next clock]
SC_SQ_prim_type	SC→SQ	3	Real time command need to load tex cords from alternate buffer. Line AA, Point AA and Sprite reads their parameters from GEN_T and GEN_S ~~registers~~GPRs. 000 : Normal 011 : Real Time 100 : Line AA 101 : Point AA 110 : Sprite
SC_SQ_new_vector	SC→SQ	1	This primitive comes from a new vector of vertices. Make sure that the corresponding vertex shader has finished before starting the group of pixels.
SC_SQ_RTRn	SQ→SC	1	Stalls the PA in n clocks
SC_SQ_RTS	SC→SQ	1	SC ready to send data

~~25.1.3~~25.1.2 *SQ to SP: Interpolator bus*

Formatted: Bullets and Numbering

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

~~25.1.4~~25.1.3 *SQ to SP: GPR Input Mux select*

Formatted: Bullets and Numbering

This interface is synchronized with the Interpolator bus. This controls the input mux to the GPRs. The three types of data are: generated index, Interpolated data, vertex index data (coming from the staging ~~registers~~registers).

Name	Direction	Bits	Description
SQ_SPx_data_type	SQ→SPx	2	00: Interpolated data 01: Staging register data 1x: Count
SQ_SPx_index_count	SQ→SPx	12?	Index count, common for all shader pipes
SQ_SPx_stage_addr	SQ→SPx	1	Staging register address 0: First staging register 1: second staging register

~~25.1.5 SQ to SPx: Parameter cache write control~~

Formatted: Bullets and Numbering

~~25.1.6~~25.1.4 SQ to SP: Parameter Cache Read control bus

The four following interfaces (SQ→SP, SQ→SX,SP→SX and SX→Interpolators) are all SYNCHRONIZED together.

Name	Direction	Bits	Description
SQ_SPx_ptr0	SQ→SPx	9	Pointer of PC
SQ_SPx_ptr1	SQ→SPx	9	Pointer of PC
SQ_SPx_ptr2	SQ→SPx	9	Pointer of PC
SQ_SP0_read_ena	SQ→SP0	4	Read enables for the 4 memories in the SP0
SQ_SP1_read_ena	SQ→SP1	4	Read enables for the 4 memories in the SP1
SQ_SP2_read_ena	SQ→SP2	4	Read enables for the 4 memories in the SP2
SQ_SP3_read_ena	SQ→SP3	4	Read enables for the 4 memories in the SP3

~~25.1.7~~25.1.5 SQ to SX: Parameter Cache Mux control Bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SXx_mux0	SQ→SXx	4	Mux control for PC (4 MSbs of Pointer)
SQ_SXx_mux1	SQ→SXx	4	Mux control for PC (4 MSbs of Pointer)
SQ_SXx_mux2	SQ→SXx	4	Mux control for PC (4 MSbs of Pointer)

~~25.1.8 SP to SX: Parameter data~~

Formatted: Bullets and Numbering

~~25.1.9 SX to Interpolators: Parameter Cache Return bus~~

Formatted: Bullets and Numbering

~~25.1.10~~25.1.6 SQ to SP~~0~~: Staging Register Data

Name	Direction	Bits	Description
SQ_SP0_vgt_vsisr_data	SQ→SP0	96	Pointers of indexes or HOS surface information
SQ_SP0_vgt_vsisr_double	SQ→SP0	1	0: Normal 96 bits per vert 1: double 192 bits per vert
SQ_SP0_data_valid	SQ→SP0	1	Data is valid
SQ_SP1_vgt_vsisr_data	SQ→SP1	96	Pointers of indexes or HOS surface information
SQ_SP1_vgt_vsisr_double	SQ→SP1	1	0: Normal 96 bits per vert 1: double 192 bits per vert
SQ_SP1_data_valid	SQ→SP1	1	Data is valid
SQ_SP2_vgt_vsisr_data	SQ→SP2	96	Pointers of indexes or HOS surface information
SQ_SP2_vgt_vsisr_double	SQ→SP2	1	0: Normal 96 bits per vert 1: double 192 bits per vert
SQ_SP2_data_valid	SQ→SP2	1	Data is valid
SQ_SP3_vgt_vsisr_data	SQ→SP3	96	Pointers of indexes or HOS surface information
SQ_SP3_vgt_vsisr_double	SQ→SP3	1	0: Normal 96 bits per vert 1: double 192 bits per vert
SQ_SP3_data_valid	SQ→SP3	1	Data is valid

~~25.1.11~~25.1.7 PA to SQ : Vertex interface

Formatted: Bullets and Numbering

~~25.1.11.1~~25.1.7.1 Interface Signal Table

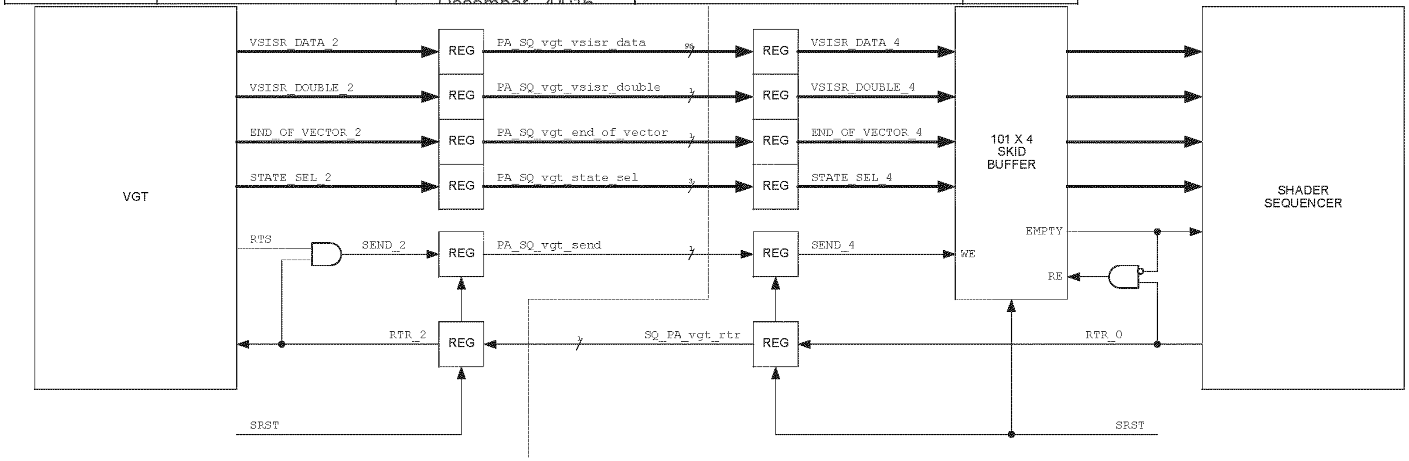
The area difference between the two methods is not sufficient to warrant complicating the interface or the state requirements of the VSISRs. **Therefore, the POR for this interface is that the VGT will transmit the data to the**

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
PA_SQ_vgt_vsisr_data	96	Pointers of indexes or HOS surface information
PA_SQ_vgt_vsisr_double	1	0: Normal 96 bits per vert 1: double 192 bits per vert
PA_SQ_vgt_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 second vector)
PA_SQ_vgt_state	3	Render State (6*3+3 for constants). This signal is guaranteed to be correct when "PA_SQ_vgt_end_of_vector" is high.
PA_SQ_vgt_send	1	Data on the VGT_SQ is valid receive (see write-up for standard R400 SEND/RTR interface handshaking)
SQ_PA_vgt_rtr	1	Ready to receive (see write-up for standard R400 SEND/RTR interface handshaking)

~~25.1.11.2~~25.1.7.2 Interface Diagrams

Formatted: Bullets and Numbering


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

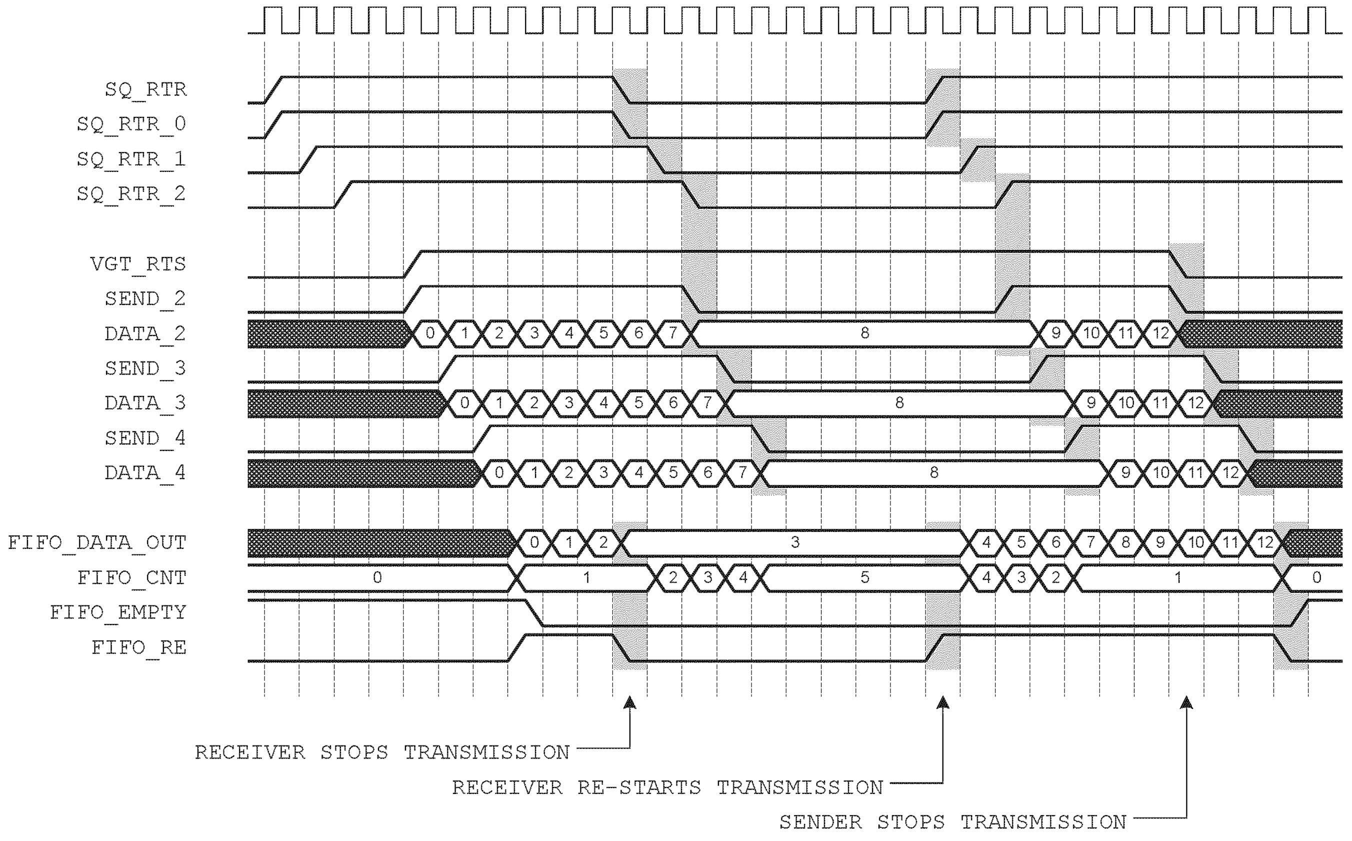


Figure 1. Detailed Logical Diagram for PA_SQ_vgt Interface.

~~25.1.12~~25.1.8 *SQ to CP: State report*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_CP_vrtx_ state	SEQ→CP	3	Oldest vertex state still in the pipe
SQ_CP_pix_state	SEQ→CP	3	Oldest pixel state still in the pipe

~~25.1.13 *SP to SX : Pixel/Vertex write to SX*~~

Formatted: Bullets and Numbering

~~25.1.14~~25.1.9 *SQ to SX: Control bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SXx_exp_Pixel	SQ→SXx	1	1: Pixel 0: Vertex
SQ_SXx_exp_start	SQ→SXx	1	Raised to indicate that the SQ is starting an export
SQ_SXx_exp_Clause	SQ→SXx	3	Clause number, which is needed for vertex clauses
SQ_SXx_exp_State	SQ→SXx	3	State ID, which is needed for vertex clauses

These fields are sent synchronously with SP export data, described in SP0→SX0 interface
{ISSUE: Where are the PC pointers}

~~25.1.15~~25.1.10 *SX to SQ : Output file control*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SXx_SQ_Export_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_Export_Position	SXx→SQ	1	Specifies whether there is room for another position.
SXx_SQ_Export_Buffer	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

~~25.1.16 *Shader Engine to Fetch Unit Bus*~~

Formatted: Bullets and Numbering

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

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

~~25.1.17~~25.1.11 ~~*Sequencer to Fetch Unit bus*~~*SQ to TP: Control bus*

Formatted: Bullets and Numbering

Once every clock, the fetch unit sends to the sequencer on which clause it is now working and if the data in the ~~registers~~GPRs is ready or not. This way the sequencer can update the fetch counters 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	Direction	Bits	Description
TPx_SQ_data_rdy	TPx→ SQ	1	Data ready

TPx_SQ_clause_num	TPx→ SQ	3	Clause number
SQ_TPx_const	SQ→TPx	64	Fetch state sent over 4 clocks
SQ_TPx_instuct	SQ→TPx	24	Fetch instruction sent over 4 clocks
SQ_TPx_end_of_clause	SQ→TPx	1	Last instruction of the clause
SQ_TPx_phase	SQ→TPx	2	Write phase signal
SQ_TP0_lod_correct	SQ→TP0	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP0_pmask	SQ→TP0	4	Pixel mask 1 bit per pixel
SQ_TP1_lod_correct	SQ→TP1	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP1_pmask	SQ→TP1	4	Pixel mask 1 bit per pixel
SQ_TP2_lod_correct	SQ→TP2	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP2_pmask	SQ→TP2	4	Pixel mask 1 bit per pixel
SQ_TP3_lod_correct	SQ→TP3	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP3_pmask	SQ→TP3	4	Pixel mask 1 bit per pixel
SQ_TPx_clause_num	SQ→TPx	3	Clause number
SQ_TPx_write_gpr_index	SQ->TPx	7	Index into Register file for write of returned Fetch Data

Formatted: Bullets and Numbering

25.1.12 *TP to SQ: Texture stall*

The TP sends this signal to the SQ when its input buffer is full. The SQ is going to send it to the SP X clocks after reception (maximum of 3 clocks of pipeline delay).

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

Formatted: Bullets and Numbering

25.1.13 *SQ to SP: Texture stall*

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

Formatted: Bullets and Numbering

~~25.1.18~~25.1.14 *~~Sequencer~~Q to SP: GPR and Parameter cache control*

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_re_addr	SQ→SPx	1	Read Enable
SQ_SPx_gpr_we_addr	SQ→SPx	1	Write Enable for the GPRs
SQ_SPx_gpr_phase_mux	SQ→SPx	2	The phase mux
SQ_SPx_~~gpr~~_channel_mask	SQ→SPx	4	The channel mask
SQ_SP0_~~gpr~~_pixel_mask	SQ→SP0	4	The pixel mask
SQ_SP1_~~gpr~~_pixel_mask	SQ→SP1	4	The pixel mask
SQ_SP2_~~gpr~~_pixel_mask	SQ→SP2	4	The pixel mask
SQ_SP3_~~gpr~~_pixel_mask	SQ→SP3	4	The pixel mask
SQ_SPx_pc_we_addr	SQ→SPx	1	Write Enable for the parameter caches

~~25.1.19~~25.1.15 ~~Sequencer~~ SQ to SPx: Instructions

Formatted: Bullets and Numbering

Name ~~Name~~	Direction~~Direction~~	Bits~~Bits~~	Description~~Description~~
SQ_SPx_instruct_start~~SQ_SPx_instruct_start~~	SQ→SPx~~SQ→SPx~~	1~~1~~	Instruction start~~Instruction start~~
SQ_SP_instruct~~SQ_SP_instruct~~	SQ→SPx~~SQ→SPx~~	20~~20~~	Instruction sent over 4 clocks~~Instruction sent over 4 clocks~~
SQ_SPx_stall~~SQ_SPx_stall~~	SQ→SPx~~SQ→SPx~~	1~~1~~	Stall signal~~Stall signal~~
SQ_SPx_export_count~~SQ_SPx_Shader_Count~~	SQ→SPx~~SQ→SPx~~	33	Each set of four pixels or vectors is exported over eight clocks. This field specifies where the SP is in that sequence.~~Each set of four pixels or vectors is exported over eight clocks. This field specifies where the SP is in that sequence.~~
SQ_SPx_export_last~~SQ_SPx_Shader_Last~~	SQ→SPx~~SQ→SPx~~	1~~1~~	Asserted on the first shader count of the last export of the clause~~Asserted on the first shader count of the last export of the clause~~
SQ_SP0_export_pvalid~~SQ_SP0_Shader_PixelValid~~	SQ→SP0~~SQ→SP0~~	44	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~~Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock~~
SQ_SP0_export_wvalid~~SQ_SP0_Shader_WordValid~~	SQ→SP0~~SQ→SP0~~	22	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors~~Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors~~
SQ_SP1_ export_pvalid~~SQ_SP1_Shader_PixelValid~~	SQ→SP1~~SQ→SP1~~	44	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~~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_	SQ→SP1~~SQ→SP1~~	22	Specifies whether to write

export_wvalid~~SQ_SP1_Shader_WordValid~~			low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors~~Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors~~
SQ_SP2_export_pvalid~~SQ_SP2_Shader_PixelValid~~	SQ→SP2~~SQ→SP2~~	4~~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~~Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock~~
SQ_SP2_ export_wvalid~~SQ_SP2_Shader_WordValid~~	SQ→SP2~~SQ→SP2~~	2~~2~~	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors~~Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors~~
SQ_SP3_export_pvalid~~SQ_SP3_Shader_PixelValid~~	SQ→SP3~~SQ→SP3~~	4~~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~~Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock~~
SQ_SP3_ export_wvalid~~SQ_SP3_Shader_WordValid~~	SQ→SP3~~SQ→SP3~~	2~~2~~	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors~~Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors~~

~~25.1.20~~25.1.16 *SP to ~~Sequencer~~SQ: Constant address load*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SP0_SQ_const_addr	SP0→SQ	36	Constant address load to the sequencer
SP0_SQ_valid	SP0→SQ	1	Data valid
SP1_SQ_const_addr	SP1→SQ	36	Constant address load to the sequencer

SP1_SQ_valid	SP1→SQ	1	Data valid
SP2_SQ_const_addr	SP2→SQ	36	Constant address load to the sequencer
SP2_SQ_valid	SP2→SQ	1	Data valid
SP3_SQ_const_addr	SP3→SQ	36	Constant address load to the sequencer
SP3_SQ_valid	SP3→SQ	1	Data valid

~~25.1.21~~25.1.17 *~~Sequencer~~ SQ to SPx: constant broadcast*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_constant	SQ→SPx	128	Constant broadcast

~~25.1.22~~25.1.18 *SP0 to ~~Sequencer~~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

~~25.1.23~~25.1.19 *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)

~~25.1.24~~25.1.20 *CP to SQ: RBBM bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
rbbm_we	CP→SQ	1	Write Enable
rbbm_a	CP→SQ	18	Address -- Upper Extent is TBD
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

26. Examples of program executions

26.1.1 *Sequencer Control of a Vector of Vertices*

1. PA sends a vector of 64 vertices (actually vertex indices – 32 bits/index for 2048 bit total) to the RE's Vertex FIFO
   - state pointer as well as tag into position cache is sent along with vertices
   - space was allocated in the position cache for transformed position before the vector was sent
   - **also before the vector is sent to the RE, the CP has loaded the global instruction store with the vertex shader program (using the MH?)**
   - The vertex program is assumed to be loaded when we receive the vertex vector.
     - the SEQ then accesses the IS base for this shader using the local state pointer (provided to all sequencers by the RBBM when the CP is done loading the program)
2. SEQ arbitrates between the Pixel FIFO and the Vertex FIFO – basically the Vertex FIFO always has priority
   - at this point the vector is removed from the Vertex FIFO
   - the arbiter is not going to select a vector to be transformed if the parameter cache is full unless the pipe as nothing else to do (ie no pixels are in the pixel fifo).

3. SEQ allocates space in the SP register file for index data plus GPRs used by the program
    - the number of GPRs required by the program is stored in a local state register, which is accessed using the state pointer that came down with the vertices
    - SEQ will not send vertex data until space in the register file has been allocated
4. SEQ sends the vector to the SP register file over the RE_SP interface (which has a bandwidth of 2048 bits/cycle)
    - the 64 vertex indices are sent to the 64 register files over 4 cycles
        - RF0 of SU0, SU1, SU2, and SU3 is written the first cycle
        - RF1 of SU0, SU1, SU2, and SU3 is written the second cycle
        - RF2 of SU0, SU1, SU2, and SU3 is written the third cycle
        - RF3 of SU0, SU1, SU2, and SU3 is written the fourth cycle
    - the index is written to the least significant 32 bits **(floating point format?) (what about compound indices)** of the 128-bit location within the register file (w); the remaining data bits are set to zero (x, y, z)
5. SEQ constructs a control packet for the vector and sends it to the first reservation station (the FIFO in front of fetch state machine 0, or TSM0 FIFO)
    - the control packet contains the state pointer, the tag to the position cache and a register file base pointer.
6. TSM0 accepts the control packet and fetches the instructions for fetch clause 0 from the global instruction store
    - TSM0 was first selected by the TSM arbiter before it could start
7. all instructions of fetch clause 0 are issued by TSM0
8. the control packet is passed to the next reservation station (the FIFO in front of ALU state machine 0, or ASM0 FIFO)
    - TSM0 does not wait for requests made to the Fetch Unit to complete; it passes the register file write index for the fetch data to the TU, which will write the data to the RF as it is received
    - once the TU has written all the data to the register files, it increments a counter that is associated with ASM0 FIFO; a count greater than zero indicates that the ALU state machine can go ahead start to execute the ALU clause
9. ASM0 accepts the control packet (after being selected by the ASM arbiter) and gets the instructions for ALU clause 0 from the global instruction store
10. all instructions of ALU clause 0 are issued by ASM0, then the control packet is passed to the next reservation station (the FIFO in front of fetch state machine 1, or TSM1 FIFO)
11. the control packet continues to travel down the path of reservation stations until all clauses have been executed
    - position can be exported in ALU clause 3 (or 4?); the data (and the tag) is sent over a position bus (which is shared with all four shader pipes) back to the PA's position cache
    - A parameter cache pointer is also sent along with the position data. This tells to the PA where the data is going to be in the parameter cache.
        - there is a position export FIFO in the SP that buffers position data before it gets sent back to the PA
        - the ASM arbiter will prevent a packet from starting an exporting clause if the position export FIFO is full
    - parameter data is exported in clause 7 (as well as position data if it was not exported earlier)
        - parameter data is sent to the Parameter Cache over a dedicated bus
        - the SEQ allocates storage in the Parameter Cache, and the SEQ deallocates that space when there is no longer a need for the parameters (it is told by the PA when using a token).
        - the ASM arbiter will prevent a packet from starting on ASM7 if the parameter cache (or the position buffer if position is being exported) is full
12. after the shader program has completed, the SEQ will free up the GPRs so that they can be used by another shader program

26.1.2 *Sequencer Control of a Vector of Pixels*

1. **As with vertex shader programs, pixel shaders are loaded into the global instruction store by the CP**
    - At this point it is assumed that the pixel program is loaded into the instruction store and thus ready to be read.
2. the RE's Pixel FIFO is loaded with the barycentric coordinates for pixel quads by the detailed walker
    - the state pointer and the LOD correction bits are also placed in the Pixel FIFO
    - the Pixel FIFO is wide enough to source four quad's worth of barycentrics per cycle

3. SEQ arbitrates between Pixel FIFO and Vertex FIFO – when there are no vertices pending OR there is no space left in the register files for vertices, the Pixel FIFO is selected
4. SEQ allocates space in the SP register file for all the GPRs used by the program
   - the number of GPRs required by the program is stored in a local state register, which is accessed using the state pointer
   - SEQ will not allow interpolated data to be sent to the shader until space in the register file has been allocated
5. SEQ controls the transfer of interpolated data to the SP register file over the RE_SP interface (which has a bandwidth of 2048 bits/cycle). See interpolated data bus diagrams for details.
6. SEQ constructs a control packet for the vector and sends it to the first reservation station (the FIFO in front of fetch state machine 0, or TSM0 FIFO)
   - note that there is a separate set of reservation stations/arbiters/state machines for vertices and for pixels
   - the control packet contains the state pointer, the register file base pointer, and the LOD correction bits
   - all other information (such as quad address for example) travels in a separate FIFO
7. TSM0 accepts the control packet and fetches the instructions for fetch clause 0 from the global instruction store
   - TSM0 was first selected by the TSM arbiter before it could start
8. all instructions of fetch clause 0 are issued by TSM0
9. the control packet is passed to the next reservation station (the FIFO in front of ALU state machine 0, or ASM0 FIFO)
   - TSM0 does not wait for fetch requests made to the Fetch Unit to complete; it passes the register file write index for the fetch data to the TU, which will write the data to the RF as it is received
   - once the TU has written all the data for a particular clause to the register files, it increments a counter that is associated with the ASM0 FIFO; a count greater than zero indicates that the ALU state machine can go ahead and pop the FIFO and start to execute the ALU clause
10. ASM0 accepts the control packet (after being selected by the ASM arbiter) and gets the instructions for ALU clause 0 from the global instruction store
11. all instructions of ALU clause 0 are issued by ASM0, then the control packet is passed to the next reservation station (the FIFO in front of fetch state machine 1, or TSM1 FIFO)
12. the control packet continues to travel down the path of reservation stations until all clauses have been executed
    - pixel data is exported in the last ALU clause (clause 7)
      - it is sent to an output FIFO where it will be picked up by the render backend
      - the ASM arbiter will prevent a packet from starting on ASM7 if the output FIFO is full
13. after the shader program has completed, the SEQ will free up the GPRs so that they can be used by another shader program

26.1.3 *Notes*

14. The state machines and arbiters will operate ahead of time so that they will be able to immediately start the real threads or stall.
15. The register file base pointer for a vector needs to travel with the vector through the reservation stations, but the instruction store base pointer does not – this is because the RF pointer is different for all threads, but the IS pointer is only different for each state and thus can be accessed via the state pointer.
16. ~~Waterfalling still needs to be specked out.~~

Formatted: Bullets and Numbering

27. Open issues

~~There is currently an issue with constants. If the constants are not the same for the whole vector of vertices, we don't have the bandwidth from the fetch store to feed the ALUs. Two solutions exists for this problem:~~

~~1)Let the compiler handle the case and put those instructions in a fetch clause so we can use the bandwidth there to operate. This requires a significant amount of temporary storage in the register store.~~

Formatted: Bullets and Numbering

~~2)Waterfall down the pipe allowing only at a given time the vertices having the same constants to operate in parallel. This might in the worst case slow us down by a factor of 16.~~

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

Saving power?

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 2015~~7~~ ~~January, 200211~~	GEN-CXXXXX-REVA	1 of 47

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SQ

Version 1.6~~5~~

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 CO~~NTA~~read_ptrINS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIME~~NTA~~read_ptrL 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.

1. Overview

The sequencer is based on the R300 design. It chooses two ALU clauses and a fetch clause to execute, and executes all of the instructions in a clause before looking for a new clause of the same type. Two ALU clauses 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. 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.

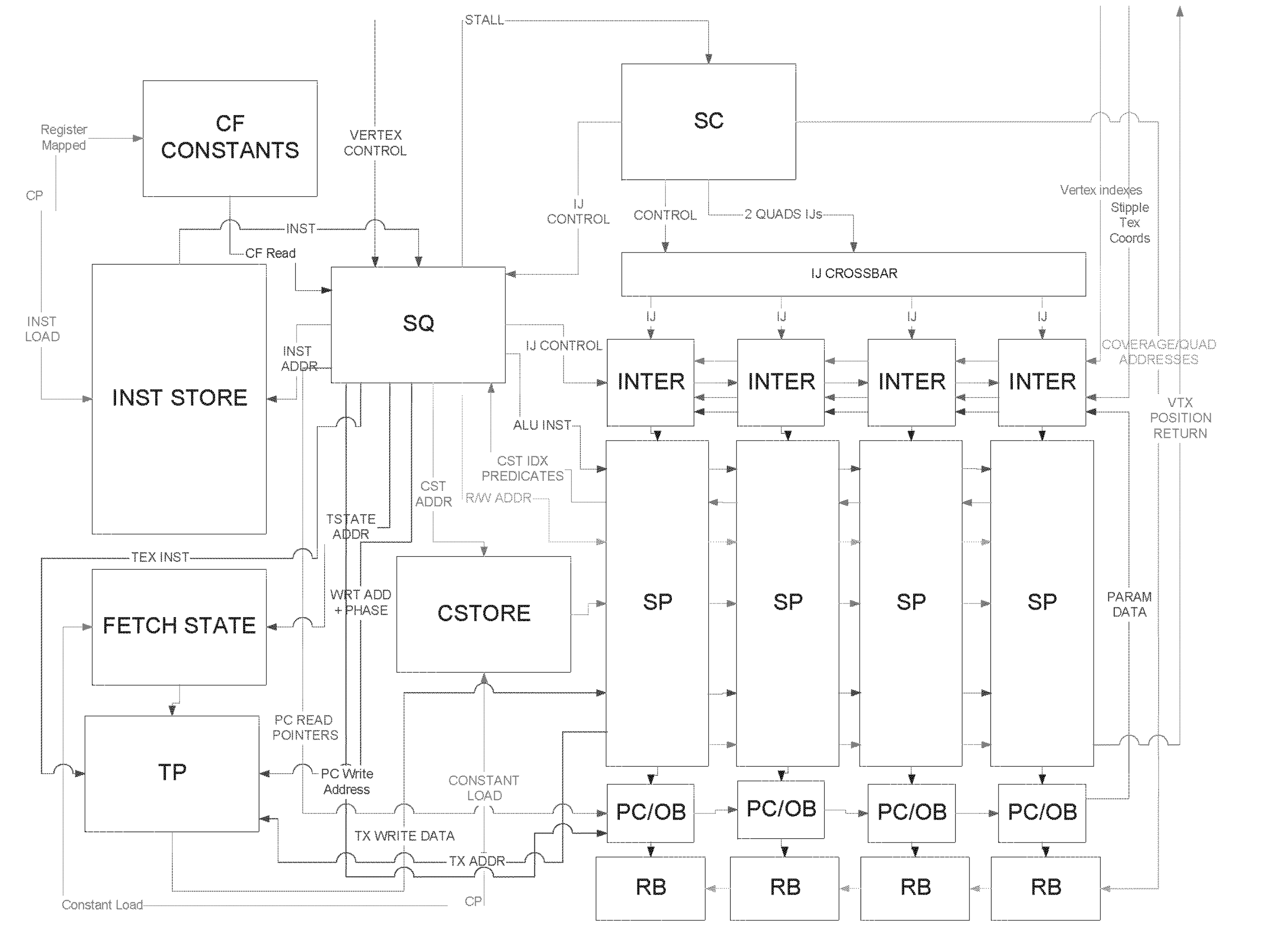
To support the shader pipe the sequencer also contains the shader instruction cache, constant store, control flow constants and texture state. The four shader pipes also execute the same instruction thus there is only one sequencer for the whole chip.

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.

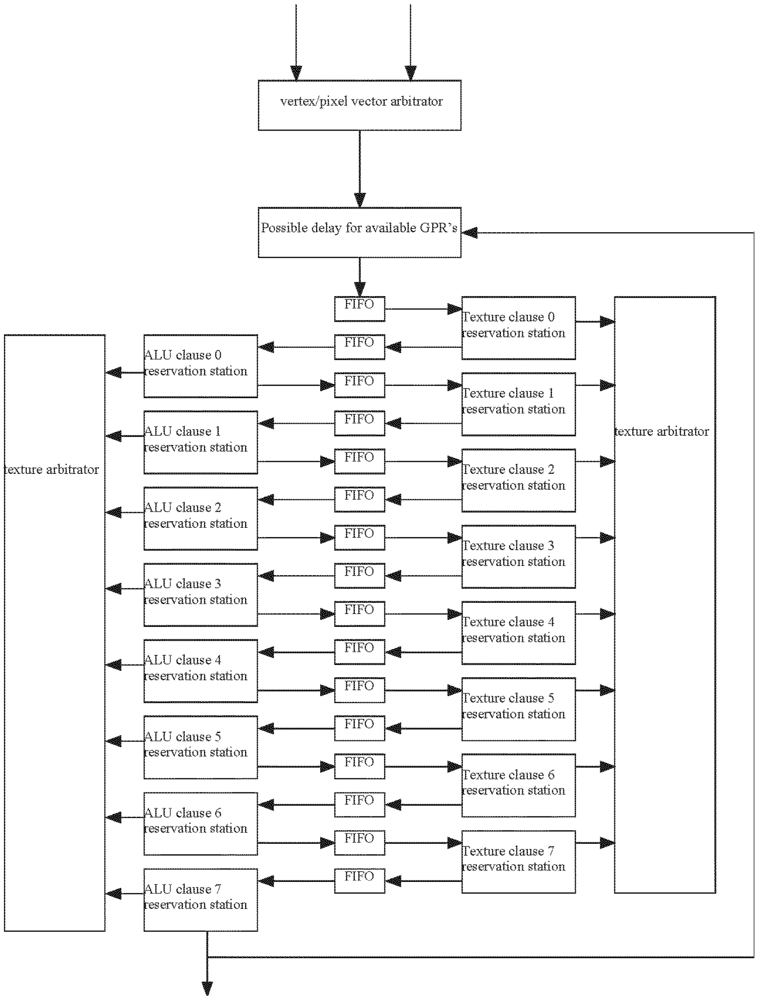
The vertex or pixel program specifies how many GPRs it needs to execute. The sequencer will not start the next vector until the needed space is available in the GPRs.

PROTECTIVE ORDER MATERIAL

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 2015~~7~~ ~~January, 200211~~	GEN-CXXXXX-REVA	7 of 47



1.1 Top Level Block Diagram


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

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.

~~{ISSUE: How do we handle parameter cache pointers (computed, semi-computed or not computed)?}~~

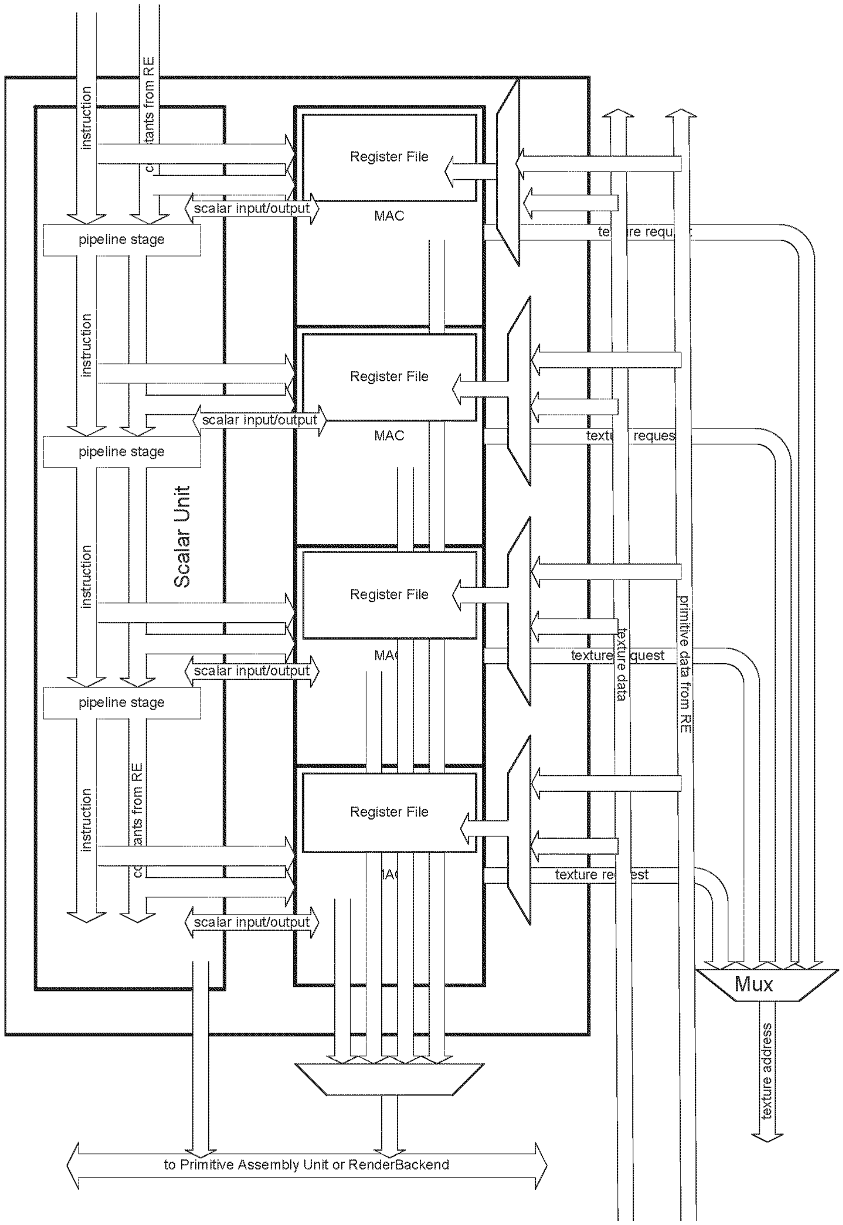
A special case is for multipass vertex shaders, which can export 12 parameters per last 6 clauses to the output buffer. If the output buffer is full or doesn't have enough space the sequencer will prevent such a vertex group to enter an exporting clause.

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).

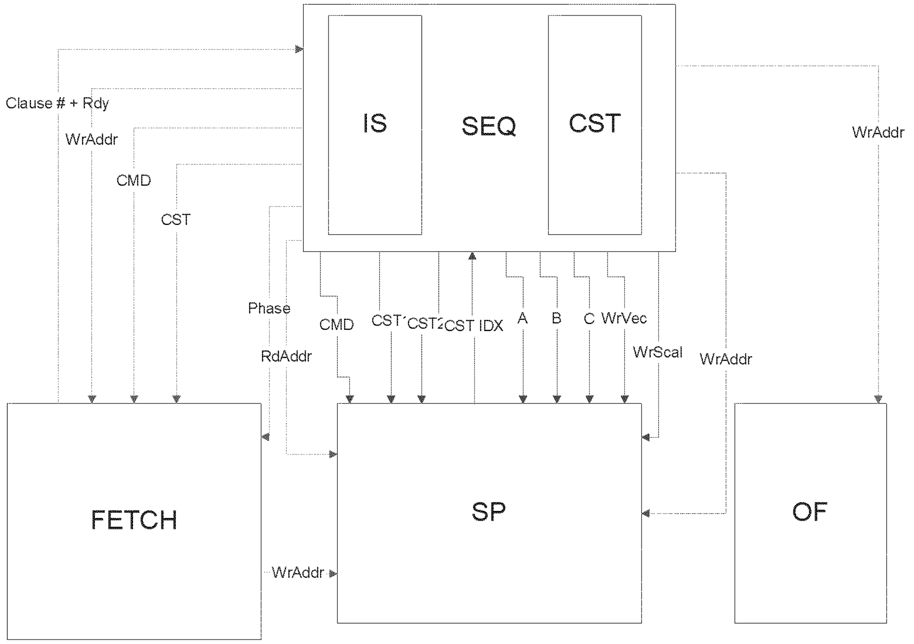
Only one pair of interleaved ALU state machines may have access to the register file address bus or the instruction decode bus at one time. Similarly, only one fetch state machine may have access to the register file address bus at one time. Arbitration is performed by three arbiter blocks (two for the 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.

1.2 Data Flow graph (SP)



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

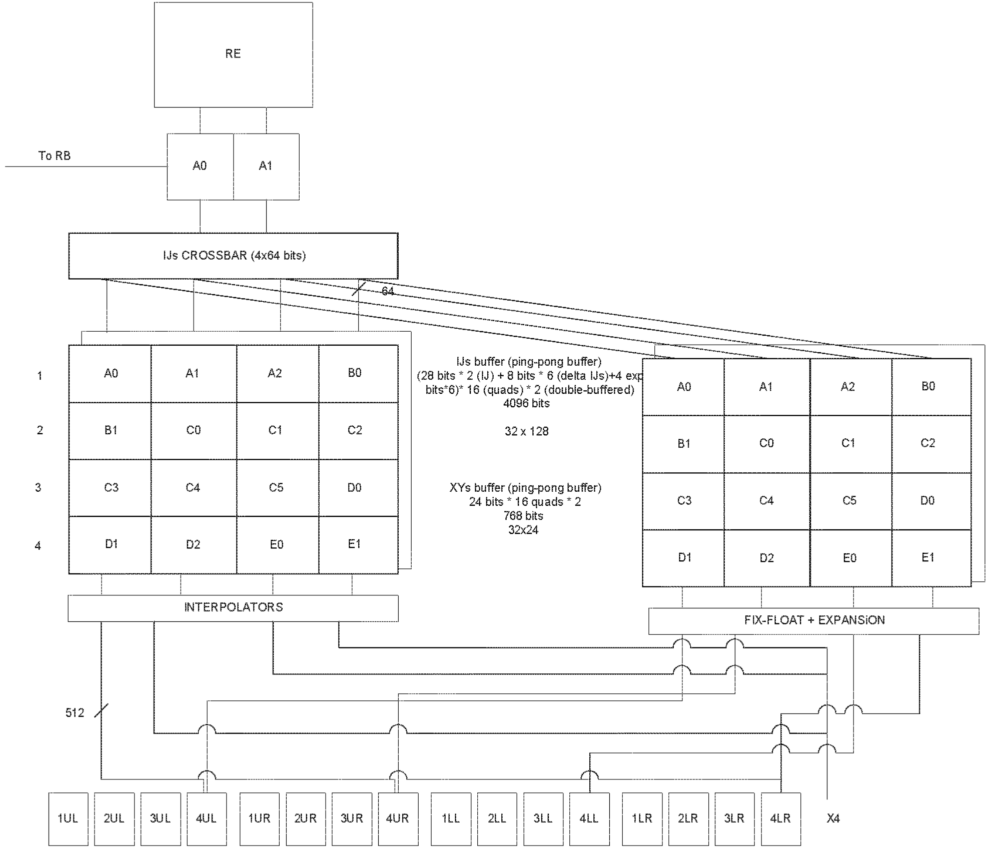
1.3 Control Graph



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.


RE
To RB
A0
A1
IJs CROSSBAR (4x64 bits)
64
1
2
3
4
A0
A1
A2
B0
B1
C0
C1
C2
C3
C4
C5
D0
D1
D2
E0
E1
IJs buffer (ping-pong buffer)
(28 bits * 2 (IJ) + 8 bits * 6 (delta IJs)+4 exp
bits*6)* 16 (quads) * 2 (double-buffered)
4096 bits
32 x 128
XYs buffer (ping-pong buffer)
24 bits * 16 quads * 2
768 bits
32x24
INTERPOLATORS
FIX-FLOAT + EXPANSiON
512
1UL
2UL
3UL
4UL
1UR
2UR
3UR
4UR
1LL
2LL
3LL
4LL
1LR
2LR
3LR
4LR
X4

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001	4 September, 2015~~7~~ ~~January, 200211~~	GEN-CXXXXX-REVA	13 of 47

	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		

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.

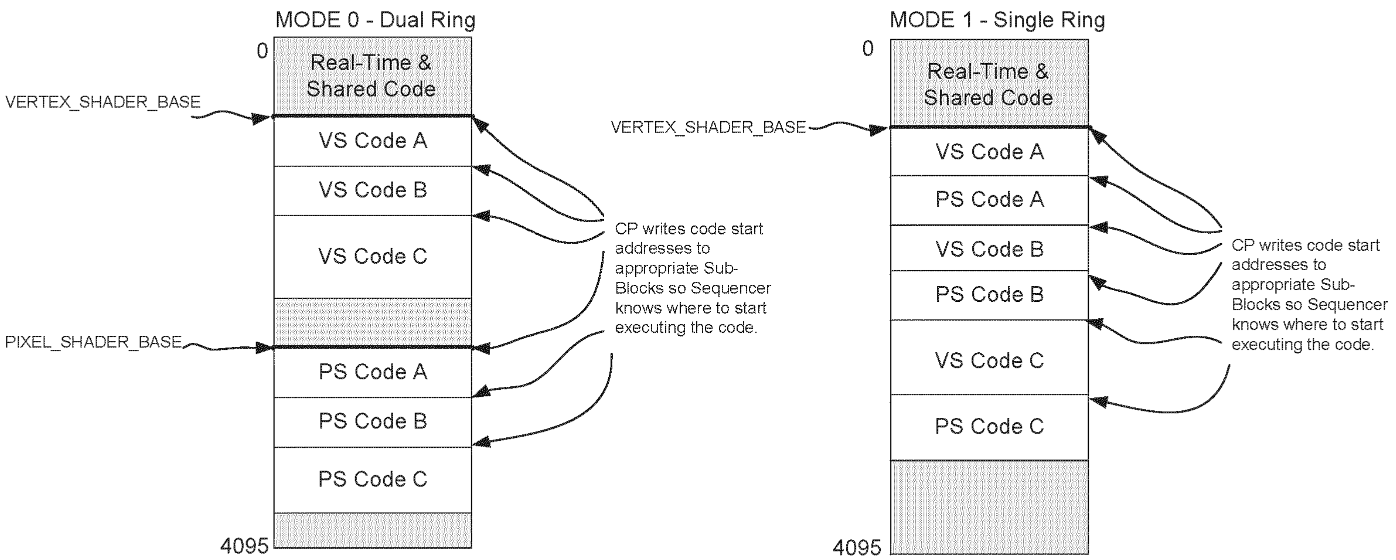
The instruction store is loaded by the CP thru the ~~INST_DATA, INST_INDEX_PORT control register. The INST_INDEX_PORT is auto-incremented on both reads and writes to the INST_DATA register.~~register mapped registers.

The next picture shows the various modes the CP can load the memory. The Sequencer has to keep track of the loading modes in order to wrap around the correct boundaries. ~~The MSB of the INST_INDEX_PORT register contains the packet type for the sequencer to know where it must wrap around.~~ The ~~wrap around points are~~wrap around points are arbitrary and they are specified in the VS_BASE and PIX_BASE registers.

For the Real time commands the story is quite the same but for some small differences. ~~The CP will use the INST_INDEX_PORT_RT and INST_DATA_RT register pair instead of the regular ones and T~~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.

R400 CP's Views of Instruction Memory

Updated: 11/14/2001
John A. Carey



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.

The texture state is also kept in a similar memory. The size of this memory is 192x128. 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 16 lines (each line addresses 2 texture state lines in the real memory). The write granularity is 2 texture state lines (or 384 bits). The driver sends 512 bits but the CP ignores the top 128 bits. It thus takes 12 clocks to write the two texture states.

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 state change. 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.

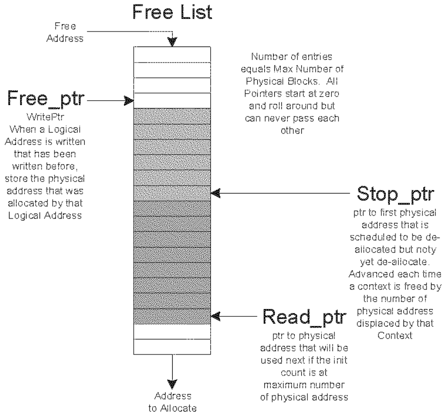
~~The CP is loading the constant store using the CONST_DATA and CONST_ADDR registers. It does so by writing to the CONST_ADDR register the logical address for the constant block it wants to update and then writes 16 times to the CONST_DATA register. The CONST_ADDR is auto-incremented on both reads and writes to the CONST_DATA register.~~ 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 re-mapping tables

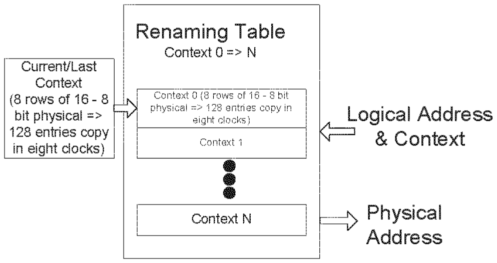
The sequencer is responsible to manage two re-mapping tables (one for the constant store and one for the texture state). On a state change (by the driver), the sequencer will broadside copy the contents of its re-mapping tables to a new one. We have 8 different re-mapping tables we can use concurrently.

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

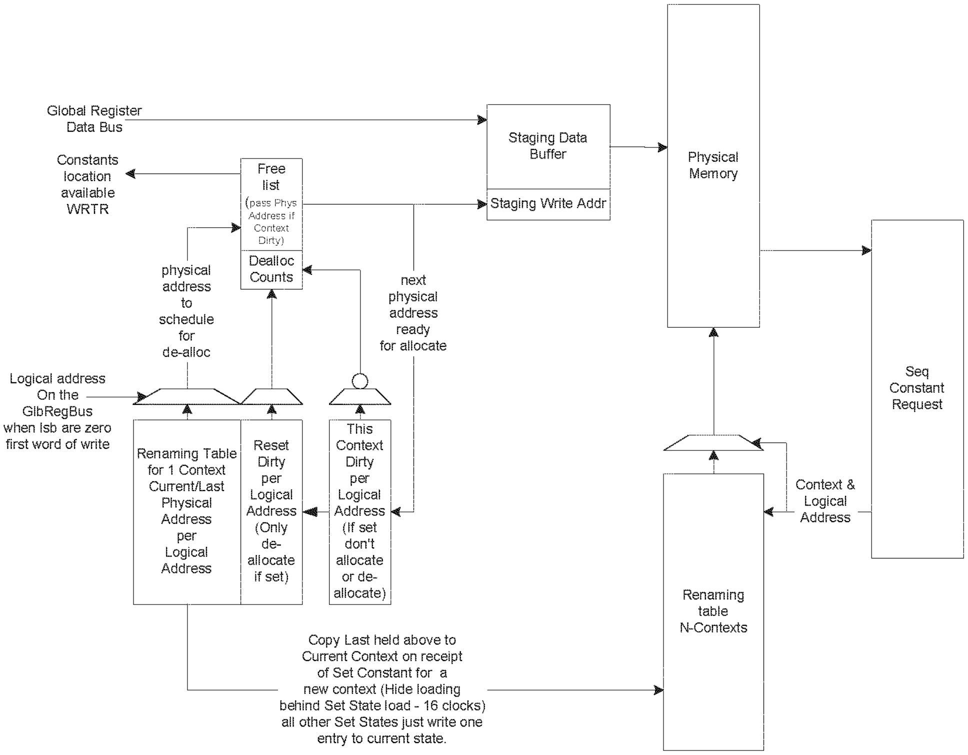
For this model to work, the requirement is that the physical memory MUST be at least twice as large as the logical address space. In our case, since the logical address space is 512, the memory must be of sizes 1024 and above.



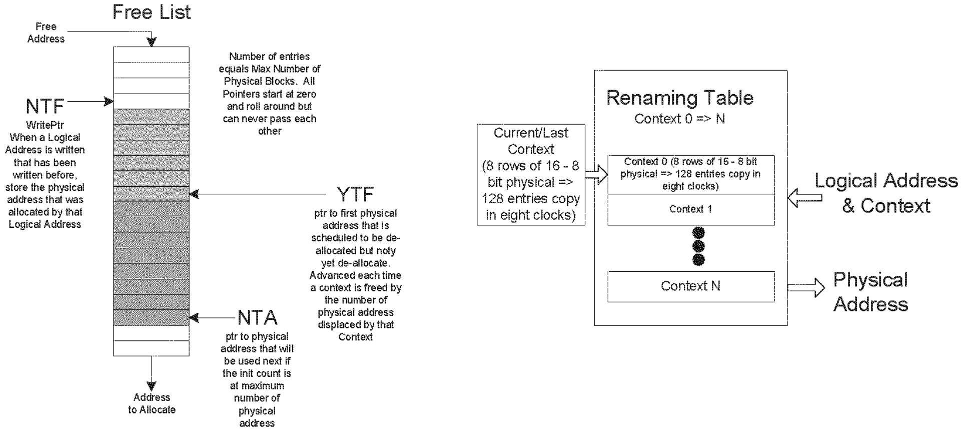
Free List
Free Address
Free_ptr
WritePtr
When a Logical Address is written that has been written before, store the physical address that was allocated by that Logical Address
Number of entries equals Max Number of Physical Blocks. All Pointers start at zero and roll around but can never pass each other
Stop_ptr
ptr to first physical address that is scheduled to be de-allocated but noty yet de-allocate. Advanced each time a context is freed by the number of physical address displaced by that Context
Read_ptr
ptr to physical address that will be used next if the init count is at maximum number of physical address
Address to Allocate

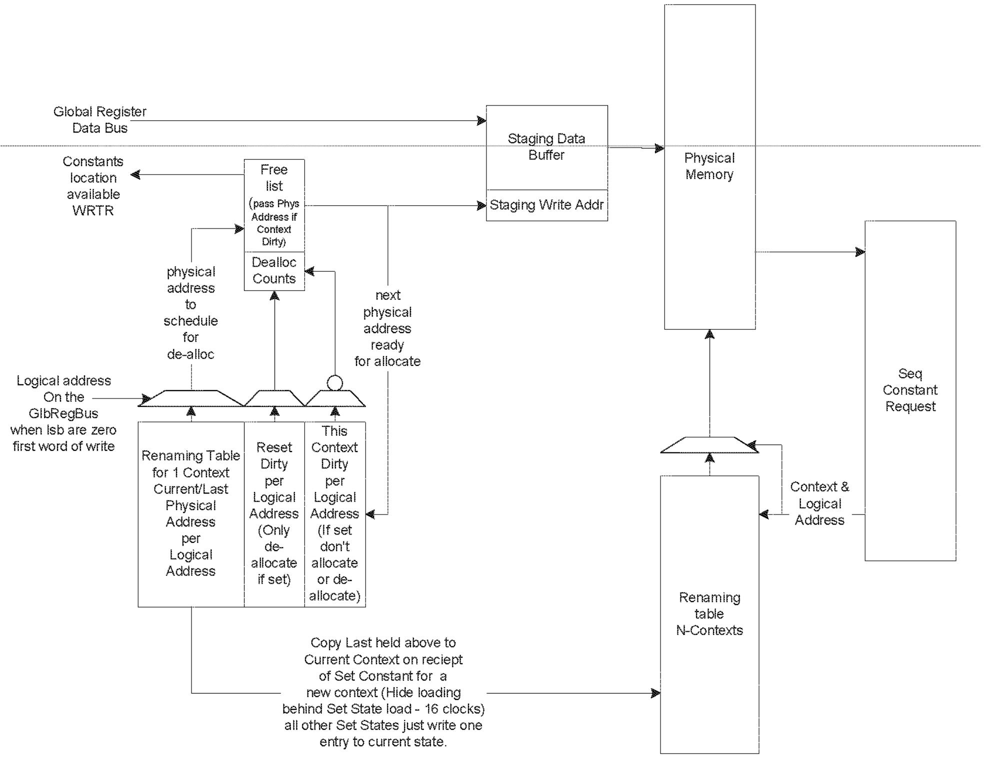


Renaming Table
Context 0 => N
Current/Last Context (8 rows of 16 - 8 bit physical => 128 entries copy in eight clocks)
Context 0 (8 rows of 16 - 8 bit physical => 128 entries copy in eight clocks)
Context 1
Context N
Logical Address & Context
Physical Address



Global Register Data Bus
Staging Data Buffer
Staging Write Addr
Physical Memory
Constants location available WRTR
Free list (pass Phys Address if Context Dirty)
Dealloc Counts
physical address to schedule for de-alloc
next physical address ready for allocate
Logical address On the GlbRegBus when lsb are zero first word of write
Renaming Table for 1 Context Current/Last Physical Address per Logical Address
Reset Dirty per Logical Address (Only de-allocate if set)
This Context Dirty per Logical Address (If set don't allocate or de-allocate)
Seq Constant Request
Context & Logical Address
Renaming table N-Contexts
Copy Last held above to Current Context on receipt of Set Constant for a new context (Hide loading behind Set State load - 16 clocks) all other Set States just write one entry to current state.


Free List
Free Address
Number of entries equals Max Number of Physical Blocks. All Pointers start at zero and roll around but can never pass each other
NTF
WritePtr
When a Logical Address is written that has been written before, store the physical address that was allocated by that Logical Address
YTF
ptr to first physical address that is scheduled to be de-allocated but noty yet de-allocate. Advanced each time a context is freed by the number of physical address displaced by that Context
NTA
ptr to physical address that will be used next if the init count is at maximum number of physical address
Address to Allocate
Renaming Table
Context 0 => N
Current/Last Context (8 rows of 16 - 8 bit physical => 128 entries copy in eight clocks)
Context 0 (8 rows of 16 - 8 bit physical => 128 entries copy in eight clocks)
Context 1
Context N
Logical Address & Context
Physical Address


Global Register Data Bus
Staging Data Buffer
Staging Write Addr
Physical Memory
Constants location available WRTR
Free list
(pass Phys Address if Context Dirty)
Dealloc Counts
physical address to schedule for de-alloc
next physical address ready for allocate
Logical address On the GlbRegBus when lsb are zero first word of write
Renaming Table for 1 Context Current/Last Physical Address per Logical Address
Reset Dirty per Logical Address (Only de-allocate if set)
This Context Dirty per Logical Address (If set don't allocate or de-allocate)
Seq Constant Request
Context & Logical Address
Renaming table N-Contexts
Copy Last held above to Current Context on reciept of Set Constant for a new context (Hide loading behind Set State load - 16 clocks) all other Set States just write one entry to current state.

5.2.1 *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 when ever 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.

5.2.2 *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.

~~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 NTF (Next To Free). 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 YTF (Yet To Free). The YTF pointer will be advanced by the number of address chunks de-allocates when a context finishes. The address between the YTF and NTF cannot be reused because they are still in use. But as soon as the context using then is dismissed the YTF will be advanced.~~

~~The third pointer will be called NTA (Next To Allocate). This pointer will point will point to the next address that can be used for allocation as long as the NTA does not equal the YTF and the IFC is at its maximum count.~~

5.2.3 *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.

~~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 NTF 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 NTF 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.~~

5.2.4 *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.

Formatted: Bullets and Numbering

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.

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.~~The basic operation of the model would start with the NTF, YTF, NTA 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 a set constant comes with a different than last context, 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 NTF pointer location on the free list and the NTF will be incremented. The de-allocation counter for the previous context (zero) 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 NTA pointer if NTA != to YTF .~~
~~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 NTF 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.~~

Formatted: Bullets and Numbering

~~This process will continue as long as set states arrive. This block will provide back pressure to the CP when ever he has not free list entries available (counter at max and YTF == NTA). The command stream will keep a count of contexts of constants in use and prevent more than max constants contexts from being sent.~~
~~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 YTF pointer. This will make all the physical addresses used by this context available to the NTA 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.~~

5.3 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.

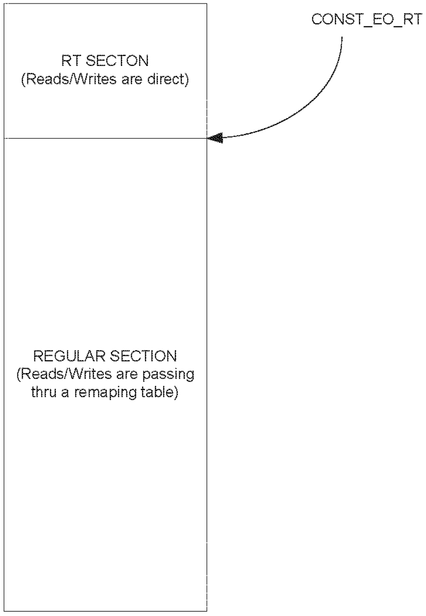
5.4 Real Time Commands

The real time commands constants are written by the CP using the ~~CONST_DATA_RT and CONST_ADDR_RT GPRs~~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.

5.5 Constant Waterfalling

Formatted: Bullets and Numbering

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


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

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

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 eleven basic instructions:

Execute
Conditional_execute
Conditional_Execute_Predicates
Conditional_jump
Conditionnal_Call
Return
Loop_start
Loop_end
End_of_clause
Conditional_End_of_clause
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_or_Jump executes a block of instructions or jumps to an address is the condition is not met.
Conditional_execute_Predicates executes a block of instructions if all bits in the predicate vectors meet the condition.
End_of_clause marks the end of a clause.
Conditional_End_of_clause marks the end of a clause if the condition is met.
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.

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… 42	41 … 24	23 … 12	11 … 0
Addressing	00001	RESERVED	Instruction count	Exec Address

Execute up to 4k instructions at the specified address in the instruction memory.

NOP		
47	46 … 42	41 … 0

Addressing	00010	RESERVED

This is a regular NOP.

Conditional_Execute							
47	46 … 42	41	40 … 33	32	31 … 24	23 … 12	11 … 0
Addressing	00011	RESERVED	Boolean address	Condition	RESERVED	Instruction count	Exec Address

If the specified Boolean (8 bits can address 256 Booleans) meets the specified condition then execute the specified instructions (up to 4k instructions)

Conditional_Execute_Predicates							
47	46 … 42	41 … 35	34 … 33	32	31 … 24	23 … 12	11 … 0
Addressing	00100	RESERVED	Predicate vector	Condition	RESERVED	Instruction 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.

Loop_Start				
47	46 … 42	41 … 17	16 … 12	11 … 0
Addressing	00101	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 … 42	41 … 17	16 … 12	11 … 0
Addressing	00111	RESERVED	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.

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 … 42	41 … 35~~41 … 12~~	34 … 33	32	31 … 12	11 … 0
Addressing	01000	RESERVED	Predicate vector	Condition	RESERVED	Jump address

If the condition is met, j~~J~~umps to the specified address and pushes the control flow program counter on the stack.

Return		
47	46 … 42	41 … 0
Addressing	01001	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 … 42	41	40 … 33	32	31	30 … 12	11 … 0

Addressing	01010	RESERVED	Boolean address	Condition	FW only	RESERVED	Jump address

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.

Conditional_End_of_Clause					
47	46 … 42	41	40 … 33	32	31 … 0
Addressing	01011	RESERVED	Boolean address	Condition	RESERVED

This is an optimization in the case of very short shaders (where the control flow instruction can't be hidden anymore and thus are not free. In this case, if the condition is met, the clause is ended, else we continue the execution of the clause.

End_of_Clause		
47	46 … 42	41 … 0
Addressing	01011	RESERVED

Marks the end of a clause.

To prevent infinite loops, we will keep 9 bits loop counters 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 Data dependant predicate instructions

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

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

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

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

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

P0_ADD_# R0,R1,R2

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

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

6.4 HW Detection of PV,PS

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

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

6.5 Register file indexing

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

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

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

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

Index = Loop_iterator*Loop_step + Loop_start.

The index is going to return 0 if it is out of the range. We loop until loop_iterator = loop_count. Loop_step is a signed value [-128…127].

6.6 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.**

6.7 Debugging the Shaders

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

6.7.1 *Method 1: Debugging registers*

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

The sequencer will detect the following groups of errors:
- count overflow
~~- jump error~~
~~-relative jump address > size of the control flow program~~
~~-relative jump address > length of the shader program~~
- ~~-~~constant overflow
-
- register overflow
~~- register overflow~~
~~- call stack~~
~~-call with stack full~~
~~-return with stack empty~~
Compiler recognizable errors:
 - jump errors

Formatted: Bullets and Numbering

relative jump address > size of the control flow program
relative jump address > length of the shader program
- call stack
call with stack full
return with stack empty

With two of the errors, a jump error or a register overflow will 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 the other errors, program can continue to run, potentially to worst-case limits.

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

{ISSUE : Interrupt to the driver or not?}

6.7.2 Method 2: Exporting the values in the GPRs (12)

The sequencer will have a 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 :

1) Normal
2) Debug Kill
3) Debug Addr + Count

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 clause 7 is always exporting 12 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:

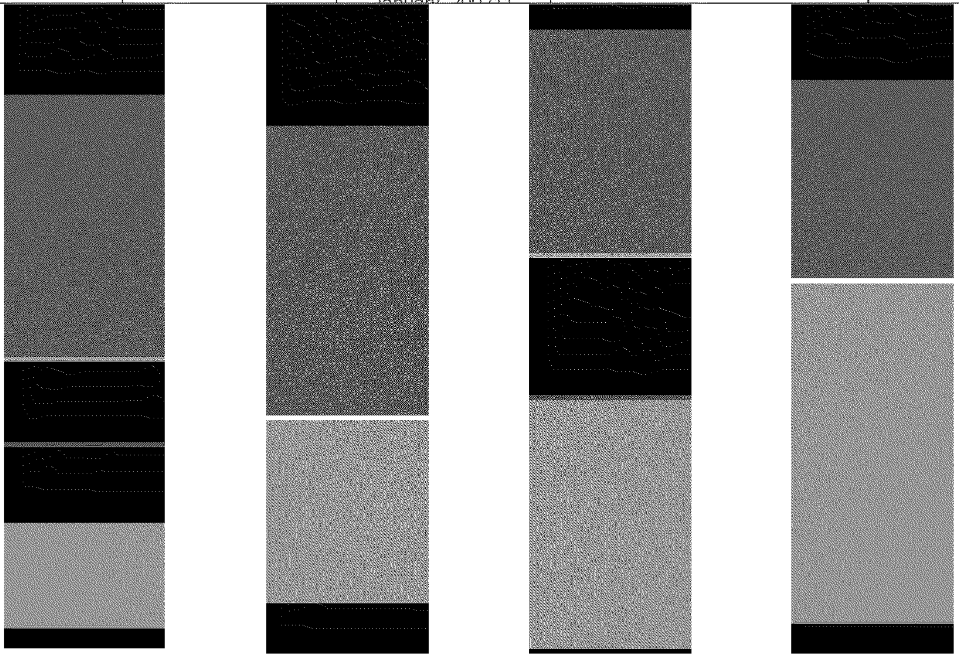
MASK_SETE
MASK_SETNE
MASK_SETGT
MASK_SETGTE

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 VERTEX_REG_SIZE for vertices and 256-VERTEX_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.

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).

$$\begin{aligned}
\Delta 01I &= I(1) - I(0)\\
\Delta 01J &= J(1) - J(0)\\
\Delta 02I &= I(2) - I(0)\\
\Delta 02J &= J(2) - J(0)\\
\Delta 03I &= I(3) - I(0)\\
\Delta 03J &= J(3) - J(0)
\end{aligned}$$

P0	P1
P2	P3

$$\begin{aligned}
P0 &= C + I(0)*(A-C) + J(0)*(B-C)\\
P1 &= P0 + \Delta 01I*(A-C) + \Delta 01J*(B-C)\\
P2 &= P0 + \Delta 02I*(A-C) + \Delta 02J*(B-C)\\
P3 &= P0 + \Delta 03I*(A-C) + \Delta 03J*(B-C)
\end{aligned}$$

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

Formatted: Bullets and Numbering

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

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

The sequencer will re-arrange them in this fashion:

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

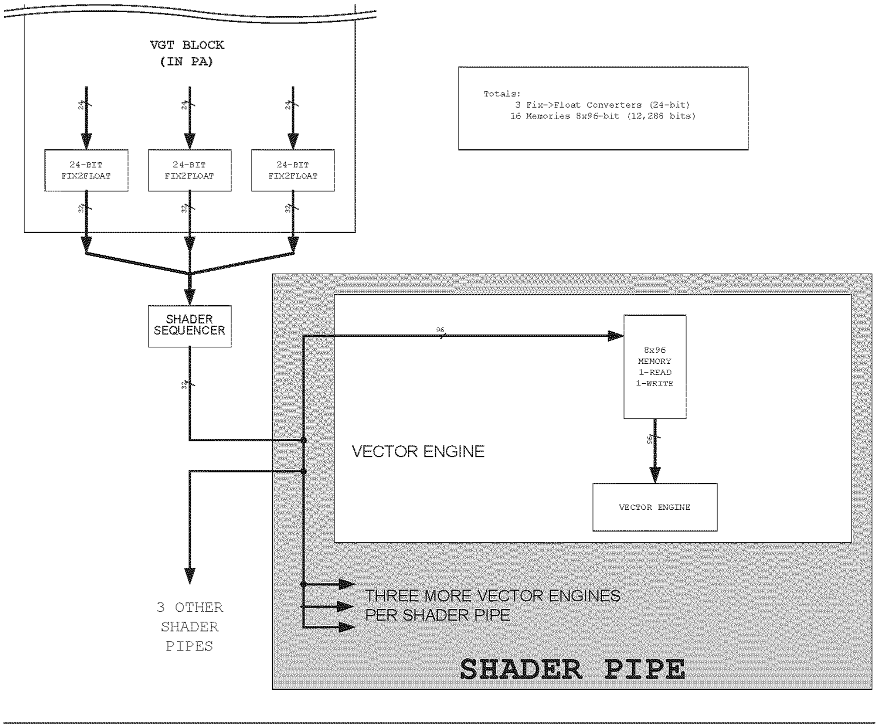
The || markers show the SP divisions. In the event a shader pipe is broken, the VGT will send padding to account for the missing pipe. For example, if SP1 is broken, vertices 4 5 6 7 20 21 22 23 36 37 38 39 52 53 54 55 will still be sent by the VGT to the SQ **BUT** will not be processed by the SP and thus should be considered invalid (by the SU and VGT).

Formatted

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 2. 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 1.

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 1:Area Estimate for VGT to Shader Interface



Totals:
3 Fix->Float Converters (24-bit)
16 Memories 8x96-bit (12,288 bits)

Figure 2:VGT to Shader Interface

Formatted: Bullets and Numbering

~~16.~~17. The parameter cache

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

MEMORY NUMBER 4 bits	ADDRESS 7 bits

The PA generates the parameter cache addresses as the positions comes 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_6 (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_6 = 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 1*VS_EXPORT_COUNT_6 to Current_Location and reset the memory count to 0 before the next vector begins).

Formatted

Formatted

Formatted

Formatted: Bullets and Numbering

~~17.~~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.

Formatted: Bullets and Numbering

~~18.~~19. Exporting Arbitration

Any type of exporting clause can be co-issued. The sequencer will have to make sure back to back memory exports (position/straight memory exports) are interleaved with NOPs as we don't have the bandwidth to service them at full speed.

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

Formatted: Bullets and Numbering

~~1)Position exports and position exports cannot be co-issued.~~
~~2)Position exports and memory exports cannot be co-issued.~~
~~3)Position exports and Z/Color exports cannot be co-issued.~~

~~4)Memory exports and Z/Color exports cannot be co-issued.~~
~~5)Memory exports and memory exports cannot be co-issued.~~
~~6)Z/color exports and Z/color exports cannot be co-issued.~~
~~7)Parameter exports and Z/Color exports CAN be co-issued.~~

~~8)Parameter exports and parameter exports CAN be co-issued.~~
~~9)Parameter exports and memory exports CAN be co-issued.~~

~~19.~~20. 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:

~~19.1~~20.1 Vertex Shading

Formatted: Bullets and Numbering

0:15	- 16 parameter cache
16:31	- Empty (Reserved?)
32:43	- 12 vertex exports to the frame buffer and index
44:47	- Empty
48:59	- 12 debug export (interpret as normal vertex export)
60	- export addressing mode
61	- Empty
62	- sprite size export that goes with position export (point_h,point_w,edgeflag,misc)
63	- position

~~19.2~~20.2 Pixel Shading

Formatted: Bullets and Numbering

0	- Color for buffer 0 (primary)
1	- Color for buffer 1
2	- Color for buffer 2
3	- Color for buffer 3
4:7	- 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?)
32:43	- 12 exports for multipass pixel shaders.
44:47	- Empty
48:59	- 12 debug exports (interpret as normal pixel export)
60	- export addressing mode
61:62	- Empty
63	- Z for primary buffer (Z exported to 'alpha' component)

~~20.~~21. Special Interpolation modes

Formatted: Bullets and Numbering

~~20.1~~21.1 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 realime 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.

~~20.2~~21.2 Sprites/ XY screen coordinates/ FB information

Formatted: Bullets and Numbering

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_I0 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_I0 disable, snd_xy disable, no gen_st – I0 = No modification
Param_Gen_I0 disable, snd_xy disable, gen_st – I0 = No modification
Param_Gen_I0 disable, snd_xy enable, no gen_st – I0 = No modification
Param_Gen_I0 disable, snd_xy enable, gen_st – I0 = No modification
Param_Gen_I0 enable, snd_xy disable, no gen_st – I0 = garbage, garbage, garbage, faceness
Param_Gen_I0 enable, snd_xy disable, gen_st – I0 = garbage, garbage, s, t
Param_Gen_I0 enable, snd_xy enable, no gen_st – I0 = screen x, screen y, garbage, faceness
Param_Gen_I0 enable, snd_xy enable, gen_st – I0 = screen x, screen y, s, t

~~20.3~~21.3 Auto generated counters

Formatted: Bullets and Numbering

In the cases we are dealing with multipass shaders, the sequencer is going to generate a vector count to be able to both use this count to write the 1st pass data to memory and then use the count to retrieve the data on the 2nd pass. The count is always generated in the same way but it is passed to the shader in a slightly different way depending on the shader type (pixel or vertex). This is toggled on and off using the GEN_INDEX 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.

~~20.3.1~~21.3.1 *Vertex shaders*

Formatted: Bullets and Numbering

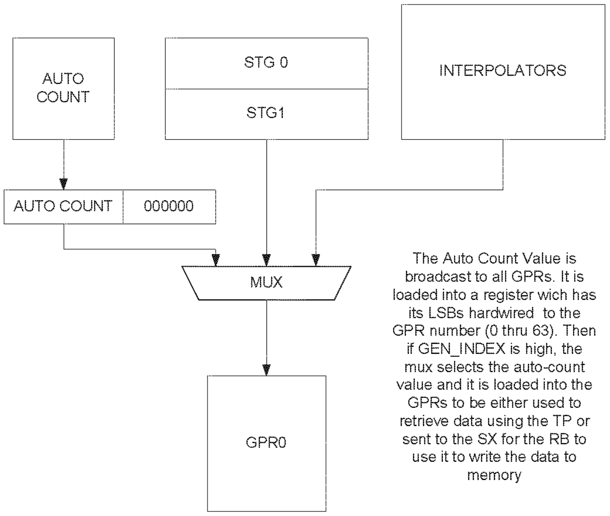
In the case of vertex shaders, if GEN_INDEX is set, the data will be put into the x field of the third register (it means that the compiler must allocate 3 GPRs in all multipass vertex shader modes).

~~20.3.2~~21.3.2 *Pixel shaders*

Formatted: Bullets and Numbering

In the case of pixel shaders, if GEN_INDEX is set and Param_Gen_I0 is enabled, the data will be put in the x field of the 2nd register (R~~I~~1.x), else if GEN_INDEX is set the data will be put into the x field of the 1st register (R0.x).

Formatted



The Auto Count Value is broadcast to all GPRs. It is loaded into a register wich has its LSBs hardwired to the GPR number (0 thru 63). Then if GEN_INDEX is high, the mux selects the auto-count value and it is loaded into the GPRs to be either used to retrieve data using the TP or sent to the SX for the RB to use it to write the data to memory

~~21.~~22. State management

Formatted: Bullets and Numbering

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.

~~21.1~~22.1 Parameter cache synchronization

Formatted: Bullets and Numbering

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.

~~22.~~23. XY Address imports

Formatted: Bullets and Numbering

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 21.22~~1.2~~20.2 for details on how to control the interpolation in this mode.

~~22.1~~23.1 Vertex indexes imports

Formatted: Bullets and Numbering

In order to import vertex indexes, we have ~~64x2x96~~ 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.

~~23.~~24. Registers

Formatted: Bullets and Numbering

~~23.1~~24.1 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_STORE_ALLOC	interleaved, separate
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_ADDR	This is where the CP puts the base address of the instruction writes and type (auto-incremented on reads/writes)
INSTRUCTION_DATA	This is where the CP puts the actual data going to the instruction memory
~~CONSTANT_DATA~~	~~This is where the CP puts constant data (32 bits)~~
~~CONSTANT_ADDR~~	~~This is where the CP puts the logical constant address (9 bits)~~CONSTANTS 512*4 ALU constants + 32*6 Texture state 32 bits registers (logically mapped)
INSTRUCTION_ADDR_RT	This is where the CP puts the base address of the instruction writes and type for Real Time (auto-incremented on reads/writes)
INSTRUCTION_DATA_RT	This is where the CP puts the actual data going to the instruction memory for Real Time
~~CONSTANT_DATA_RT~~	~~This is where the CP puts constant data for Real Time (32 bits)~~

~~CONSTANT_ADDR_RT~~	~~This is where the CP puts the logical constant address for Real Time (9 bits)~~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
EXPORT_LATE	Controls whether or not we are exporting position from clause 3. If set, position exports occur at clause 7.

~~23.2~~24.2 Context

Formatted: Bullets and Numbering

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)
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_MASK	which of the last 6 ALU clauses is exporting (multipass only)
VS_EXPORT_MODE	0: position (1 vector), 1: position (2 vectors), 3:multipass
VS_EXPORT _COUNT_{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
GEN_INDEX	Auto generates an address from 0 to XX. Puts the results into ~~R1~~ R0-1 for pixel shaders and ~~R3~~ 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)

~~24.~~25. DEBUG Registers

Formatted: Bullets and Numbering

~~24.1~~25.1 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_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)

Formatted: Bullets and Numbering

~~25.~~26. Interfaces

~~25.1~~26.1 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.

Formatted: Bullets and Numbering

~~25.1.1~~26.1.1 *SC to SQ : IJ Control bus*

This is the control information sent to the sequencer in order to control the IJ fifos and all other information needed to execute a shader program on the sent pixels. This information is sent over 2 clocks, if SENDXY is asserted the next control packet is going to be ignored and XY information is going to be sent on the IJ bus (for the quads that where just sent). All pixels from the group of quads are from the same primitive, all quads of a vector are from the same render state.

Name	Direction	Bits	Description
SC_SQ_q_wr_mask	SC→SQ	4	Quad Write mask left to right
SC_SQ_lod_correct	SC→SQ	24	LOD correction per quad (6 bits per quad)
SC_SQ_flat_vertex	SC→SQ	2	Provoking vertex for flat shading
SC_SQ_param_ptr0	SC→SQ	11	P Store pointer for vertex 0
SC_SQ_param_ptr1	SC→SQ	11	P Store pointer for vertex 1
SC_SQ_param_ptr2	SC→SQ	11	P Store pointer for vertex 2
SC_SQ_end_of_vect	SC→SQ	1	End of the vector
SC_SQ_store_dealloc	SC→SQ	1	Deallocation token for the P Store
SC_SQ_state	SC→SQ	3	State/constant pointer
SC_SQ_valid_pixel	SC→SQ	16	Valid bits for all pixels
SC_SQ_null_prim	SC→SQ	1	Null Primitive (for PC deallocation purposes)
SC_SQ_end_of_prim	SC→SQ	1	End Of the primitive
SC_SQ_send_xy	SC→SQ	1	Sending XY information [XY information is going to be sent on the next clock]
SC_SQ_prim_type	SC→SQ	3	Real time command need to load tex cords from alternate buffer. Line AA, Point AA and Sprite reads their parameters from GEN_T and GEN_S GPRs. 000 : Normal 011 : Real Time 100 : Line AA 101 : Point AA 110 : Sprite
SC_SQ_new_vector	SC→SQ	1	This primitive comes from a new vector of vertices. Make sure that the corresponding vertex shader has finished before starting the group of pixels.
SC_SQ_RTRn	SQ→SC	1	Stalls the PA in n clocks
SC_SQ_RTS	SC→SQ	1	SC ready to send data

Formatted: Bullets and Numbering

~~25.1.2~~26.1.2 *SQ to SP: Interpolator bus*

Name	Direction	Bits	Description
SQ_SPx_interp_prim_type	SQ→SPx	3	Type of the primitive 000 : Normal 011 : Real Time 100 : Line AA 101 : Point AA 110 : Sprite

SQ_SPx_interp_flat_vtx	SQ→SPx	2	Provoking vertex for flat shading
SQ_SPx_interp_flat_gouraud	SQ→SPx	1	Flat or gouraud shading
SQ_SPx_interp_cyl_wrap	SQ→SPx	4	Wich parameter needs to be cylindrical wrapped
SQ_SPx_interp_ijline	SQ→SPx	2	Line in the IJ/XY buffer to use to interpolate
SQ_SPx_interp_buff_swap	SQ→SPx	1	Swap the IJ/XY buffers at the end of the interpolation
SQ_SPx_interp_gen_I0	SQ→SPx	1	Generate I0 or not. This tells the interpolators not to use the parameter cache but rather overwrite the data with interpolated 1 and 0. Overwrite if gen_I0 is high.

~~25.1.3SQ to SP: GPR Input Mux select~~

Formatted: Bullets and Numbering

~~This interface is synchronized with the Interpolator bus. This controls the input mux to the GPRs. The three types of data are: generated index, Interpolated data, vertex index data (coming from the staging registers).~~

~~25.1.4~~26.1.3 SQ to SP: Parameter Cache Read control bus

Formatted: Bullets and Numbering

The four following interfaces (SQ→SP, SQ→SX,SP→SX and SX→Interpolators) are all SYNCHRONIZED together.

Name	Direction	Bits	Description
SQ_SPx_ptr0	SQ→SPx	9	Pointer of PC
SQ_SPx_ptr1	SQ→SPx	9	Pointer of PC
SQ_SPx_ptr2	SQ→SPx	9	Pointer of PC
SQ_SP0_read_ena	SQ→SP0	4	Read enables for the 4 memories in the SP0
SQ_SP1_read_ena	SQ→SP1	4	Read enables for the 4 memories in the SP1
SQ_SP2_read_ena	SQ→SP2	4	Read enables for the 4 memories in the SP2
SQ_SP3_read_ena	SQ→SP3	4	Read enables for the 4 memories in the SP3

~~25.1.5~~26.1.4 SQ to SX: Parameter Cache Mux control Bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SXx_mux0	SQ→SXx	4	Mux control for PC (4 MSbs of Pointer)
SQ_SXx_mux1	SQ→SXx	4	Mux control for PC (4 MSbs of Pointer)
SQ_SXx_mux2	SQ→SXx	4	Mux control for PC (4 MSbs of Pointer)

~~25.1.6~~26.1.5 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~~0~~_vgt_vsisr_data	SQ→SPx~~0~~	96	Pointers of indexes or HOS surface information
SQ_SP~~0~~x_vgt_vsisr_double	SQ→SPx~~0~~	1	0: Normal 96 bits per vert 1: double 192 bits per vert
SQ_SP0_data_valid	SQ→SP0	1	Data is valid
SQ_SP1_data_valid	SQ→SP1	1	Data is valid
SQ_SP2_data_valid	SQ→SP2	1	Data is valid
SQ_SP3_data_valid	SQ→SP3	1	Data is valid

~~25.1.7~~26.1.6 PA to SQ : Vertex interface

Formatted: Bullets and Numbering

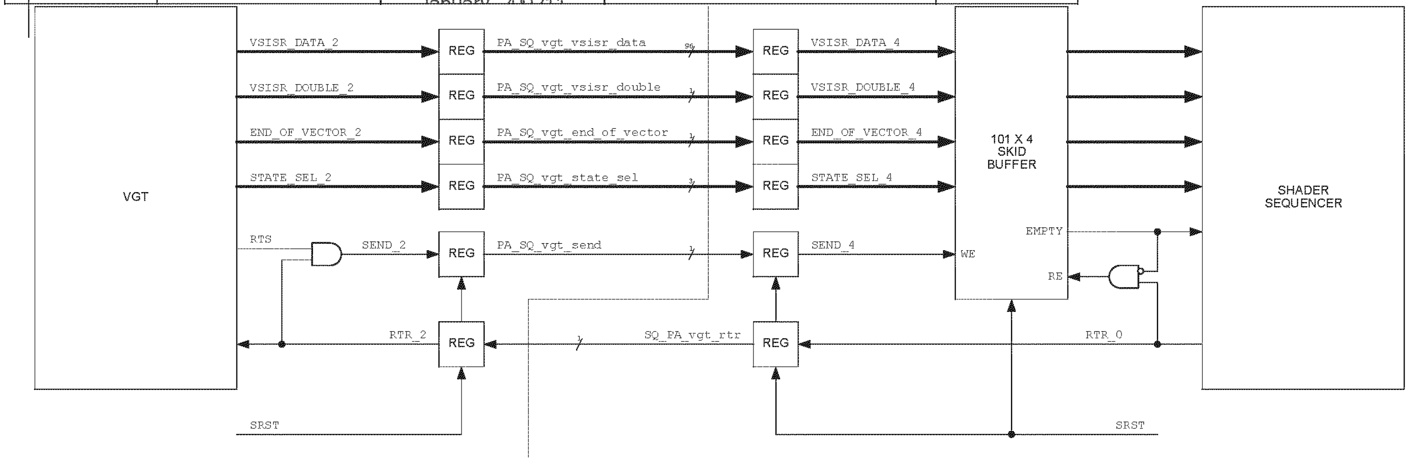
~~25.1.7.1~~26.1.6.1 Interface Signal Table

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
PA_SQ_vgt_vsisr_data	96	Pointers of indexes or HOS surface information
PA_SQ_vgt_vsisr_double	1	0: Normal 96 bits per vert 1: double 192 bits per vert
PA_SQ_vgt_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 second vector)
PA_SQ_vgt_state	3	Render State (6*3+3 for constants). This signal is guaranteed to be correct when "PA_SQ_vgt_end_of_vector" is high.
PA_SQ_vgt_send	1	Data on the VGT_SQ is valid receive (see write-up for standard R400 SEND/RTR interface handshaking)
SQ_PA_vgt_rtr	1	Ready to receive (see write-up for standard R400 SEND/RTR interface handshaking)

~~25.1.7.2~~26.1.6.2 Interface Diagrams

Formatted: Bullets and Numbering


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

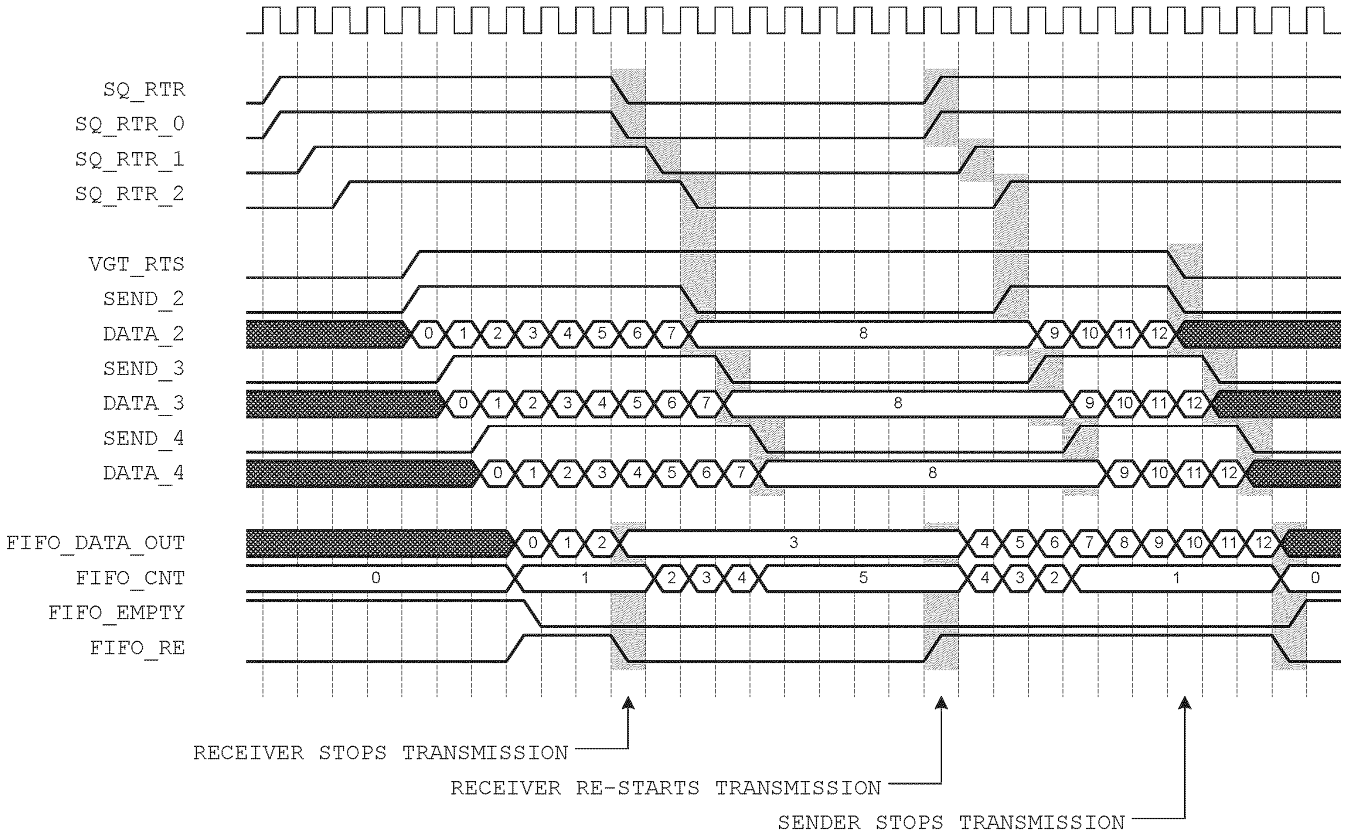


Figure 1. Detailed Logical Diagram for PA SQ vgt Interface.

~~25.1.8~~26.1.7 SQ to CP: State report

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_CP_vrtx_ state	SEQ→CP	3	Oldest vertex state still in the pipe
SQ_CP_pix_state	SEQ→CP	3	Oldest pixel state still in the pipe

~~25.1.9~~26.1.8 SQ to SX: Control bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SXx_exp_Pixel	SQ→SXx	1	1: Pixel 0: Vertex
SQ_SXx_exp_start	SQ→SXx	1	Raised to indicate that the SQ is starting an export
SQ_SXx_exp_Clause	SQ→SXx	3	Clause number, which is needed for vertex clauses
SQ_SXx_exp_State	SQ→SXx	3	State ID, which is needed for vertex clauses

These fields are sent synchronously with SP export data, described in SP0→SX0 interface
{ISSUE: Where are the PC pointers}

~~25.1.10~~26.1.9 SX to SQ : Output file control

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SXx_SQ_Export_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_Export_Position	SXx→SQ	1	Specifies whether there is room for another position.
SXx_SQ_Export_Buffer	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

~~25.1.11~~26.1.10 SQ to TP: Control bus

Formatted: Bullets and Numbering

Once every clock, the fetch unit sends to the sequencer on which clause it is now working and if the data in the GPRs is ready or not. This way the sequencer can update the fetch counters 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	Direction	Bits	Description
TPx_SQ_data_rdy	TPx→ SQ	1	Data ready
TPx_SQ_clause_num	TPx→ SQ	3	Clause number
SQ_TPx_const	SQ→TPx	~~64~~48	Fetch state sent over 4 clocks (192 bits total)
SQ_TPx_instuct	SQ→TPx	24	Fetch instruction sent over 4 clocks
SQ_TPx_end_of_clause	SQ→TPx	1	Last instruction of the clause
SQ_TPx_phase	SQ→TPx	2	Write phase signal
SQ_TP0_lod_correct	SQ→TP0	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP0_pmask	SQ→TP0	4	Pixel mask 1 bit per pixel
SQ_TP1_lod_correct	SQ→TP1	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP1_pmask	SQ→TP1	4	Pixel mask 1 bit per pixel
SQ_TP2_lod_correct	SQ→TP2	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP2_pmask	SQ→TP2	4	Pixel mask 1 bit per pixel
SQ_TP3_lod_correct	SQ→TP3	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP3_pmask	SQ→TP3	4	Pixel mask 1 bit per pixel
SQ_TPx_clause_num	SQ→TPx	3	Clause number
SQ_TPx_write_gpr_index	SQ->TPx	7	Index into Register file for write of returned Fetch Data

~~25.1.12~~26.1.11 *TP to SQ: Texture stall*

Formatted: Bullets and Numbering

The TP sends this signal to the SQ when its input buffer is full. The SQ is going to send it to the SP X clocks after reception (maximum of 3 clocks of pipeline delay).

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

~~25.1.13~~26.1.12 *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

~~25.1.14~~26.1.13 *SQ to SP: GPR, ~~and~~ Parameter cache control and auto counter*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_wr_addr	SQ→SPx	7	Write address
SQ_SPx_gpr_rd_addr	SQ→SPx	7	Read address
SQ_SPx_gpr_re_addr	SQ→SPx	1	Read Enable
SQ_SPx_gpr_we_addr	SQ→SPx	1	Write Enable for the GPRs
SQ_SPx_gpr_phase_mux	SQ→SPx	2	The phase mux (arbitrates between inputs, ALU SRC reads and writes)
SQ_SPx_channel_mask	SQ→SPx	4	The channel mask
SQ_SP0_pixel_mask	SQ→SP0	4	The pixel mask
SQ_SP1_pixel_mask	SQ→SP1	4	The pixel mask
SQ_SP2_pixel_mask	SQ→SP2	4	The pixel mask
SQ_SP3_pixel_mask	SQ→SP3	4	The pixel mask
SQ_SPx_pc_we_addr	SQ→SPx	1	Write Enable for the parameter caches
SQ_SPx_gpr_input_mux	SQ→SPx	2	When the phase mux selects the inputs this tells from which source to read from: Interpolated data, VTX0, VTX1, autogen counter.
SQ_SPx_index_count	SQ→SPx	12?	Index count, common for all shader pipes

~~25.1.15~~26.1.14 SQ to SPx: Instructions

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_instruct_start	SQ→SPx	1	Instruction start
SQ_SP_instruct	SQ→SPx	20	Instruction sent over 4 clocks
SQ_SPx_stall	SQ→SPx	1	Stall signal
SQ_SPx_export_count	SQ→SPx	3	Each set of four pixels or vectors is exported over eight clocks. This field specifies where the SP is in that sequence.
SQ_SPx_export_last	SQ→SPx	1	Asserted on the first shader count of the last export of the clause
SQ_SP0_export_pvalid	SQ→SP0	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP0_export_wvalid	SQ→SP0	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SQ_SP1_ export_pvalid	SQ→SP1	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP1_ export_wvalid	SQ→SP1	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SQ_SP2_ export_pvalid	SQ→SP2	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP2_ export_wvalid	SQ→SP2	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SQ_SP3_ export_pvalid	SQ→SP3	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP3_ export_wvalid	SQ→SP3	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors

~~25.1.16~~26.1.15 SP to SQ: Constant address load

Formatted: Bullets and Numbering

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

~~25.1.17~~26.1.16 SQ to SPx: constant broadcast

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_constant	SQ→SPx	128	Constant broadcast

~~25.1.18~~26.1.17 *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

~~25.1.19~~26.1.18 *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)

~~25.1.20~~26.1.19 *CP to SQ: RBBM bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
rbbm_we	CP→SQ	1	Write Enable
rbbm_a	CP→SQ	18	Address -- Upper Extent is TBD
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

~~26.~~27. Examples of program executions

Formatted: Bullets and Numbering

~~26.1.1~~27.1.1 *Sequencer Control of a Vector of Vertices*

1. PA sends a vector of 64 vertices (actually vertex indices – 32 bits/index for 2048 bit total) to the RE's Vertex FIFO
   - state pointer as well as tag into position cache is sent along with vertices
   - space was allocated in the position cache for transformed position before the vector was sent
   - **also before the vector is sent to the RE, the CP has loaded the global instruction store with the vertex shader program (using the MH?)**
   - The vertex program is assumed to be loaded when we receive the vertex vector.
     - the SEQ then accesses the IS base for this shader using the local state pointer (provided to all sequencers by the RBBM when the CP is done loading the program)
2. SEQ arbitrates between the Pixel FIFO and the Vertex FIFO – basically the Vertex FIFO always has priority
   - at this point the vector is removed from the Vertex FIFO
   - the arbiter is not going to select a vector to be transformed if the parameter cache is full unless the pipe as nothing else to do (ie no pixels are in the pixel fifo).
3. SEQ allocates space in the SP register file for index data plus GPRs used by the program
   - the number of GPRs required by the program is stored in a local state register, which is accessed using the state pointer that came down with the vertices
   - SEQ will not send vertex data until space in the register file has been allocated
4. SEQ sends the vector to the SP register file over the RE_SP interface (which has a bandwidth of 2048 bits/cycle)
   - the 64 vertex indices are sent to the 64 register files over 4 cycles
     - RF0 of SU0, SU1, SU2, and SU3 is written the first cycle
     - RF1 of SU0, SU1, SU2, and SU3 is written the second cycle
     - RF2 of SU0, SU1, SU2, and SU3 is written the third cycle
     - RF3 of SU0, SU1, SU2, and SU3 is written the fourth cycle

- the index is written to the least significant 32 bits **(floating point format?) (what about compound indices)** of the 128-bit location within the register file (w); the remaining data bits are set to zero (x, y, z)

5. SEQ constructs a control packet for the vector and sends it to the first reservation station (the FIFO in front of fetch state machine 0, or TSM0 FIFO)
   - the control packet contains the state pointer, the tag to the position cache and a register file base pointer.
6. TSM0 accepts the control packet and fetches the instructions for fetch clause 0 from the global instruction store
   - TSM0 was first selected by the TSM arbiter before it could start
7. all instructions of fetch clause 0 are issued by TSM0
8. the control packet is passed to the next reservation station (the FIFO in front of ALU state machine 0, or ASM0 FIFO)
   - TSM0 does not wait for requests made to the Fetch Unit to complete; it passes the register file write index for the fetch data to the TU, which will write the data to the RF as it is received
   - once the TU has written all the data to the register files, it increments a counter that is associated with ASM0 FIFO; a count greater than zero indicates that the ALU state machine can go ahead start to execute the ALU clause
9. ASM0 accepts the control packet (after being selected by the ASM arbiter) and gets the instructions for ALU clause 0 from the global instruction store
10. all instructions of ALU clause 0 are issued by ASM0, then the control packet is passed to the next reservation station (the FIFO in front of fetch state machine 1, or TSM1 FIFO)
11. the control packet continues to travel down the path of reservation stations until all clauses have been executed
    - position can be exported in ALU clause 3 (or 4?); the data (and the tag) is sent over a position bus (which is shared with all four shader pipes) back to the PA's position cache
    - A parameter cache pointer is also sent along with the position data. This tells to the PA where the data is going to be in the parameter cache.
      - there is a position export FIFO in the SP that buffers position data before it gets sent back to the PA
      - the ASM arbiter will prevent a packet from starting an exporting clause if the position export FIFO is full
    - parameter data is exported in clause 7 (as well as position data if it was not exported earlier)
      - parameter data is sent to the Parameter Cache over a dedicated bus
      - the SEQ allocates storage in the Parameter Cache, and the SEQ deallocates that space when there is no longer a need for the parameters (it is told by the PA when using a token).
      - the ASM arbiter will prevent a packet from starting on ASM7 if the parameter cache (or the position buffer if position is being exported) is full
12. after the shader program has completed, the SEQ will free up the GPRs so that they can be used by another shader program

~~26.1.2~~27.1.2 *Sequencer Control of a Vector of Pixels*

Formatted: Bullets and Numbering

1. **As with vertex shader programs, pixel shaders are loaded into the global instruction store by the CP**
   - At this point it is assumed that the pixel program is loaded into the instruction store and thus ready to be read.
2. the RE's Pixel FIFO is loaded with the barycentric coordinates for pixel quads by the detailed walker
   - the state pointer and the LOD correction bits are also placed in the Pixel FIFO
   - the Pixel FIFO is wide enough to source four quad's worth of barycentrics per cycle
3. SEQ arbitrates between Pixel FIFO and Vertex FIFO – when there are no vertices pending OR there is no space left in the register files for vertices, the Pixel FIFO is selected
4. SEQ allocates space in the SP register file for all the GPRs used by the program
   - the number of GPRs required by the program is stored in a local state register, which is accessed using the state pointer
   - SEQ will not allow interpolated data to be sent to the shader until space in the register file has been allocated
5. SEQ controls the transfer of interpolated data to the SP register file over the RE_SP interface (which has a bandwidth of 2048 bits/cycle). See interpolated data bus diagrams for details.

6. SEQ constructs a control packet for the vector and sends it to the first reservation station (the FIFO in front of fetch state machine 0, or TSM0 FIFO)
   - note that there is a separate set of reservation stations/arbiters/state machines for vertices and for pixels
   - the control packet contains the state pointer, the register file base pointer, and the LOD correction bits
   - all other information (such as quad address for example) travels in a separate FIFO
7. TSM0 accepts the control packet and fetches the instructions for fetch clause 0 from the global instruction store
   - TSM0 was first selected by the TSM arbiter before it could start
8. all instructions of fetch clause 0 are issued by TSM0
9. the control packet is passed to the next reservation station (the FIFO in front of ALU state machine 0, or ASM0 FIFO)
   - TSM0 does not wait for fetch requests made to the Fetch Unit to complete; it passes the register file write index for the fetch data to the TU, which will write the data to the RF as it is received
   - once the TU has written all the data for a particular clause to the register files, it increments a counter that is associated with the ASM0 FIFO; a count greater than zero indicates that the ALU state machine can go ahead and pop the FIFO and start to execute the ALU clause
10. ASM0 accepts the control packet (after being selected by the ASM arbiter) and gets the instructions for ALU clause 0 from the global instruction store
11. all instructions of ALU clause 0 are issued by ASM0, then the control packet is passed to the next reservation station (the FIFO in front of fetch state machine 1, or TSM1 FIFO)
12. the control packet continues to travel down the path of reservation stations until all clauses have been executed
    - pixel data is exported in the last ALU clause (clause 7)
      - it is sent to an output FIFO where it will be picked up by the render backend
      - the ASM arbiter will prevent a packet from starting on ASM7 if the output FIFO is full
13. after the shader program has completed, the SEQ will free up the GPRs so that they can be used by another shader program

~~26.1.3~~27.1.3 *Notes*

Formatted: Bullets and Numbering

14. The state machines and arbiters will operate ahead of time so that they will be able to immediately start the real threads or stall.
15. The register file base pointer for a vector needs to travel with the vector through the reservation stations, but the instruction store base pointer does not – this is because the RF pointer is different for all threads, but the IS pointer is only different for each state and thus can be accessed via the state pointer.

~~27.~~28. Open issues

Formatted: Bullets and Numbering

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

Saving power?

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 20154 ~~February, 20029~~	GEN-CXXXXX-REVA	1 of 50

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SQ

Version 1.7~~6~~

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 CONTA~~read_ptr~~INS CONFIDENTIAL INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTA~~read_ptr~~L TO THE INTEREST OF ATI TECHNOLOGIES INC. THROUGH UNAUTHORIZED USE OR DISCLOSURE.

ATI 2023
LG v. ATI
IPR2015-00325

Table Of Contents

Revision Changes:

Rev 0.1 (Laurent Lefebvre) Date: May 7, 2001	First draft.
Rev 0.2 (Laurent Lefebvre) Date : July 9, 2001	Changed the interfaces to reflect the changes in the SP. Added some details in the arbitration section.
Rev 0.3 (Laurent Lefebvre) Date : August 6, 2001	Reviewed the Sequencer spec after the meeting on August 3, 2001.
Rev 0.4 (Laurent Lefebvre) Date : August 24, 2001	Added the dynamic allocation method for register file and an example (written in part by Vic) of the flow of pixels/vertices in the sequencer.
Rev 0.5 (Laurent Lefebvre) Date : September 7, 2001	Added timing diagrams (Vic)
Rev 0.6 (Laurent Lefebvre) Date : September 24, 2001	Changed the spec to reflect the new R400 architecture. Added interfaces.
Rev 0.7 (Laurent Lefebvre) Date : October 5, 2001	Added constant store management, instruction store management, control flow management and data dependant predication.
Rev 0.8 (Laurent Lefebvre) Date : October 8, 2001	Changed the control flow method to be more flexible. Also updated the external interfaces.
Rev 0.9 (Laurent Lefebvre) Date : October 17, 2001	Incorporated changes made in the 10/18/01 control flow meeting. Added a NOP instruction, removed the conditional_execute_or_jump. Added debug registers.
Rev 1.0 (Laurent Lefebvre) Date : October 19, 2001	Refined interfaces to RB. Added state registers.
Rev 1.1 (Laurent Lefebvre) Date : October 26, 2001	Added SEQ→SP0 interfaces. Changed delta precision. Changed VGT→SP0 interface. Debug Methods added.
Rev 1.2 (Laurent Lefebvre) Date : November 16, 2001	Interfaces greatly refined. Cleaned up the spec.
Rev 1.3 (Laurent Lefebvre) Date : November 26, 2001	Added the different interpolation modes.
Rev 1.4 (Laurent Lefebvre) Date : December 6, 2001	Added the auto incrementing counters. Changed the VGT→SQ interface. Added content on constant management. Updated GPRs.
Rev 1.5 (Laurent Lefebvre) Date : December 11, 2001	Removed from the spec all interfaces that weren't directly tied to the SQ. Added explanations on constant management. Added PA→SQ synchronization fields and explanation.
Rev 1.6 (Laurent Lefebvre) Date : January 7, 2002	Added more details on the staging register. Added detail about the parameter caches. Changed the call instruction to a Conditionnal_call instruction. Added details on constant management and updated the diagram.
Rev 1.7 (Laurent Lefebvre) Date : February 4, 2002	Added Real Time parameter control in the SX interface. Updated the control flow section.

1. Overview

The sequencer is based on the R300 design. It chooses two ALU clauses and a fetch clause to execute, and executes all of the instructions in a clause before looking for a new clause of the same type. Two ALU clauses 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. 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.

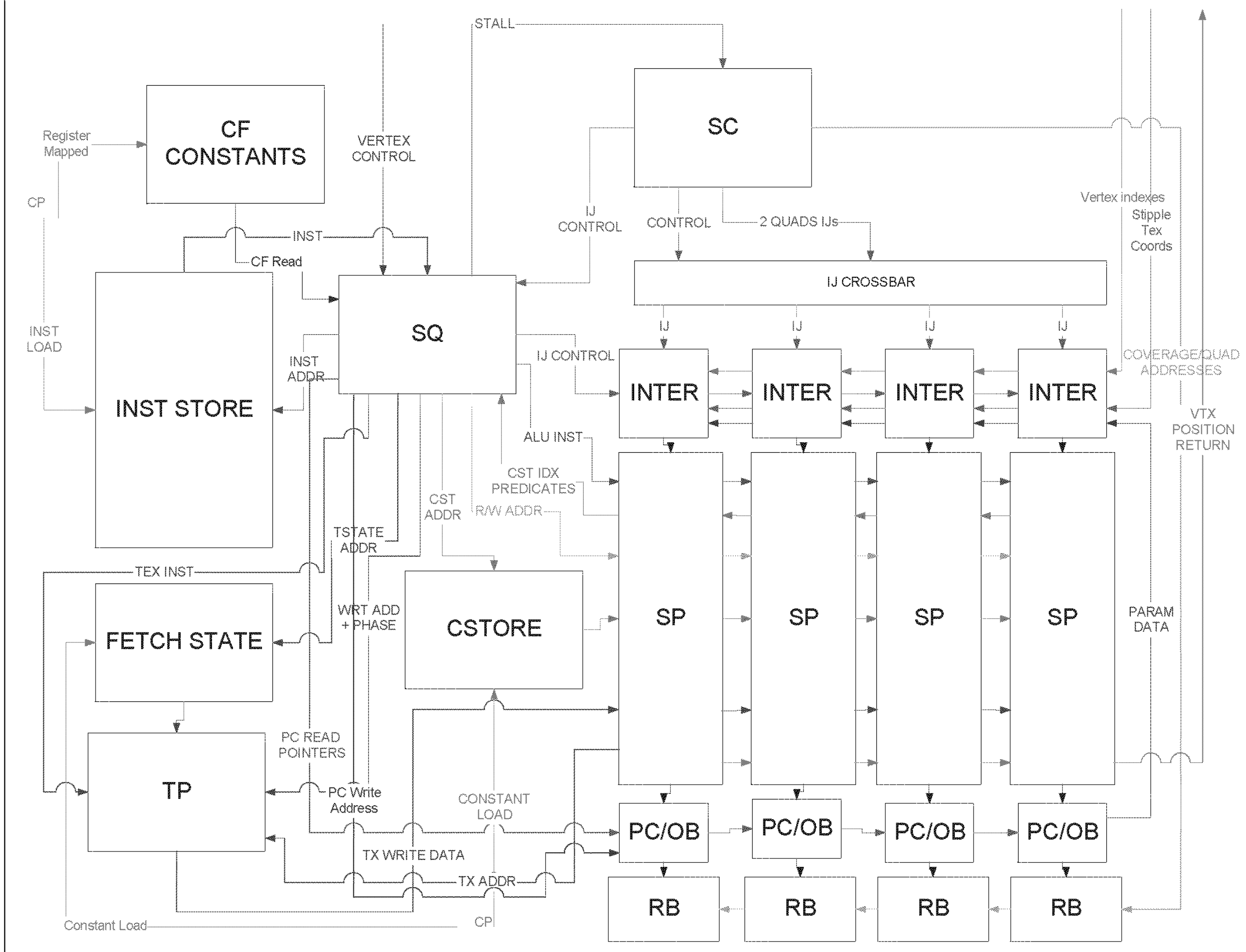


Figure 1: General Sequencer overview

1.1 Top Level Block Diagram

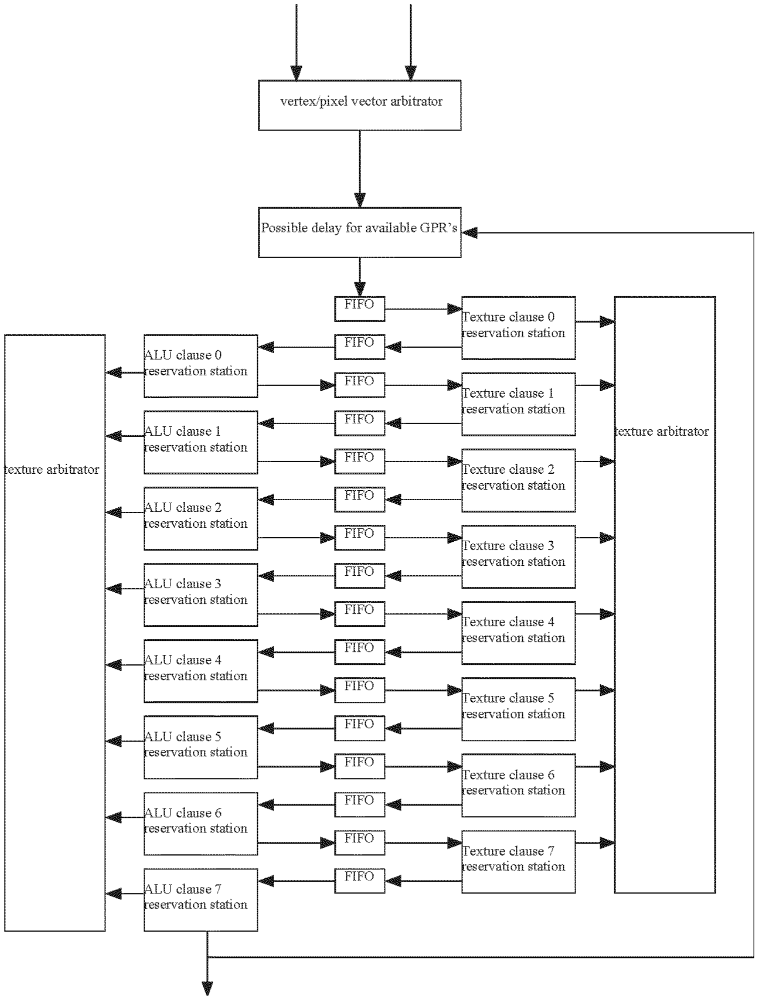

vertex/pixel vector arbitrator
Possible delay for available GPR's
FIFO
Texture clause 0 reservation station
Texture clause 1 reservation station
Texture clause 2 reservation station
Texture clause 3 reservation station
Texture clause 4 reservation station
Texture clause 5 reservation station
Texture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

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~~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.

1.2 Data Flow graph (SP)

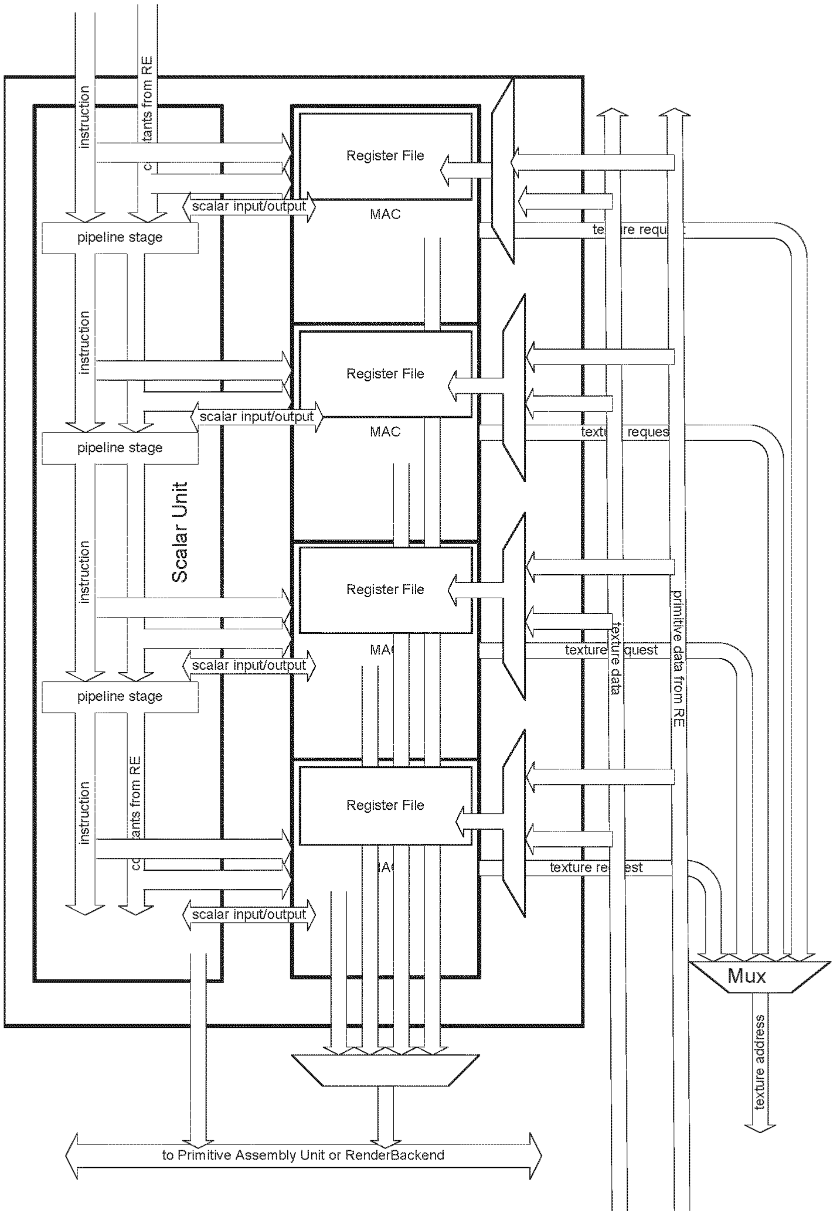


Figure 3: The shader Pipe

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

1.3 Control Graph

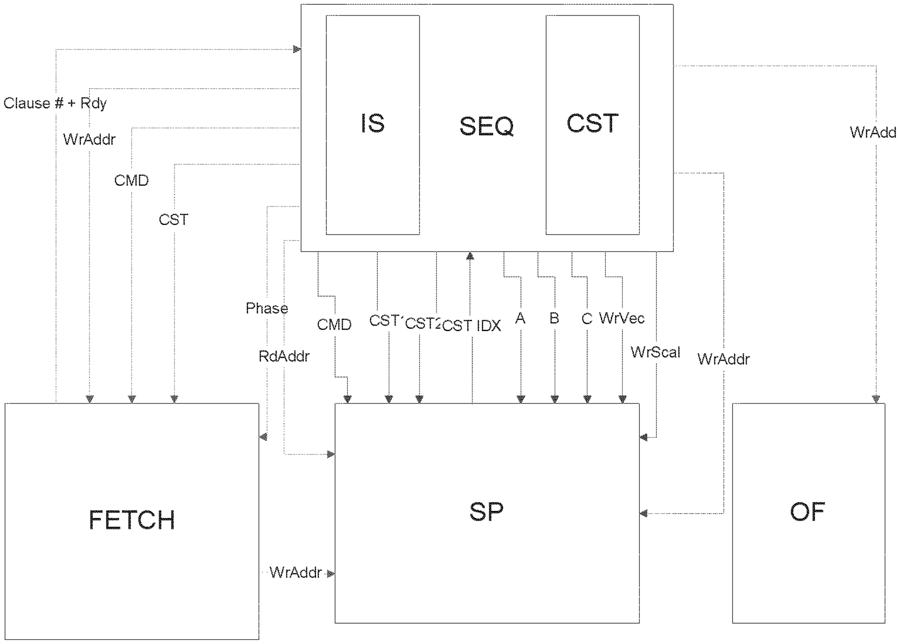


Figure 4: Sequencer Control interfaces

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.

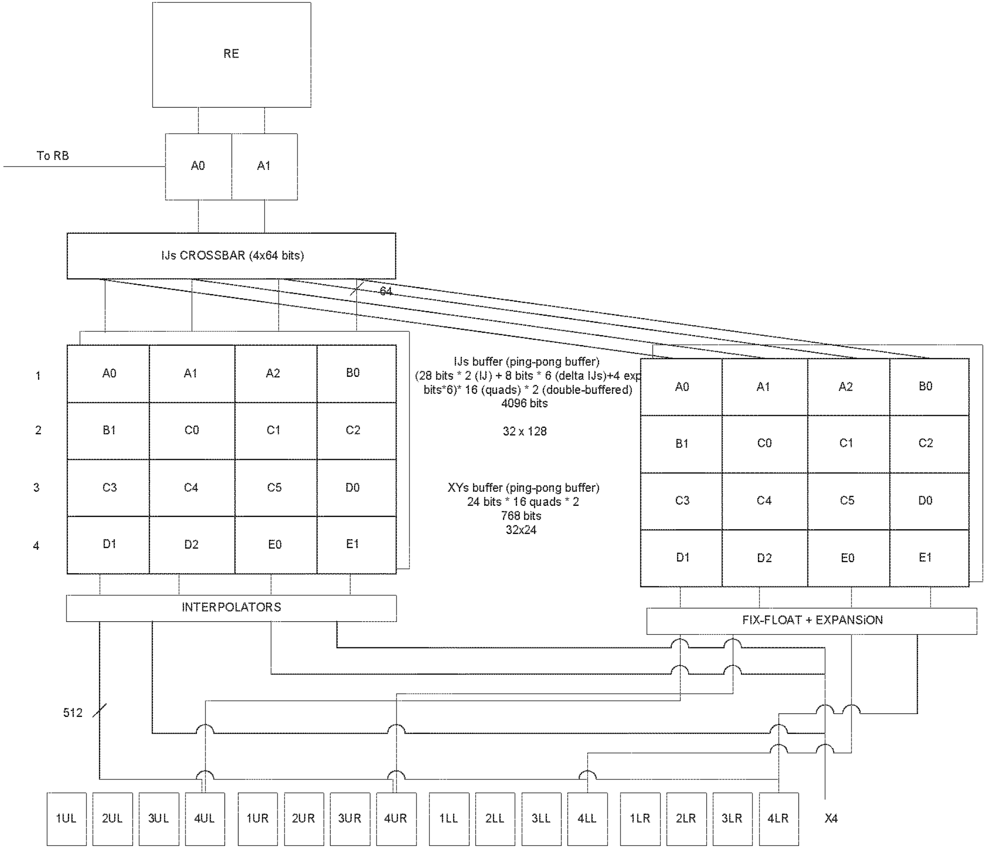


Figure 5: Interpolation buffers

	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.

{ISSUE : Do we do the center + centroid approach using both IJ buffers?}

3. Instruction Store

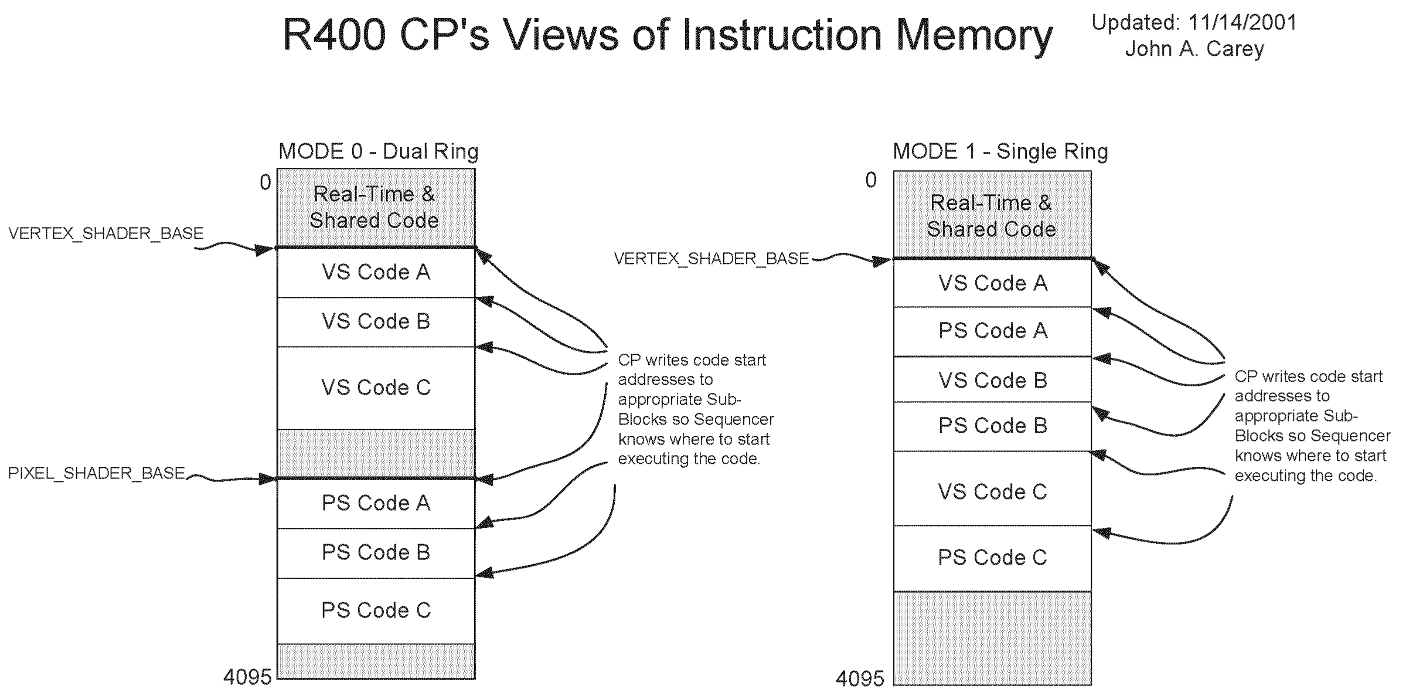
There is going to be only one instruction store for the whole chip. It will contain 4096 instructions of 96 bits each.

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 next picture shows the various modes the CP can load the memory. The Sequencer has to keep track of the loading modes in order to wrap around the correct boundaries. The wrap--around points are arbitrary and they are specified in the VS_BASE and PIX_BASE control 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.



R400 CP's Views of Instruction Memory

Updated: 11/14/2001
John A. Carey

Figure 7: The CP's view of the instruction memory

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 ~~192x128~~128x192 bits. 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 ~~16~~ 32 lines (each line addresses ~~2~~ 1 texture state lines in the real memory). The CP write granularity is ~~2~~ 1 textur~~ee~~ state lines (or ~~384~~ 192 bits). The driver sends 512 bits but the CP ignores the top ~~128~~ 320 bits. It thus takes ~~12~~ 6 clocks to write the ~~two~~ texture state~~s~~. 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 state change. 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 re-mapping tables

5.2.1 *R400 Constant management*

Formatted: Bullets and Numbering

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.

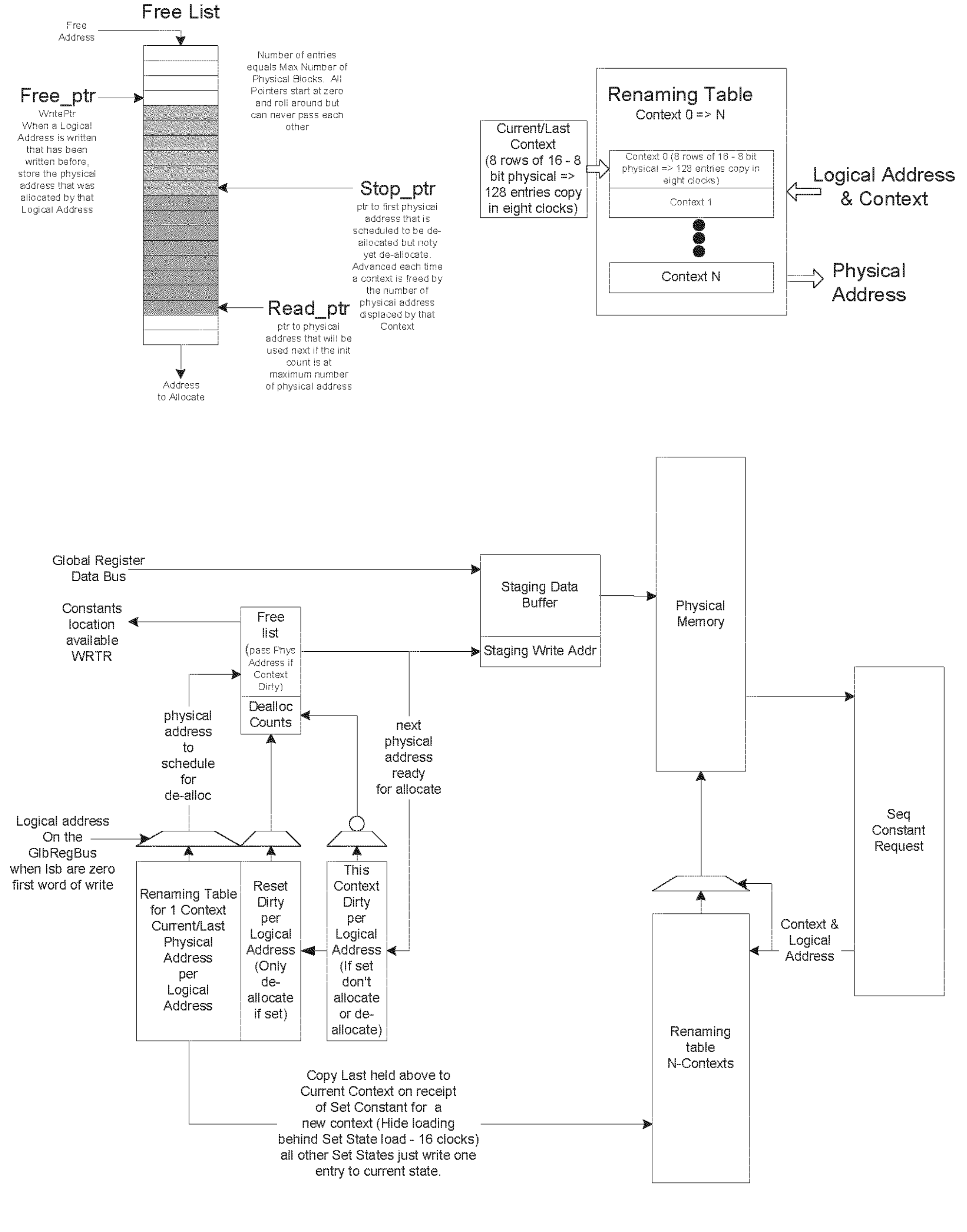
For this model to work in its simplest form, the requirement is that the physical memory MUST be at least twice as large as the logical address space + the space allocated for Real Time. In our case, since the logical address space is 512 and the reserved RT space can be up to 256 entries, the memory must be of sizes ~~1024~~ 1280 and above. Similarly the size of the texture store must be of 32*2+32 = 96 entries and above.

~~5.2.1~~5.2.2 *Proposal for R400LE constant management*

Formatted: Bullets and Numbering

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 9: De-allocation mechanism~~Figure 9: De-allocation mechanismFigure 2: De-allocation mechanism Figure 1: Dealocation 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). ~~Be careful to set only those bits that the CURENT STATE IS USING (if for example the current state uses only 64 constants we set only lines 0 thru 15 to 1). This is ok to do so because the blocks are idle and thus the context has finished drawing.~~



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

Figure 8: Constant management

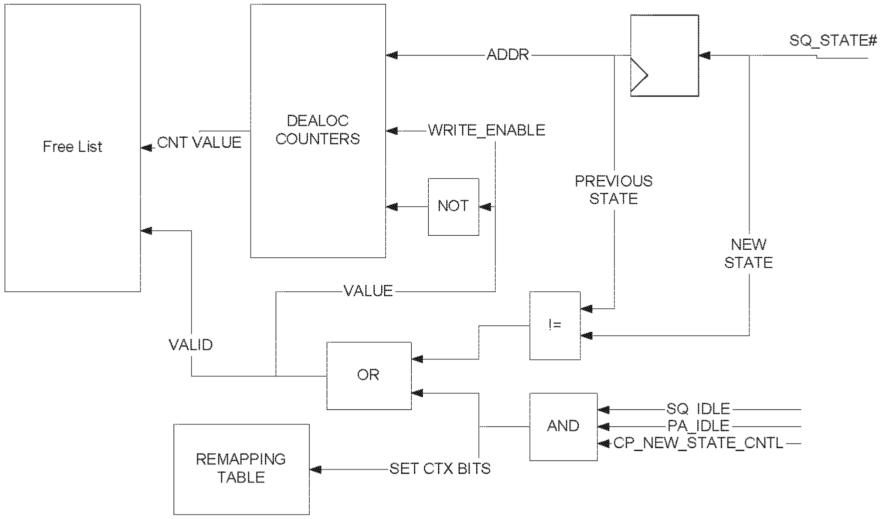


Figure 9: De-allocation mechanism for R400LE

~~5.2.1~~5.2.3 *Dirty bits*

Formatted: Bullets and Numbering

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 when ever 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.

~~5.2.2~~5.2.4 *Free List Block*

Formatted: Bullets and Numbering

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.

~~5.2.3~~5.2.5 *De-allocate Block*

Formatted: Bullets and Numbering

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.

~~5.2.4~~5.2.6 *Operation of Incremental model*

Formatted: Bullets and Numbering

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.

5.3 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.

5.4 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.

5.5 Constant Waterfalling

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

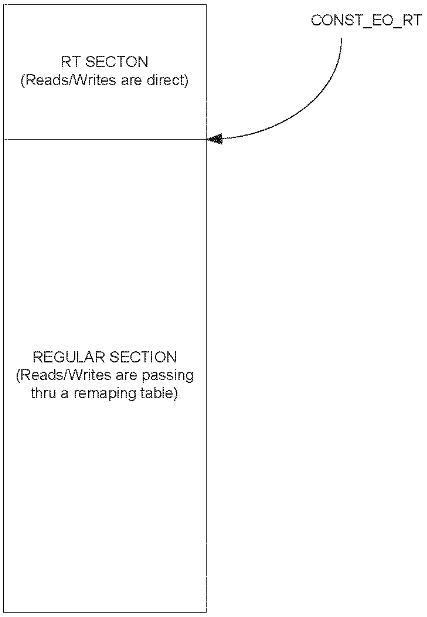

CONST_EO_RT
RT SECTON
(Reads/Writes are direct)
REGULAR SECTION
(Reads/Writes are passing thru a remaping table)

Figure 10: 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

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 eleven basic instructions:

Execute
Conditional_execute
Conditional_Execute_Predicates
Conditional_jump
Conditionnal_Call
Return
Loop_start
Loop_end
End_of_clause
Conditional_End_of_clause
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_or_Jump executes a block of instructions or jumps to an address is the condition is not met.~~
Conditional_execute_Predicates executes a block of instructions if all bits in the predicate vectors meet the condition.
End_of_clause marks the end of a clause.
Conditional_End_of_clause marks the end of a clause if the condition is met.
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.

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… 42	41 … 24	23 … 12	11 … 0
Addressing	00001	RESERVED	Instruction count	Exec Address

Execute up to 4k instructions at the specified address in the instruction memory.

NOP		
47	46 … 42	41 … 0
Addressing	00010	RESERVED

This is a regular NOP.

Conditional_Execute							
47	46 … 42	41	40 … 33	32	31 … 24	23 … 12	11 … 0
Addressing	00011	RESERVED	Boolean address	Condition	RESERVED	Instruction count	Exec Address

If the specified Boolean (8 bits can address 256 Booleans) meets the specified condition then execute the specified instructions (up to 4k instructions)

Conditional_Execute_Predicates							
47	46 … 42	41 … 35	34 … 33	32	31 … 24	23 … 12	11 … 0
Addressing	00100	RESERVED	Predicate vector	Condition	RESERVED	Instruction 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.

Loop_Start				
47	46 … 42	41 … 17	16 … 12	11 … 0
Addressing	00101	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 … 42	41 … 17	16 … 12	11 … 0
Addressing	00110~~1~~	RESERVED	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.

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 … 42	41 … 35	34 … 33	32	31 … 12	11 … 0
Addressing	0~~1~~0111~~0~~ ~~00~~	RESERVED	Predicate vector	Condition	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 … 42	41 … 0
Addressing	01000~~00~~ ~~1~~	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 … 42	41	40 … 33	32	31	30 … 12	11 … 0
Addressing	01001~~10~~	RESERVED	Boolean address	Condition	FW only	RESERVED	Jump address

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.

Conditional_End_of_Clause					
47	46 … 42	41	40 … 33	32	31 … 0
Addressing	01010~~1~~	RESERVED	Boolean address	Condition	RESERVED

This is an optimization in the case of very short shaders (where the control flow instruction can't be hidden anymore and thus are not free. In this case, if the condition is met, the clause is ended, else we continue the execution of the clause.

End_of_Clause		
47	46 … 42	41 … 0
Addressing	01011~~0~~ ~~11~~	RESERVED

Marks the end of a clause.

To prevent infinite loops, we will keep ~~9~~ 9 bits loop ~~counters~~ 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 Data dependant predicate instructions

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

PRED_SETE_# - similar to SETE except that the result is 'exported' to the sequencer.

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

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

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

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

P0_ADD_# R0,R1,R2

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

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

6.4 HW Detection of PV,PS

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

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

6.5 Register file indexing

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

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

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

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

Index = Loop_iterator*Loop_step + Loop_start.

~~The index is going to return 0 if it is out of the range.~~
We loop until loop_iterator = loop_count. Loop_step is a signed value [-128…127]. The computed index value is a 10 bit counter that is also signed. Its real range is [-256,256]. The tenth bit is only there so that we can provide an out of range value to the "indexing logic" so that it knows when the provided index is out of range and thus can make the necessary arrangements.

6.6 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.**

6.7 Debugging the Shaders

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

6.7.1 *Method 1: Debugging registers*

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

The sequencer will detect the following groups of errors:
- count overflow
-- constant indexing overflow
- register indexing overflow

Formatted: Bullets and Numbering

Compiler recognizable errors:
- jump errors
 relative jump address > size of the control flow program
 ~~relative jump address > length of the shader program~~
-- call stack
 call with stack full
 return with stack empty

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

{ISSUE : Interrupt to the driver or not?}

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

The sequencer will have a 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 :
1) Normal
2) Debug Kill
3) Debug Addr + Count

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 clause 7 is always exporting 12 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:

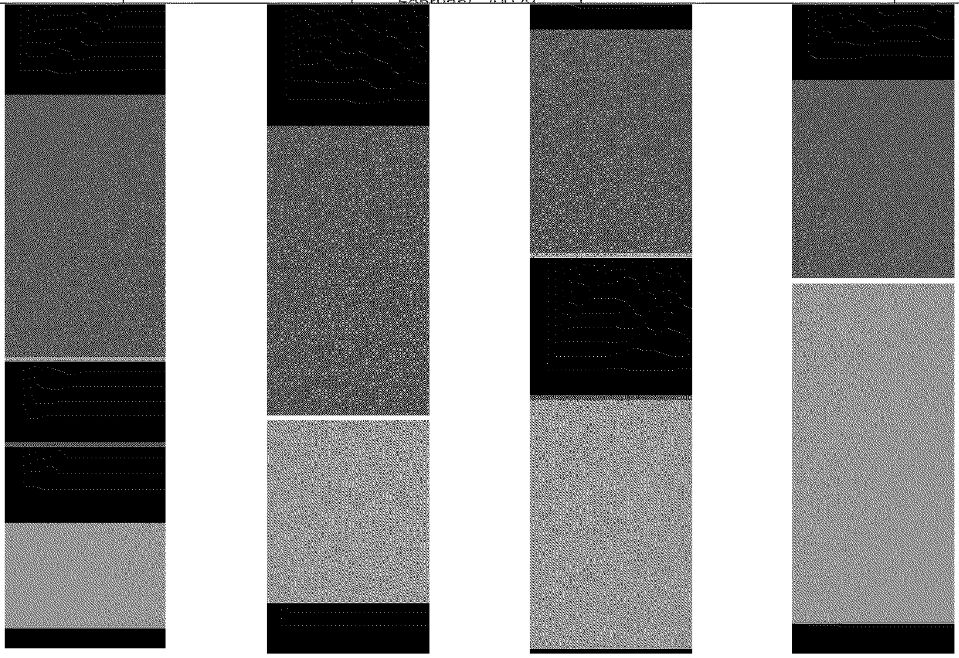
MASK_SETE
MASK_SETNE
MASK_SETGT
MASK_SETGTE

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 ~~256-~~ ~~VERTEX~~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:

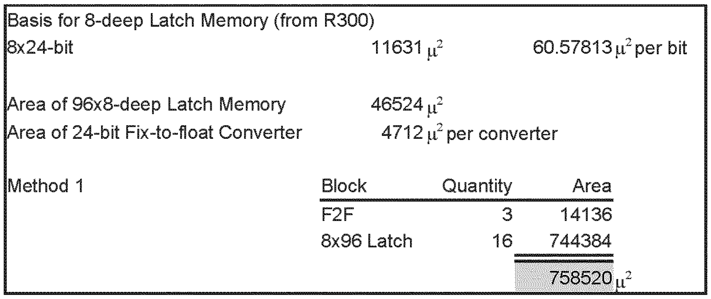
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 || 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 || 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 || 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

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 12~~Figure 12Figure 2~~. 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 11~~Figure 11Figure 1~~.



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

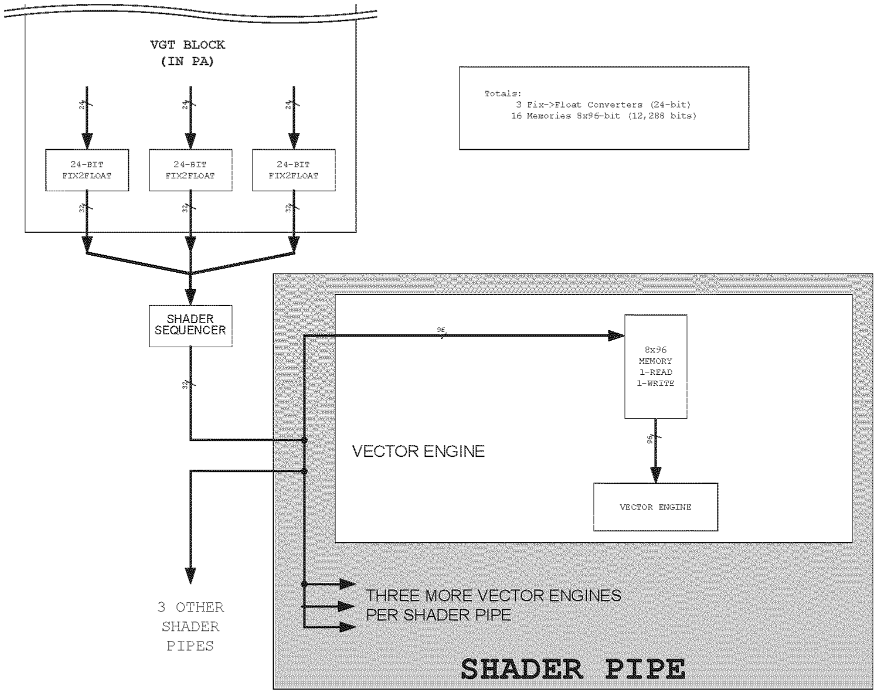


Figure 12~~122~~: VGT to Shader Interface

17. The parameter cache

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

MEMORY NUMBER 4 bits	ADDRESS 7 bits

The PA generates the parameter cache addresses as the positions come~~s~~ 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_~~6~~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_~~6~~ 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 ~~1~~2*VS_EXPORT_COUNT_~~6~~ 7to Current_Location and reset the memory count to 0 before the next vector begins).

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.

Formatted: Bullets and Numbering

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

{ISSUE: Do we move the parameter caches to the SX?}

~~Any type of exporting clause can be co-issued. The sequencer will have to make sure back to back memory exports (position/straight memory exports) are interleaved with NOPs as we don't have the bandwidth to service them at full speed.~~

20. 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:

20.1 Vertex Shading

0:15	- 16 parameter cache
16:31	- Empty (Reserved?)
32:43	- 12 vertex exports to the frame buffer and index
44:47	- Empty
48:59	- 12 debug export (interpret as normal vertex export)
60	- export addressing mode
61	- Empty
62	- position ~~sprite size export that goes with position export (point_h,point_w,edgeflag,misc)~~
63	- ~~position~~sprite size export that goes with position export (point_h,point_w,edgeflag,misc)

20.2 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?)
32:43	- 12 exports for multipass pixel shaders.
44:47	- Empty
48:59	- 12 debug exports (interpret as normal pixel export)
60	- export addressing mode
61:62	- Empty
63	- Z for primary buffer (Z exported to 'alpha' component)

21. Special Interpolation modes

21.1 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 realtime 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.

21.2 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_I0 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_I0 disable, snd_xy disable, no gen_st – I0 = No modification
Param_Gen_I0 disable, snd_xy disable, gen_st – I0 = No modification
Param_Gen_I0 disable, snd_xy enable, no gen_st – I0 = No modification
Param_Gen_I0 disable, snd_xy enable, gen_st – I0 = No modification
Param_Gen_I0 enable, snd_xy disable, no gen_st – I0 = garbage, garbage, garbage, faceness
Param_Gen_I0 enable, snd_xy disable, gen_st – I0 = garbage, garbage, s, t
Param_Gen_I0 enable, snd_xy enable, no gen_st – I0 = screen x, screen y, garbage, faceness
Param_Gen_I0 enable, snd_xy enable, gen_st – I0 = screen x, screen y, s, t

21.3 Auto generated counters

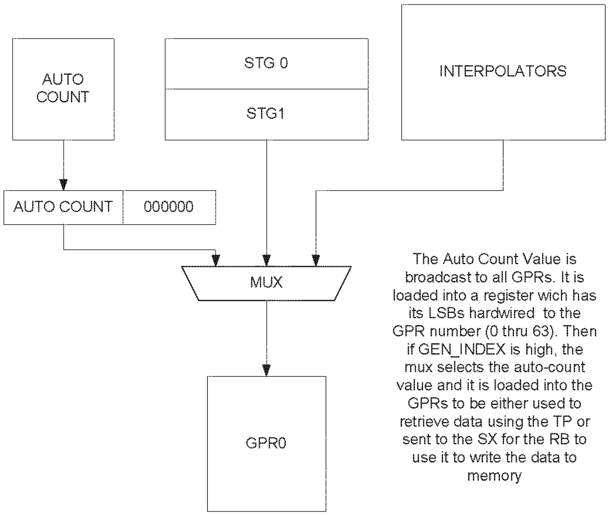
In the cases we are dealing with multipass shaders, the sequencer is going to generate a vector count to be able to both use this count to write the 1st pass data to memory and then use the count to retrieve the data on the 2nd pass. The count is always generated in the same way but it is passed to the 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.

21.3.1 *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).

21.3.2 *Pixel shaders*

In the case of pixel shaders, if GEN_INDEX is set and Param_Gen_I0 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).



The Auto Count Value is broadcast to all GPRs. It is loaded into a register wich has its LSBs hardwired to the GPR number (0 thru 63). Then if GEN_INDEX is high, the mux selects the auto-count value and it is loaded into the GPRs to be either used to retrieve data using the TP or sent to the SX for the RB to use it to write the data to memory

Figure 13: GPR input mux Control

22. 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.

22.1 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.

23. 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 21.2 for details on how to control the interpolation in this mode.

23.1 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.

24. Registers

24.1 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_STORE_ALLOC interleaved, separate
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~~_ADDR~~ This is where the CP puts the base address of the instruction writes and type (auto-incremented on reads/writes) Register mapped
~~INSTRUCTION_DATA This is where the CP puts the actual data going to the instruction memory~~
CONSTANTS 512*4 ALU constants + 32*6 Texture state 32 bits registers (logically mapped)
~~INSTRUCTION_ADDR_RT This is where the CP puts the base address of the instruction writes and type for Real Time (auto-incremented on reads/writes)~~
~~INSTRUCTION_DATA_RT This is where the CP puts the actual data going to the instruction memory for Real Time~~
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_LATE Controls whether or not we are exporting position from clause 3. If set, position exports occur at clause 7.

24.2 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)
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_MASK which of the last 6 ALU clauses is exporting (multipass only)
VS_EXPORT_MODE 0: position (1 vector), 1: position (2 vectors), 3:multipass
VS_EXPORT

_COUNT_{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
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)

25. DEBUG Registers

25.1 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_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)

25.2 Control

Formatted: Bullets and Numbering

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

26. Interfaces

26.1 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.

26.1.1 *SC to SQ : IJ Control bus*

This is the control information sent to the sequencer in order to control the IJ fifos and all other information needed to execute a shader program on the sent pixels. This information is sent over 2 clocks, if SENDXY is asserted the next control packet is going to be ignored and XY information is going to be sent on the IJ bus (for the quads that where just sent). All pixels from the group of quads are from the same primitive, all quads of a vector are from the same render state.

Name	Direction	Bits	Description
SC_SQ_q_wr_mask	SC→SQ	4	Quad Write mask left to right
SC_SQ_lod_correct	SC→SQ	24	LOD correction per quad (6 bits per quad)
SC_SQ_flat_vertex	SC→SQ	2	Provoking vertex for flat shading
SC_SQ_param_ptr0	SC→SQ	11	P Store pointer for vertex 0
SC_SQ_param_ptr1	SC→SQ	11	P Store pointer for vertex 1
SC_SQ_param_ptr2	SC→SQ	11	P Store pointer for vertex 2
SC_SQ_end_of_vect	SC→SQ	1	End of the vector
SC_SQ_store_dealloc	SC→SQ	1	Deallocation token for the P Store
SC_SQ_state	SC→SQ	3	State/constant pointer
SC_SQ_valid_pixel	SC→SQ	16	Valid bits for all pixels
SC_SQ_null_prim	SC→SQ	1	Null Primitive (for PC deallocation purposes)
SC_SQ_end_of_prim	SC→SQ	1	End Of the primitive
SC_SQ_send_xy	SC→SQ	1	Sending XY information [XY information is going to be sent on the next clock]
SC_SQ_prim_type	SC→SQ	3	Real time command need to load tex cords from alternate buffer. Line AA, Point AA and Sprite reads their parameters from GEN_T and GEN_S GPRs. 000 : Normal 011 : Real Time 100 : Line AA 101 : Point AA 110 : Sprite
SC_SQ_new_vector	SC→SQ	1	This primitive comes from a new vector of vertices. Make sure that the corresponding vertex shader has finished before starting the group of pixels.
SC_SQ_RTRn	SQ→SC	1	Stalls the PA in n clocks
SC_SQ_RTS	SC→SQ	1	SC ready to send data

26.1.2 *SQ to SP: Interpolator bus*

Name	Direction	Bits	Description
SQ_SPx_interp_prim_type	SQ→SPx	3	Type of the primitive 000 : Normal 011 : Real Time 100 : Line AA 101 : Point AA 110 : Sprite
SQ_SPx_interp_ijline	SQ→SPx	2	Line in the IJ/XY buffer to use to interpolate
SQ_SPx_interp_buff_swap	SQ→SPx	1	Swap the IJ/XY buffers at the end of the interpolation
SQ_SPx_interp_gen_I0	SQ→SPx	1	Generate I0 or not. This tells the interpolators not to use the parameter cache but rather overwrite the data with interpolated 1 and 0. Overwrite if gen_I0 is high.

26.1.3 *SQ to SX: Interpolator bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_interp_flat_vtx	SQ→SPx	2	Provoking vertex for flat shading
SQ_SPx_interp_flat_gouraud	SQ→SPx	1	Flat or gouraud shading
SQ_SPx_interp_cyl_wrap	SQ→SPx	4	Wich channel needs to be cylindrical wrapped

~~26.1.3~~26.1.4 *SQ to SP: Parameter Cache Read control bus*

Formatted: Bullets and Numbering

The four following interfaces (SQ→SP, SQ→SX,SP→SX and SX→Interpolators) are all SYNCHRONIZED together.

Name	Direction	Bits	Description
SQ_SPx_ptr0~~SQ_SPx_ptr0~~	SQ→SPx~~SQ→SPx~~	7~~9~~	Parameter Pointer into PC ~~Pointer of PC~~
SQ_SPx_ptr1~~SQ_SPx_ptr1~~	SQ→SPx~~SQ→SPx~~	7~~9~~	Parameter Pointer into PC ~~Pointer of PC~~

SQ_SPx_ptr2~~SQ_SPx_ptr2~~	SQ→SPx~~SQ→SPx~~	79	Parameter Pointer into Parameter Cache~~Pointer of PC~~
SQ_SPx_pc0_addr_sel~~SQ_SP0_read_ena~~	SQ→SPx~~SQ→SP0~~	24	Selection one of the pointers for parameter cache 0~~Read enables for the 4 memories in the SP0~~
SQ_SPx_pc1_addr_sel~~SQ_SP1_read_ena~~	SQ→SPx~~SQ→SP1~~	24	Selection one of the pointers for parameter cache 1~~Read enables for the 4 memories in the SP1~~
SQ_SPx_pc2_addr_sel~~SQ_SP2_read_ena~~	SQ→SPx~~SQ→SP2~~	24	Selection one of the pointers for parameter cache 2~~Read enables for the 4 memories in the SP2~~
SQ_SPx_pc3_addr_sel~~SQ_SP3_read_ena~~	SQ→SPx~~SQ→SP3~~	24	Selection one of the pointers for parameter cache 3~~Read enables for the 4 memories in the SP3~~
SQ_SP0_read_ena	SQ→SP0	4	Read enables for the 4 memories in the SP0
SQ_SP1_read_ena	SQ→SP1	4	Read enables for the 4 memories in the SP1
SQ_SP2_read_ena	SQ→SP2	4	Read enables for the 4 memories in the SP2
SQ_SP3_read_ena	SQ→SP3	4	Read enables for the 4 memories in the SP3

~~26.1.4~~26.1.5 *SQ to SX: Parameter Cache Mux control Bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SXx_mux0	SQ→SXx	4	Mux control for PC or RT (4 MSbs of Pointer in the PC case)
SQ_SXx_mux1	SQ→SXx	4	Mux control for PC or RT (4 MSbs of Pointer in the PC case)
SQ_SXx_mux2	SQ→SXx	4	Mux control for PC or RT (4 MSbs of Pointer in the PC case)
SQ_SXx_RT_switch	SQ→SXx	1	Selects between RT and Normal data

~~26.1.5~~26.1.6 *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_vgt_vsisr_data	SQ→SPx	96	Pointers of indexes or HOS surface information
SQ_SPx_vgt_vsisr_double	SQ→SPx	1	0: Normal 96 bits per vert 1: double 192 bits per vert
SQ_SP0_data_valid	SQ→SP0	1	Data is valid
SQ_SP1_data_valid	SQ→SP1	1	Data is valid
SQ_SP2_data_valid	SQ→SP2	1	Data is valid
SQ_SP3_data_valid	SQ→SP3	1	Data is valid

~~26.1.6~~26.1.7 *PA to SQ : Vertex interface*

Formatted: Bullets and Numbering

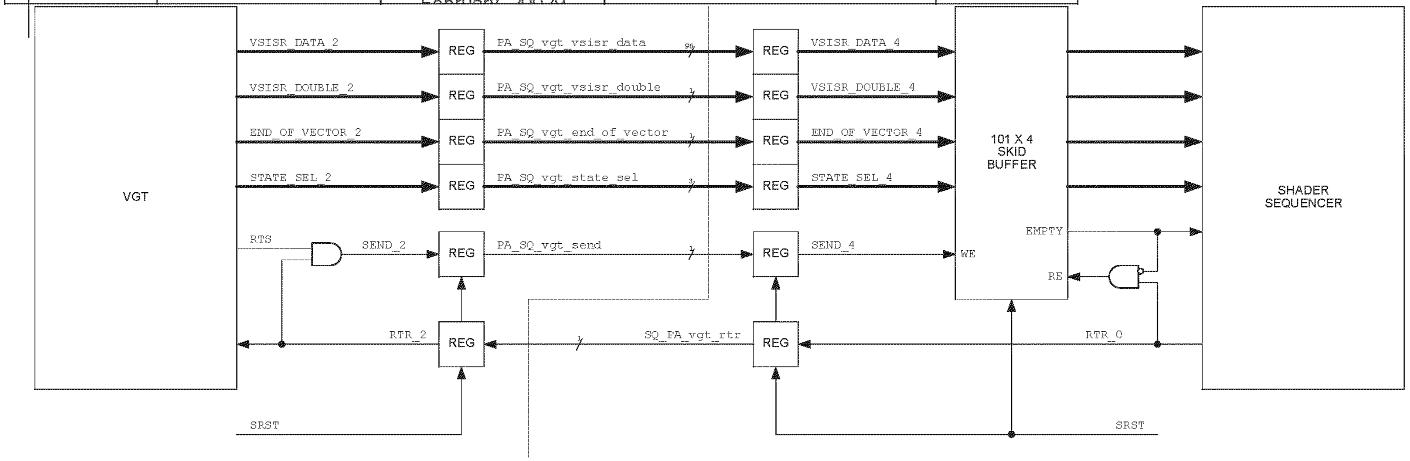
~~26.1.6.1~~26.1.7.1 Interface Signal Table

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
PA_SQ_vgt_vsisr_data	9966	Pointers of indexes or HOS surface information
PA_SQ_vgt_vsisr_double	1	0: Normal ~~96~~ 96 bits per vert 1: double ~~192~~ 192 bits per vert
PA_SQ_vgt_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 second vector)
PA_SQ_vgt_vsisr_valid	1	Vsisr data is valid
PA_SQ_vgt_state	3	Render State (6*3+3 for constants). This signal is guaranteed to be correct when "PA_SQ_vgt_end_of_vector" is high.
PA_SQ_vgt_send	1	Data on the VGT_SQ is valid receive (see write-up for standard R400 SEND/RTR interface handshaking)
SQ_PA_vgt_rtr	1	Ready to receive (see write-up for standard R400 SEND/RTR interface handshaking)

~~26.1.6.2~~26.1.7.2 Interface Diagrams

Formatted: Bullets and Numbering


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

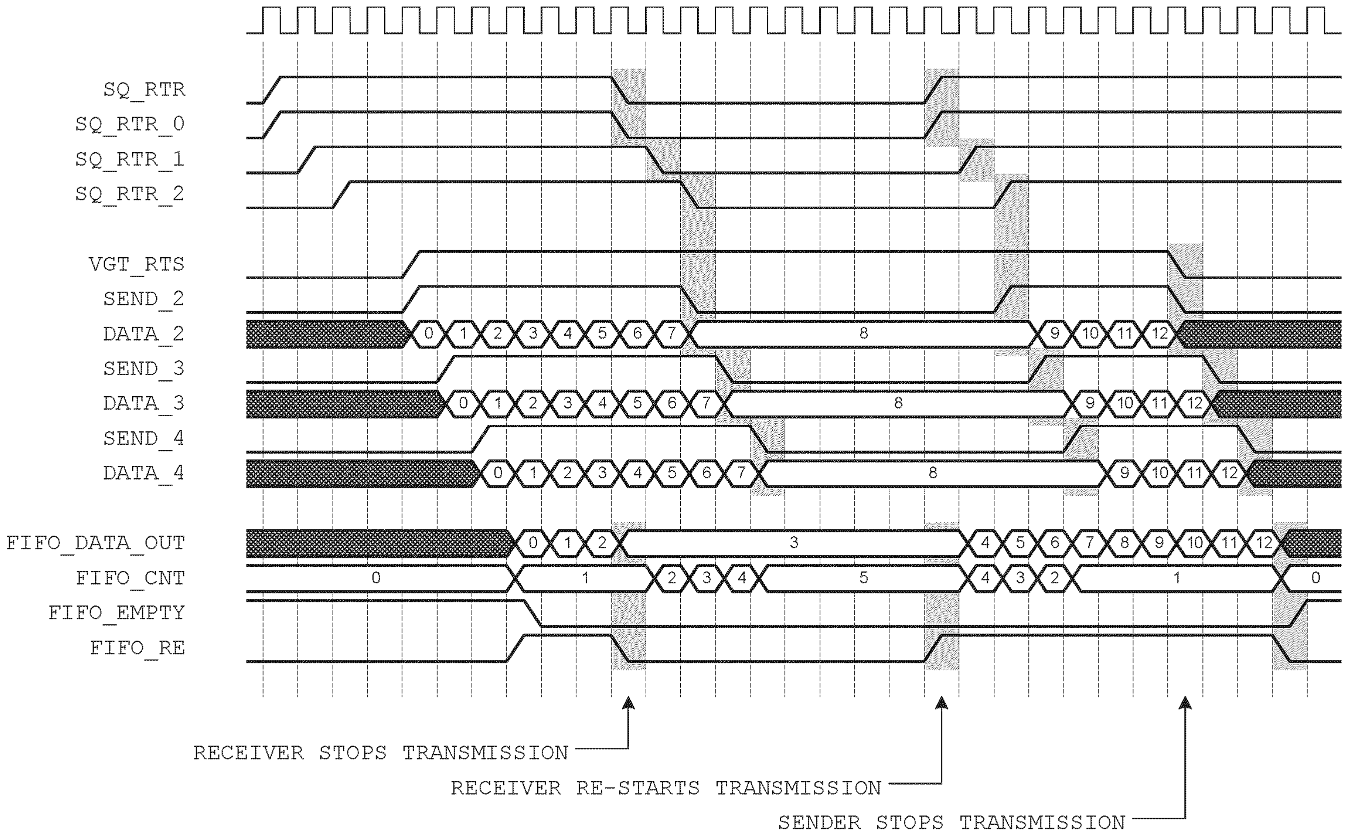


Figure 1. Detailed Logical Diagram for PA SQ vgt Interface.

~~26.1.7~~26.1.8 *SQ to CP: State report*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_CP_vrtx_ state	SEQ→CP	3	Oldest vertex state still in the pipe
SQ_CP_pix_state	SEQ→CP	3	Oldest pixel state still in the pipe

~~26.1.8~~26.1.9 *SQ to SX: Control bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SXx_exp_Pixel	SQ→SXx	1	1: Pixel 0: Vertex
SQ_SXx_exp_start	SQ→SXx	1	Raised to indicate that the SQ is starting an export
SQ_SXx_exp_Clause	SQ→SXx	3	Clause number, which is needed for vertex clauses
SQ_SXx_exp_State	SQ→SXx	3	State ID, which is needed for vertex clauses

These fields are sent synchronously with SP export data, described in SP0→SX0 interface
{ISSUE: Where are the PC pointers}

~~26.1.9~~26.1.10 *SX to SQ : Output file control*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SXx_SQ_Export_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_Export_Position	SXx→SQ	1	Specifies whether there is room for another position.
SXx_SQ_Export_Buffer	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

~~26.1.10~~26.1.11 *SQ to TP: Control bus*

Formatted: Bullets and Numbering

Once every clock, the fetch unit sends to the sequencer on which clause it is now working and if the data in the GPRs is ready or not. This way the sequencer can update the fetch counters 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	Direction	Bits	Description
TPx_SQ_data_rdy	TPx→ SQ	1	Data ready
TPx_SQ_clause_num	TPx→ SQ	3	Clause number
TPx_SQ_Type~~TPx_SQ_clause_num~~	TPx→ SQ~~TPx→ SQ~~	13	Type of data sent (0:PIXEL, 1:VERTEX)~~Clause number~~
SQ_TPx_const	SQ→TPx	48	Fetch state sent over 4 clocks (192 bits total)
SQ_TPx_instuct	SQ→TPx	24	Fetch instruction sent over 4 clocks
SQ_TPx_end_of_clause	SQ→TPx	1	Last instruction of the clause
SQ_TPx_Type	SQ→TPx	1	Type of data sent (0:PIXEL, 1:VERTEX)
SQ_TPx_phase	SQ→TPx	2	Write phase signal
SQ_TP0_lod_correct	SQ→TP0	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP0_pmask	SQ→TP0	4	Pixel mask 1 bit per pixel
SQ_TP1_lod_correct	SQ→TP1	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP1_pmask	SQ→TP1	4	Pixel mask 1 bit per pixel
SQ_TP2_lod_correct	SQ→TP2	6	LOD correct 3 bits per comp 2 components per quad

SQ_TP2_pmask	SQ→TP2	4	Pixel mask 1 bit per pixel
SQ_TP3_lod_correct	SQ→TP3	6	LOD correct 3 bits per comp 2 components per quad
SQ_TP3_pmask	SQ→TP3	4	Pixel mask 1 bit per pixel
SQ_TPx_clause_num	SQ→TPx	3	Clause number
SQ_TPx_write_gpr_index	SQ->TPx	7	Index into Register file for write of returned Fetch Data

~~26.1.11~~26.1.12 *TP to SQ: Texture stall*

Formatted: Bullets and Numbering

The TP sends this signal to the SQ when its input buffer is full. The SQ is going to send it to the SP X clocks after reception (maximum of 3 clocks of pipeline delay).

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

~~26.1.12~~26.1.13 *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

~~26.1.13~~26.1.14 *SQ to SP: GPR, Parameter cache control and auto counter*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_wr_addr	SQ→SPx	7	Write address
SQ_SPx_gpr_rd_addr	SQ→SPx	7	Read address
SQ_SPx_gpr_re_addr	SQ→SPx	1	Read Enable
SQ_SPx_gpr_we_addr	SQ→SPx	1	Write Enable for the GPRs
SQ_SPx_gpr_phase_mux	SQ→SPx	2	The phase mux (arbitrates between inputs, ALU SRC reads and writes)
SQ_SPx_channel_mask	SQ→SPx	4	The channel mask
SQ_SP0_pixel_mask	SQ→SP0	4	The pixel mask
SQ_SP1_pixel_mask	SQ→SP1	4	The pixel mask
SQ_SP2_pixel_mask	SQ→SP2	4	The pixel mask
SQ_SP3_pixel_mask	SQ→SP3	4	The pixel mask
SQ_SPx_pc_we_addr	SQ→SPx	1	Write Enable for the parameter caches
SQ_SPx_gpr_input_mux	SQ→SPx	2	When the phase mux selects the inputs this tells from which source to read from: Interpolated data, VTX0, VTX1, autogen counter.
SQ_SPx_index_count	SQ→SPx	12?	Index count, common for all shader pipes

~~26.1.14~~26.1.15 SQ to SPx: Instructions

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_instruct_start	SQ→SPx	1	Instruction start
SQ_SP_instruct	SQ→SPx	21~~20~~	Transferred over 4 cycles 0: SRC A Select 2:0 SRC A Argument Modifier 3:3 SRC A swizzle 11:4 Unused 20:12 -- - 1: SRC B Select 2:0 SRC B Argument Modifier 3:3 SRC B swizzle 11:4 Unused 20:12 -- - 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~~Instruction sent over 4 clocks~~
SQ_SPx_stall	SQ→SPx	1	Stall signal
SQ_SPx_export_count	SQ→SPx	3	Each set of four pixels or vectors is exported over eight clocks. This field specifies where the SP is in that sequence.
SQ_SPx_export_last	SQ→SPx	1	Asserted on the first shader count of the last export of the clause
SQ_SP0_export_pvalid	SQ→SP0	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP0_export_wvalid	SQ→SP0	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SQ_SP1_ export_pvalid	SQ→SP1	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP1_ export_wvalid	SQ→SP1	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SQ_SP2_ export_pvalid	SQ→SP2	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per clock
SQ_SP2_ export_wvalid	SQ→SP2	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors
SQ_SP3_ export_pvalid	SQ→SP3	4	Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers). 4x4 because 16 pixels are computed per

			clock
SQ_SP3_ export_wvalid	SQ→SP3	2	Specifies whether to write low and/or high 32-bit word of the 64-bit export data from each of the 16 pixels or vectors

~~26.1.15~~26.1.16 *SP to SQ: Constant address load/ Predicate Set*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SP0_SQ_const_addr	SP0→SQ	36	Constant address load / predicate vector load (4 bits only) 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

~~26.1.16~~26.1.17 *SQ to SPx: constant broadcast*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
SQ_SPx_constant	SQ→SPx	128	Constant broadcast

~~26.1.17~~26.1.18 *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

~~26.1.18~~26.1.19 *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)

~~26.1.19~~26.1.20 *CP to SQ: RBBM bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
rbbm_we	CP→SQ	1	Write Enable
rbbm_a	CP→SQ	~~18~~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. Examples of program executions

27.1.1 *Sequencer Control of a Vector of Vertices*

1. PA sends a vector of 64 vertices (actually vertex indices – 32 bits/index for 2048 bit total) to the RE's Vertex FIFO
   - state pointer as well as tag into position cache is sent along with vertices
   - space was allocated in the position cache for transformed position before the vector was sent
   - **also before the vector is sent to the RE, the CP has loaded the global instruction store with the vertex shader program (using the MH?)**
   - The vertex program is assumed to be loaded when we receive the vertex vector.
     - the SEQ then accesses the IS base for this shader using the local state pointer (provided to all sequencers by the RBBM when the CP is done loading the program)
2. SEQ arbitrates between the Pixel FIFO and the Vertex FIFO – basically the Vertex FIFO always has priority
   - at this point the vector is removed from the Vertex FIFO
   - the arbiter is not going to select a vector to be transformed if the parameter cache is full unless the pipe as nothing else to do (ie no pixels are in the pixel fifo).
3. SEQ allocates space in the SP register file for index data plus GPRs used by the program
   - the number of GPRs required by the program is stored in a local state register, which is accessed using the state pointer that came down with the vertices
   - SEQ will not send vertex data until space in the register file has been allocated
4. SEQ sends the vector to the SP register file over the RE_SP interface (which has a bandwidth of 2048 bits/cycle)
   - the 64 vertex indices are sent to the 64 register files over 4 cycles
     - RF0 of SU0, SU1, SU2, and SU3 is written the first cycle
     - RF1 of SU0, SU1, SU2, and SU3 is written the second cycle
     - RF2 of SU0, SU1, SU2, and SU3 is written the third cycle
     - RF3 of SU0, SU1, SU2, and SU3 is written the fourth cycle
   - the index is written to the least significant 32 bits **(floating point format?) (what about compound indices)** of the 128-bit location within the register file (w); the remaining data bits are set to zero (x, y, z)
5. SEQ constructs a control packet for the vector and sends it to the first reservation station (the FIFO in front of fetch state machine 0, or TSM0 FIFO)
   - the control packet contains the state pointer, the tag to the position cache and a register file base pointer.
6. TSM0 accepts the control packet and fetches the instructions for fetch clause 0 from the global instruction store
   - TSM0 was first selected by the TSM arbiter before it could start
7. all instructions of fetch clause 0 are issued by TSM0
8. the control packet is passed to the next reservation station (the FIFO in front of ALU state machine 0, or ASM0 FIFO)
   - TSM0 does not wait for requests made to the Fetch Unit to complete; it passes the register file write index for the fetch data to the TU, which will write the data to the RF as it is received
   - once the TU has written all the data to the register files, it increments a counter that is associated with ASM0 FIFO; a count greater than zero indicates that the ALU state machine can go ahead start to execute the ALU clause
9. ASM0 accepts the control packet (after being selected by the ASM arbiter) and gets the instructions for ALU clause 0 from the global instruction store
10. all instructions of ALU clause 0 are issued by ASM0, then the control packet is passed to the next reservation station (the FIFO in front of fetch state machine 1, or TSM1 FIFO)
11. the control packet continues to travel down the path of reservation stations until all clauses have been executed
    - position can be exported in ALU clause 3 (or 4?); the data (and the tag) is sent over a position bus (which is shared with all four shader pipes) back to the PA's position cache
    - A parameter cache pointer is also sent along with the position data. This tells to the PA where the data is going to be in the parameter cache.
      - there is a position export FIFO in the SP that buffers position data before it gets sent back to the PA

    - the ASM arbiter will prevent a packet from starting an exporting clause if the position export FIFO is full
  - parameter data is exported in clause 7 (as well as position data if it was not exported earlier)
    - parameter data is sent to the Parameter Cache over a dedicated bus
    - the SEQ allocates storage in the Parameter Cache, and the SEQ deallocates that space when there is no longer a need for the parameters (it is told by the PA when using a token).
    - the ASM arbiter will prevent a packet from starting on ASM7 if the parameter cache (or the position buffer if position is being exported) is full

12. after the shader program has completed, the SEQ will free up the GPRs so that they can be used by another shader program

27.1.2 *Sequencer Control of a Vector of Pixels*

1. **As with vertex shader programs, pixel shaders are loaded into the global instruction store by the CP**
   - At this point it is assumed that the pixel program is loaded into the instruction store and thus ready to be read.
2. the RE's Pixel FIFO is loaded with the barycentric coordinates for pixel quads by the detailed walker
   - the state pointer and the LOD correction bits are also placed in the Pixel FIF0
   - the Pixel FIFO is wide enough to source four quad's worth of barycentrics per cycle
3. SEQ arbitrates between Pixel FIFO and Vertex FIFO – when there are no vertices pending OR there is no space left in the register files for vertices, the Pixel FIFO is selected
4. SEQ allocates space in the SP register file for all the GPRs used by the program
   - the number of GPRs required by the program is stored in a local state register, which is accessed using the state pointer
   - SEQ will not allow interpolated data to be sent to the shader until space in the register file has been allocated
5. SEQ controls the transfer of interpolated data to the SP register file over the RE_SP interface (which has a bandwidth of 2048 bits/cycle). See interpolated data bus diagrams for details.
6. SEQ constructs a control packet for the vector and sends it to the first reservation station (the FIFO in front of fetch state machine 0, or TSM0 FIFO)
   - note that there is a separate set of reservation stations/arbiters/state machines for vertices and for pixels
   - the control packet contains the state pointer, the register file base pointer, and the LOD correction bits
   - all other information (such as quad address for example) travels in a separate FIFO
7. TSM0 accepts the control packet and fetches the instructions for fetch clause 0 from the global instruction store
   - TSM0 was first selected by the TSM arbiter before it could start
8. all instructions of fetch clause 0 are issued by TSM0
9. the control packet is passed to the next reservation station (the FIFO in front of ALU state machine 0, or ASM0 FIFO)
   - TSM0 does not wait for fetch requests made to the Fetch Unit to complete; it passes the register file write index for the fetch data to the TU, which will write the data to the RF as it is received
   - once the TU has written all the data for a particular clause to the register files, it increments a counter that is associated with the ASM0 FIFO; a count greater than zero indicates that the ALU state machine can go ahead and pop the FIFO and start to execute the ALU clause
10. ASM0 accepts the control packet (after being selected by the ASM arbiter) and gets the instructions for ALU clause 0 from the global instruction store
11. all instructions of ALU clause 0 are issued by ASM0, then the control packet is passed to the next reservation station (the FIFO in front of fetch state machine 1, or TSM1 FIFO)
12. the control packet continues to travel down the path of reservation stations until all clauses have been executed
    - pixel data is exported in the last ALU clause (clause 7)
      - it is sent to an output FIFO where it will be picked up by the render backend
      - the ASM arbiter will prevent a packet from starting on ASM7 if the output FIFO is full
13. after the shader program has completed, the SEQ will free up the GPRs so that they can be used by another shader program

27.1.3 *Notes*

14. The state machines and arbiters will operate ahead of time so that they will be able to immediately start the real threads or stall.
15. The register file base pointer for a vector needs to travel with the vector through the reservation stations, but the instruction store base pointer does not – this is because the RF pointer is different for all threads, but the IS pointer is only different for each state and thus can be accessed via the state pointer.

28. 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?

Parameter caches in SX?

Using both IJ buffers for center + centroid interpolation?