	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	7 May, 2001	8 September, 20153 September, 201513	GEN-CXXXXX-REVA	1 of 16

Author: Laurent Lefebvre

Issue To:

Copy No:

R400 Sequencer Specification

SEQ

Version 0.3~~2~~

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.

ATI 2009
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.

1. Overview

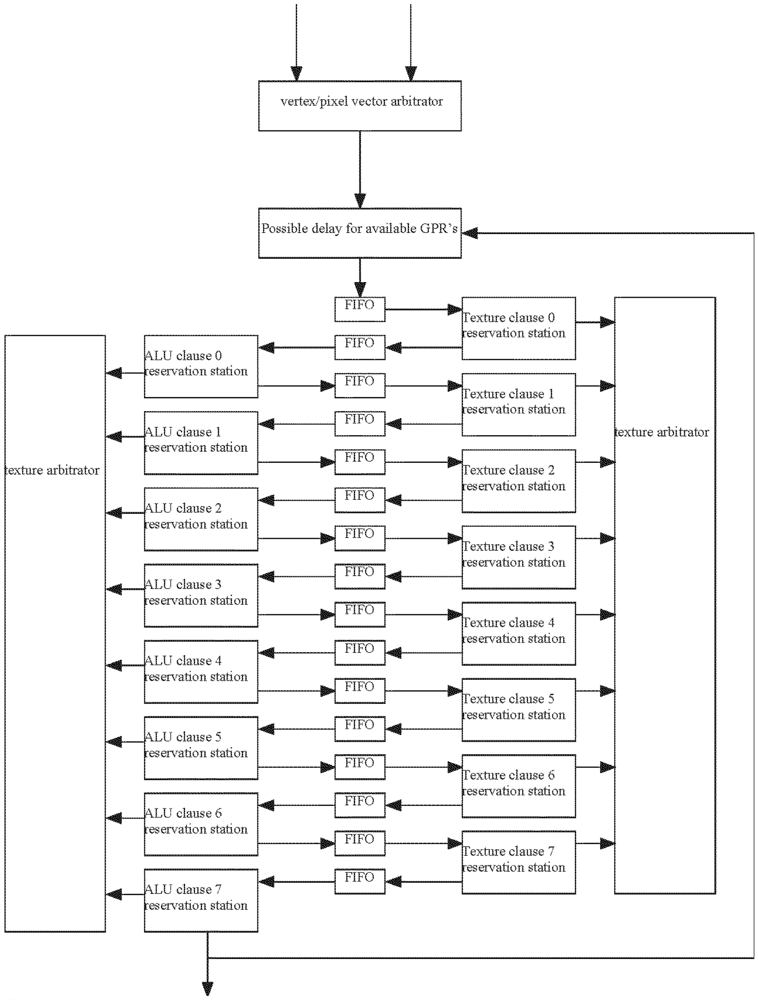
The sequencer first arbitrates between vectors of 16~~ (maybe 32)~~ vertices that arrive directly from primitive assembly and vectors of ~~8~~4 quads (16 pixels) ~~(32 pixels)~~ that are generated in the raster engine.

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

The sequencer is based on the R300 design. It chooses ~~an~~ two ALU clauses and a texture clause to execute, and executes all of the instructions in ~~a~~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 texture and eight ALU clauses, but clauses do not need to contain instructions. A vector of pixels or vertices ping-pongs along the sequencer FIFO, bouncing from texture reservation station to alu reservation station. A FIFO exists between each reservation stage, holding up vectors until the vector currently occupying a reservation station has left. A vector at a reservation station can be chosen to execute. The sequencer looks at all eight alu reservation stations to choose an alu clause to execute and all eight texture stations to choose a texture clause to execute. The arbitrator will give priority to clauses/reservation stations closer to the ~~top~~ bottom of the pipeline. It will not execute an alu clause until the texture fetches initiated by the previous texture 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.

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.

The rasterizer always checks the vertices FIFO first and if allowed by the sequencer sends the data to the shader. If the vertex FIFO is empty then, the rasterizer takes the first entry of the pixel FIFO (a vector of ~~32~~ 16 pixels) and sends it to the interpolators. Then the sequencer takes control of the packet. The packet consists of 3 bits of state, 6-7 bits for the base address of the Shader program and some information on the coverage to determine texture LOD. All other information (2x2 adresses) is put in a FIFO (one for the pixels and one for the vertices) and retrieved when the packet finishes its last clause.

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 texture machine issues a texure request and corresponding register address for the texture address (ta). A small command (tcmd) is passed to the texture system identifying the current level number (0) as well as the register ~~set being used~~write address for the texture return data. One texture request is sent every 4 clocks causing the texturing of four 2x2s worth of data (or 16 vertices). Once all the requests are sent the packet is put in FIFO 1.

Upon recept of the return data (identified by the tcmd containing the level number 0), the level 0 texture machine issues a register address for the return value (td). Then, it increments the counter of FIFO ~~one~~ 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 (~~2~~ 3 cycles later) and an instruction ~~id wich is used to index into the instruction store~~. Once the last instruction as been issued, the packet is put into FIFO 2. ~~Note that in the case of a pixel packet, the two vectors of 16 pixels are consecutive in order to hide the latency of the ALUs (8 cycles).~~

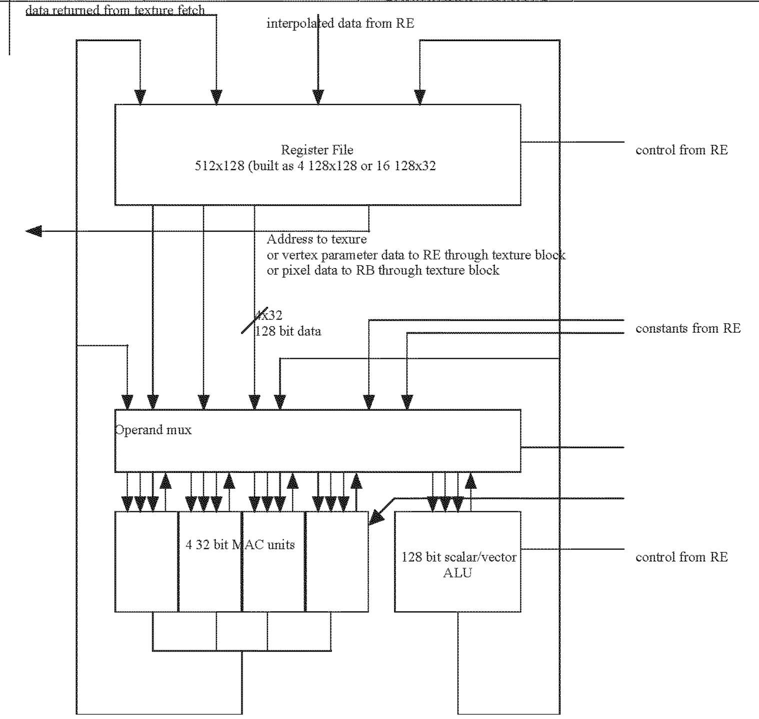
There will always be two active ALU clauses at any given time (and two arbitrers) ~~In this case, the instructions of a vector are interleaved with the instructions of the other vector~~. One arbitrer will arbitrate over the odd clock cycles and the other one will arbitrate over the even clock cycles. The only constraints between the two arbitrers is that they are not allowed to pick the same clause number as they other one is currently working on if the packet os of the same type.

If the packet is a vertex packet, upon reaching ALU clause 4, it can export the position if the position is ready. So the arbitrer must prevent ALU clause 4 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).

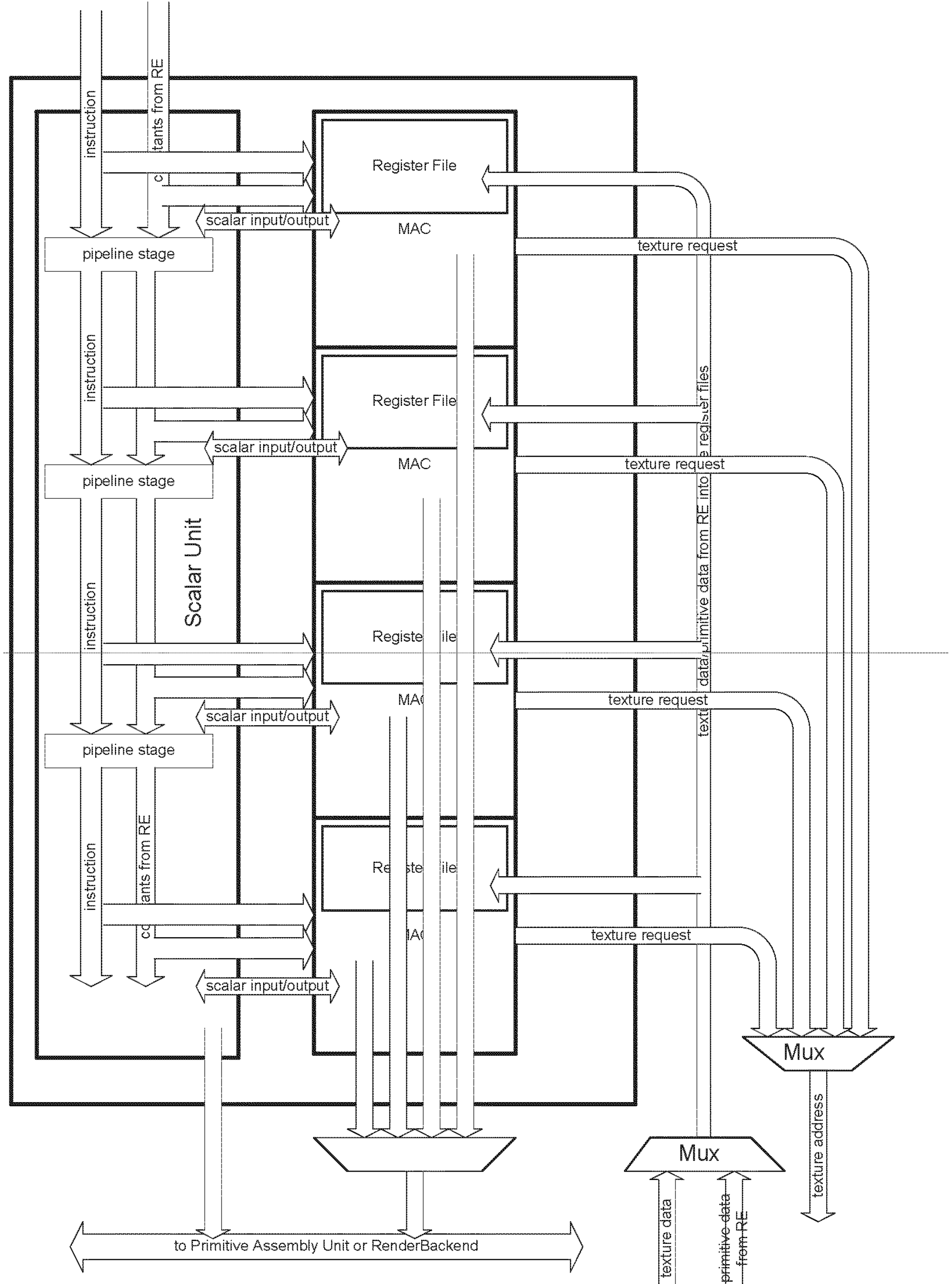
All other level process in the same way until the packet finally reaches the last ALU machine (8). On completion of the level 8 ALU clause, a valid bit is sent to the Render Backend which picks up the color data. This requires that the last instruction writes to the output register – a condition that 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~~, which picks up the data and puts it into the vertex store~~ so it can know that the data present in the parameter store is valid.

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

Each state machine maintains an address pointer specifying where the 16 ~~(or 32)~~ entries vector is located in the ~~SRAM~~register file (the texture machine has two pointers one for the read address and one for the write). Upon completion of its job, the address pointer is incremented by a predefined amount equal to the total number of registers required by the shading code. A comparison of the address pointer for the first state machine in the chain (the input state machine), and the last machine in the chain (the level 8 ALU machine), gives an indication of how much unallocated ~~SRAM~~register file memory is available

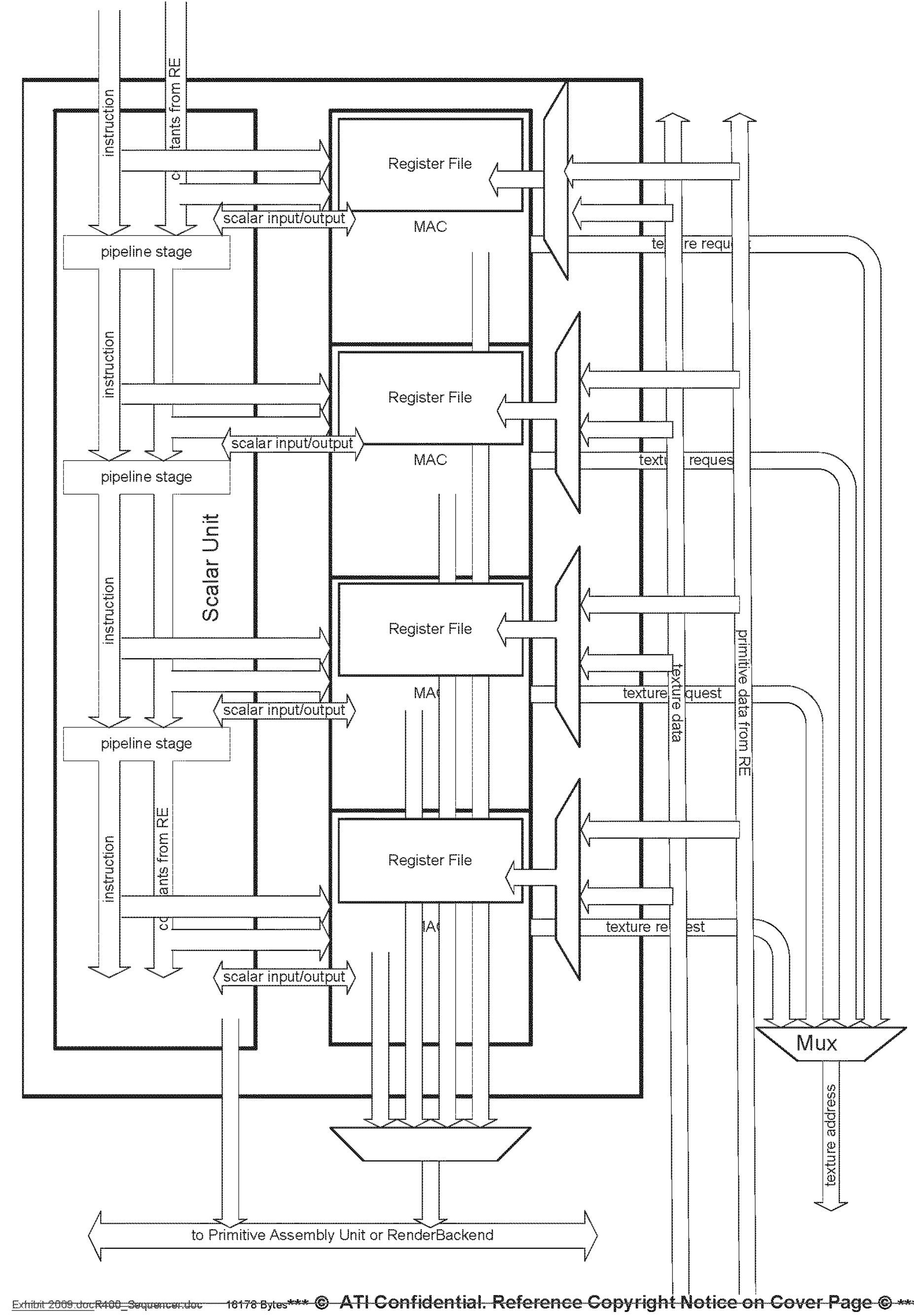


data returned from texture fetch
interpolated data from RE
Register File
512x128 (built as 4 128x128 or 16 128x32
control from RE
Address to texure
or vertex parameter data to RE through texture block
or pixel data to RB through texture block
4x32
128 bit data
constants from RE
Operand mux
4 32 bit MAC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

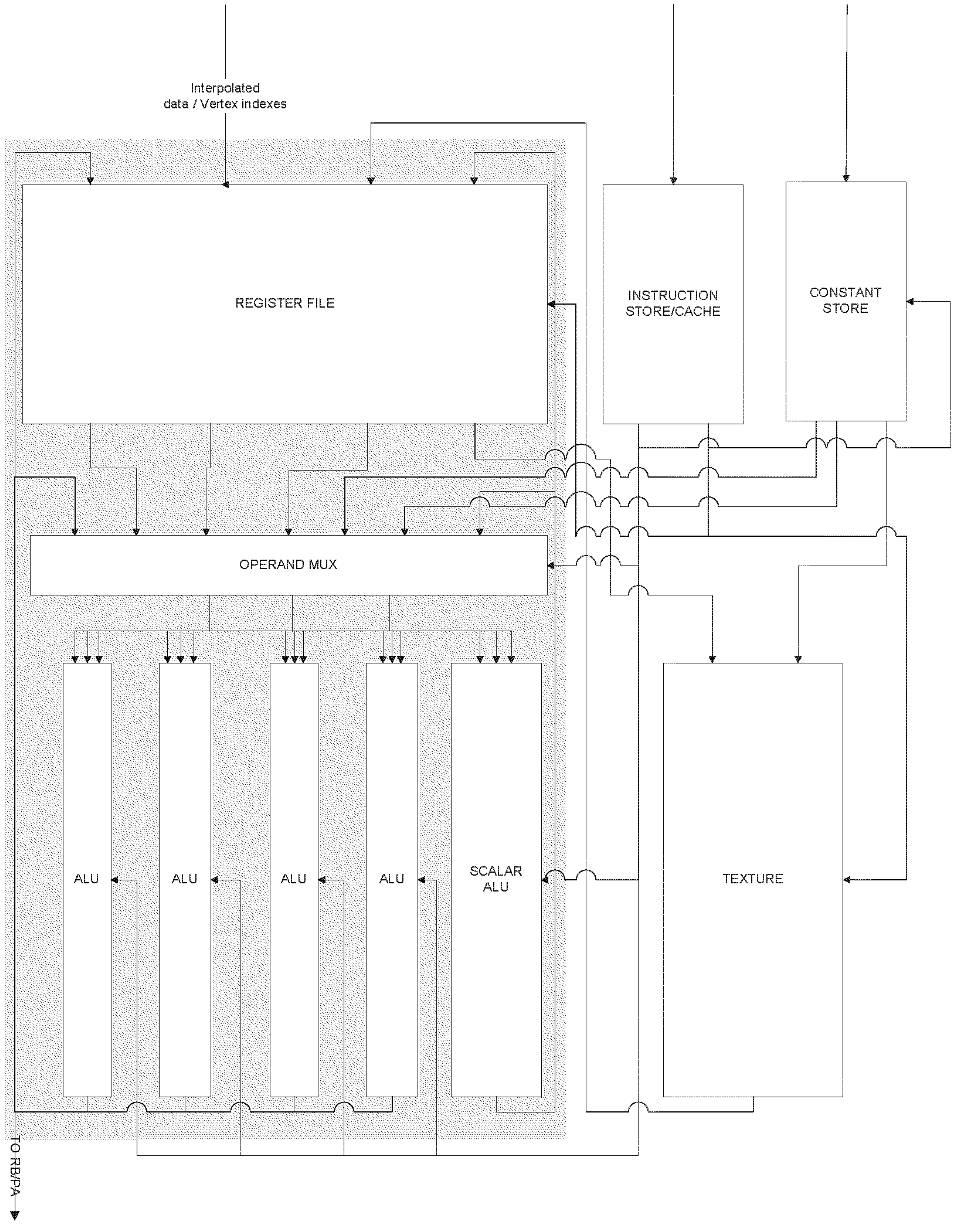


instruction
constants from RE
Register File
MAC
scalar input/output
pipeline stage
instruction
Register File
MAC
scalar input/output
pipeline stage
Scalar Unit
instruction
Register File
MAC
scalar input/output
pipeline stage
instruction
constants from RE
Register File
MAC
scalar input/output
texture request
texture request
texture request
texture request
texture data/primitive data from RE into the register files
Mux
texture address
Mux
texture data
primitive data from RE
to Primitive Assembly Unit or RenderBackend

1.2 Data Flow graph

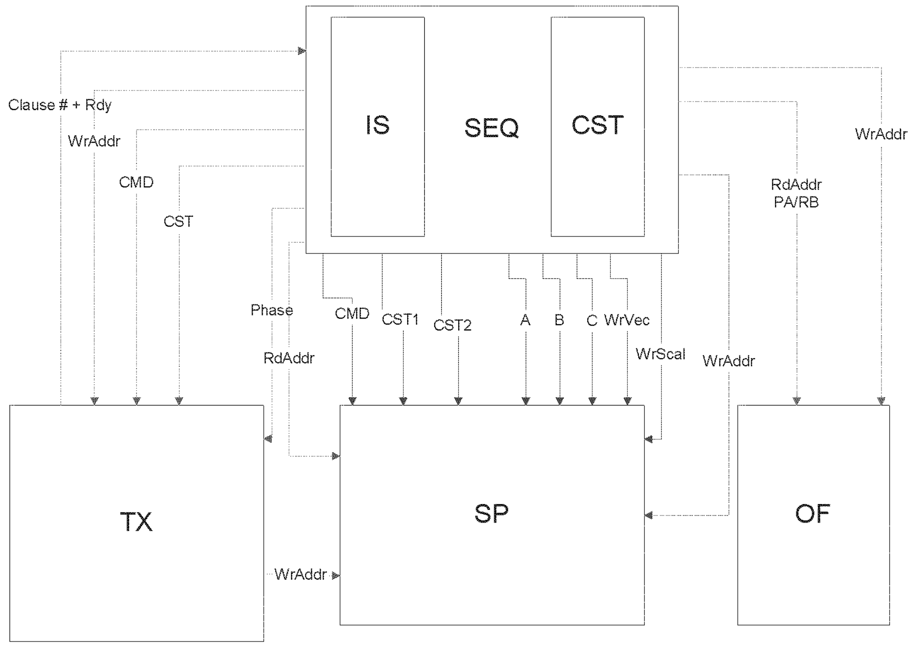


instruction
constants from RE
Register File
MAC
scalar input/output
pipeline stage
texture request
instruction
Register File
MAC
scalar input/output
pipeline stage
texture request
Scalar Unit
instruction
Register File
MAC
scalar input/output
pipeline stage
texture request
texture data
primitive data from RE
instruction
constants from RE
Register File
MAC
scalar input/output
texture request
Mux
texture address
to Primitive Assembly Unit or RenderBackend



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 Texture control interface, in red the ALU control interface, in blue the Interpolated/Vector control interface and in purple is the output file control interface.

Formatted: Bullets and Numbering

2. Interpolated data bus

Since each of the register file is actually physically divided (one 32x128 per MAC) and we don't have the place to hold a maximum size vector of vertices in the parameter buffer, we need to interpolate on a parameter basis rather than on a quad basis. So the order to the register file will be:

Q0P0 Q1P0 Q2P0 Q3P0 Q0P1 Q1P1 Q2P1 Q3P2 Q0P3 Q1P3 ...

Formatted: Bullets and Numbering

3. Instruction Store

There is going to be only one instruction store for the whole chip. It may contain up to 2000 instructions of 96 bits each. The instruction store is loaded by the sequencer using the memory hub. The read bandwith from this store is 24 bits/clock/pipe. To achieve this this instruction store is likely to be broken up into 4 blocks. An ALU instruction section (1R/1W) split in two and a texture section (1R/1W) also split in two. The bandwith out of those memories is 96 bits/clock.

4. Constant Store

Formatted: Bullets and Numbering

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 bits. The constant store is also planned to be shared. The read BW from the constant store is 512/4 bits/clock/pipe and the write bandwith is 32/4 bits/clock.

5. Looping and Branches

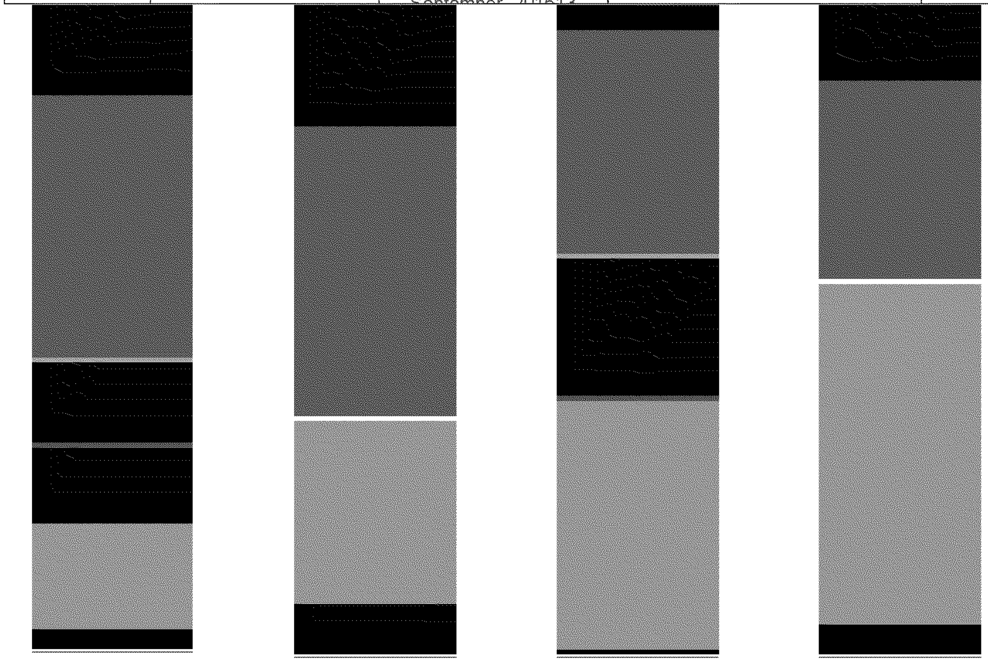
Formatted: Bullets and Numbering

Loops and branches are planned to be supported and will have to be dealt with at the sequencer level. However, it is still unclear if we plan on supporting data dependent branches or not.

6. Register file allocation

Formatted: Bullets and Numbering

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 as allowed to move again.

Formatted: Bullets and Numbering

~~2.~~7. Texture Arbitration

The texture arbitration logic chooses one of the 8 potentially pending texture 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 texture fetch per clock (or 4 fetches in one clock every 4 clocks) until all the texture fetch instructions of the clause are sent. This means that there cannot be any dependencies between two texture fetches of the same clause.

The arbitrator will not wait for the texture fetches to return prior to selecting another clause for execution. The texture pipe will be able to handle up to ~~100~~X(?) in flight texture fetches and thus there can be a fair number of active clauses waiting for their texture return data.

Formatted: Bullets and Numbering

~~3.~~8. ALU Arbitration

ALU arbitration proceeds in almost the same way than texture 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~~execute. If the packet chosen is a packet of vertices, the state machine issues one instruction every 4 clocks until the clause is finished. This means that the compiler has to insert nops between two dependent successive instructions. If the packet is a pixel packet it is made out of two sub-vectors of 16. Thus the state machine issues the first instruction for the first sub-vector and then, 4 clocks later, the first instruction of the second sub-vector and so on until the clause is finished.~~. 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):

Einst0 Oinst0 Einst1 Oinst1 Einst2 Oinst2 Einst0 Oinst3 Einst1 Oinst4 Einst2 Oinst0…
Proceeding this way hides the latency of 8 clocks of the ALUs.

Formatted: Bullets and Numbering

~~4.~~9. 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 ~~we have the ability to export at any clause~~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?). The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU arbitrer will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

Formatted: Bullets and Numbering

~~5.~~10. Content of the reservation station FIFOs

3 bits of Render State ~~and~~ 6-7 bits for the base address of the instruction store and some bits for LOD correction. Every other information (such as the coverage mask, quad address, etc.) is put in a FIFO and is retrieved when the quad exits the shader pipe to enter in the output file buffer. 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.

Formatted: Bullets and Numbering

~~6.~~11. The Output File (RB FIFO and Parameter Cache)

The output file is where program results are exported when the pixel/vertex shader finishes. It constists of a 512x128 memory cell that is statically divided between pixels and vertices. ~~Each section is a regular FIFO.~~ The output file has 1 write port and 1 read port. The sequencer is responsible for managing the addresses of this output file and for stalling the shader pipe should this output file fill up. The management is done by keeping the tail and head pointers of each sections (pixels and vertices) and incrementing them using a simple RoundRobin allocation policy. The sequencer must also arbitrate between the PA and the RB for the use of the read port. This arbitration will either be priority based or just interleaved evenly (1 read every 2 clocks for each of the blocks).

Formatted: Bullets and Numbering

~~7.~~12. Interfaces

~~7.1~~12.1 External Interfaces

~~7.1.1~~12.1.1 *Sequencer to Shader Engine Bus*

This is a bus that sends the instruction and constant data to all 4 Sub-Engines of the Shader. Because a new instruction is needed only every 4 clocks, the width of the bus is divided by 4 and both constants and instruction are sent over those 4 clocks.

Name	Direction	Bits	Description
Instruction Start	SEQ-> SP	1	High on first cycle of transfer
Constant 0	SEQ-> SP	32	128 bits transferred over 4 cycles, alpha first…blue last
Constant 1	SEQ-> SP	32	128 bits transferred over 4 cycles, alpha first…blue last
Instruction	SEQ-> SP	30	120 bits transferred over 4 cycles (order TBD) ?

Formatted: Bullets and Numbering

~~7.1.2~~12.1.2 *Shader Engine to Output File*

Every clock each Sub-Engine can output 128 bits of 'vector' data and 32 bits of 'scalar' data to an output file (?). This data will be compressed into 128 bits total prior to storage in output file.

Name	Direction	Bits	Description
UL_Vector_Out	SP-> OF	128	Vector Data out

UL_Scalar_Out	SP-> OF	32	Vector Data out
UR_Vector_Out	SP-> OF	128	Vector Data out
UR_Scalar_Out	SP-> OF	32	Vector Data out

Name	Direction	Bits	Description
LL_Vector_Out	SP-> OF	128	Vector Data out
LL_Scalar_Out	SP-> OF	32	Vector Data out
LR_Vector_Out	SP-> OF	128	Vector Data out
LR_Scalar_Out	SP-> OF	32	Vector Data out

~~7.1.3~~12.1.3 *Shader Engine to Texture Unit Bus (Fast Bus)*

Formatted: Bullets and Numbering

One quad's worth of addresses is transferred to Texture Unit every clock. These are sourced from a different pixel within each of the sub-engines repeating every 4 clocks. The ~~register file~~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.

One Quad's worth of Texture Data may be written to the ~~Register File~~Register file every clock. These are directed to a different pixel of the sub-engines repeating every 4 clocks. The ~~register file~~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_Read_Register_Index	SEQ->SP	8	Index into ~~Register File~~Register files for reading Texture Address
Tex_RegFile_Read_Data	SP->TEX	512	4 Texture Addresses read from the ~~Register File~~Register file
Tex_Write_Register_Index	SEQ->TEX	8	Index into ~~Register file~~Register file for write of returned Texture Data

~~7.1.4~~12.1.4 *Sequencer to Texture Unit bus (Slow Bus)*

Formatted: Bullets and Numbering

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

Name	Direction	Bits	Description
Tex_Ready	TEX→ SEQ	1	Data ready
Tex_Clause_Num	TEX→ SEQ	3	Clause number
Tex_cst	SEQ→TEX	?	Texture constants X bits sent over 4 clocks
Tex_Inst	SEQ→TEX	?	Texture fetch instruction X bits sent over 4 clocks

~~7.1.5~~12.1.5 *Shader Engine to RE/PA Bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Interpolator_Register_Index	SEQ->SP	8	Index into ~~Register File~~Register files for write of Interpolator/Index Data
Interpolator_Write_Mask	SEQ->SP	1	Write Mask. The same write mask is used for all 4 pixels
Interpolator_Write_Data	RE/PA->SP	512	4 interpolated vectors or vectors of indices

12.1.6 *PA to sequencer*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Adress	PA→SEQ	?	Dealocation adress sent by the PA telling the Sequencer that it is now possible to free this space in the parameter buffer. This token is a pointer in the parameter cache and 4 bits to tell the size wich is to be freed up.

Formatted

Formatted

Formatted

Formatted

Formatted: Bullets and Numbering

~~8.~~13. 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 from the texture store to feed the ALUs. Two solutions exists for this problem:

1) Let the compiler handle the case and put those instructions in a texture clause so we can use the bandwith 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. 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~~register file as well as on the ~~register file~~register file allocation method (dynamic VS static).

~~Ability to export at any clause?~~

Saving power?

~~Are we working on 32 vertices at a time or 16?~~

Size of the fifo containing the information of a vector of pixels/vertices. And size of the fifos before the reservation stations.

Sequencer Instruction memory, and constant memory.

Arbitration policy for the output file.

Loops and branches.

The parameter cache may end up in the PA rather than in the RS. Parameter cache management thus may change.

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	14 August, 2001~~14~~ ~~August, 20017 May~~	4 September, 2015~~24~~ ~~August, 20016 August~~	GEN-CXXXXX-REVA	1 of 20

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SEQ

Version 0.4~~2~~

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.

ATI 2010
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.

1. Overview

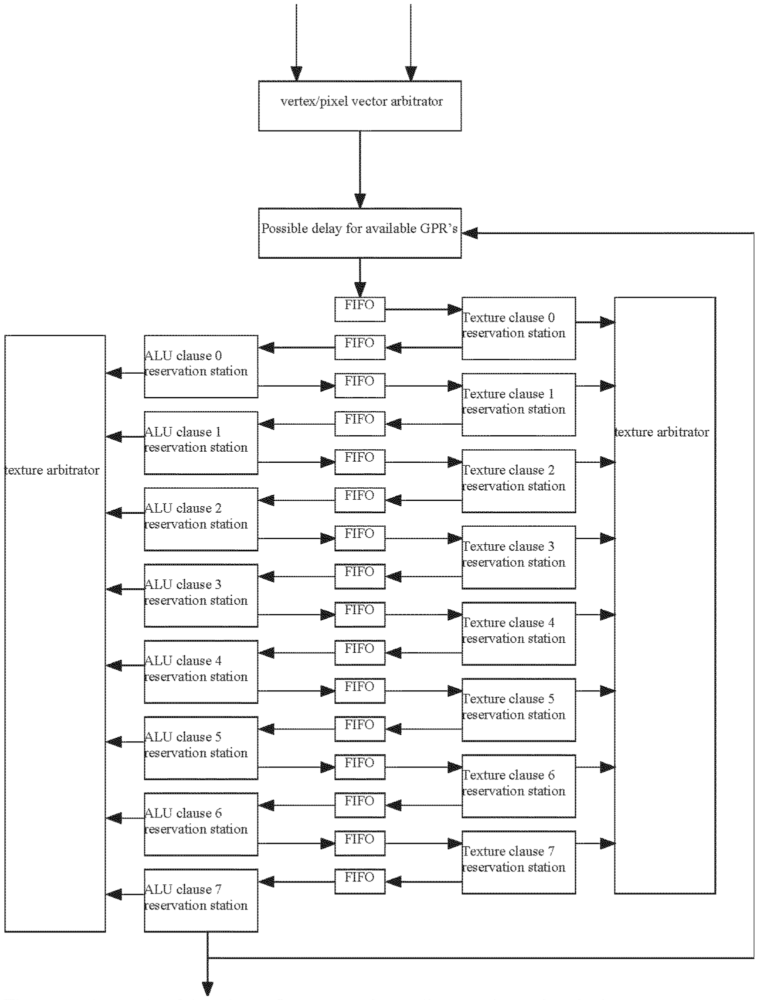
The sequencer first arbitrates between vectors of 16~~ (maybe 32)~~ vertices that arrive directly from primitive assembly and vectors of ~~8~~4 quads (16 pixels) ~~(32 pixels)~~ that are generated in the raster engine.

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

The sequencer is based on the R300 design. It chooses ~~an~~ two ALU clauses and a texture clause to execute, and executes all of the instructions in ~~a~~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 texture and eight ALU clauses, but clauses do not need to contain instructions. A vector of pixels or vertices ping-pongs along the sequencer FIFO, bouncing from texture reservation station to alu reservation station. A FIFO exists between each reservation stage, holding up vectors until the vector currently occupying a reservation station has left. A vector at a reservation station can be chosen to execute. The sequencer looks at all eight alu reservation stations to choose an alu clause to execute and all eight texture stations to choose a texture clause to execute. The arbitrator will give priority to clauses/reservation stations closer to the ~~top~~ bottom of the pipeline. It will not execute an alu clause until the texture fetches initiated by the previous texture 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.

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
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
texture arbitrator

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

The rasterizer always checks the vertices FIFO first and if allowed by the sequencer sends the data to the shader. If the vertex FIFO is empty then, the rasterizer takes the first entry of the pixel FIFO (a vector of ~~32~~ 16 pixels) and sends it to the interpolators. Then the sequencer takes control of the packet. The packet consists of 3 bits of state, 6-7 bits for the base address of the Shader program and some information on the coverage to determine texture LOD. All other information (2x2 adresses) is put in a FIFO (one for the pixels and one for the vertices) and retrieved when the packet finishes its last clause.

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 texture machine issues a texure request and corresponding register address for the texture address (ta). A small command (tcmd) is passed to the texture system identifying the current level number (0) as well as the register ~~set being used~~write address for the texture return data. One texture request is sent every 4 clocks causing the texturing of four 2x2s worth of data (or 16 vertices). Once all the requests are sent the packet is put in FIFO 1.

Upon recept of the return data ~~(identified by the tcmd containing the level number 0), the level 0 texture machine issues a register address for the return value (td)~~, the texture unit writes the data to the register file using the write address that was provided by the level 0 texture machine and sends the clause number (0) to the level 0 texture state machine to signify that the write is done and thus the data is ready. Then, the level 0 texture machine ~~it~~ increments the counter of FIFO ~~one~~ 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 (~~2~~ 3 cycles later) and an instruction ~~id wich is used to index into the instruction store~~. Once the last instruction as been issued, the packet is put into FIFO 2. ~~Note that in the case of a pixel packet, the two vectors of 16 pixels are consecutive in order to hide the latency of the ALUs (8 cycles).~~

There will always be two active ALU clauses at any given time (and two arbitrers) ~~In this case, the instructions of a vector are interleaved with the instructions of the other vector~~. One arbitrer will arbitrate over the odd clock cycles and the other one will arbitrate over the even clock cycles. The only constraints between the two arbitrers is that they are not allowed to pick the same clause number as they other one is currently working on if the packet os of the same type.

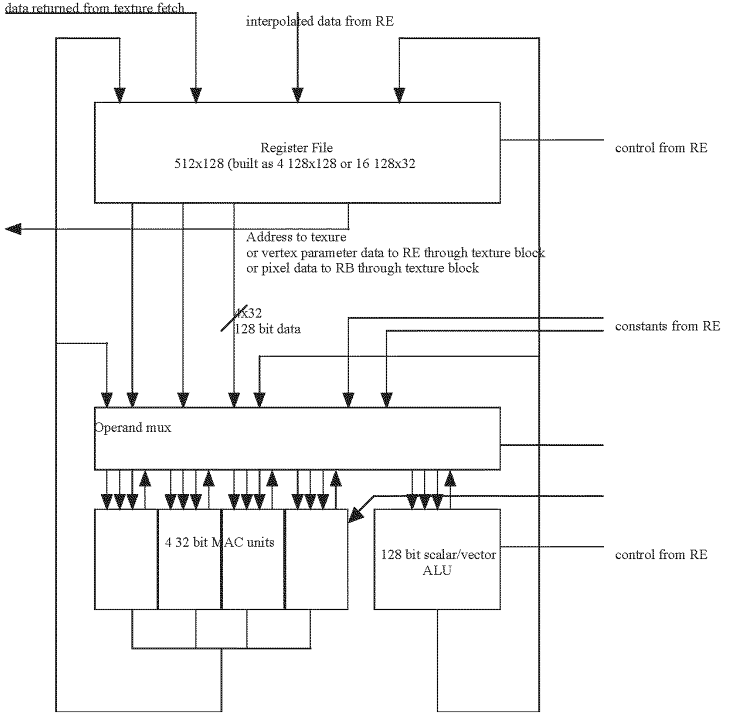
Formatted

If the packet is a vertex packet, upon reaching ALU clause 4, it can export the position if the position is ready. So the arbitrer must prevent ALU clause 4 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).

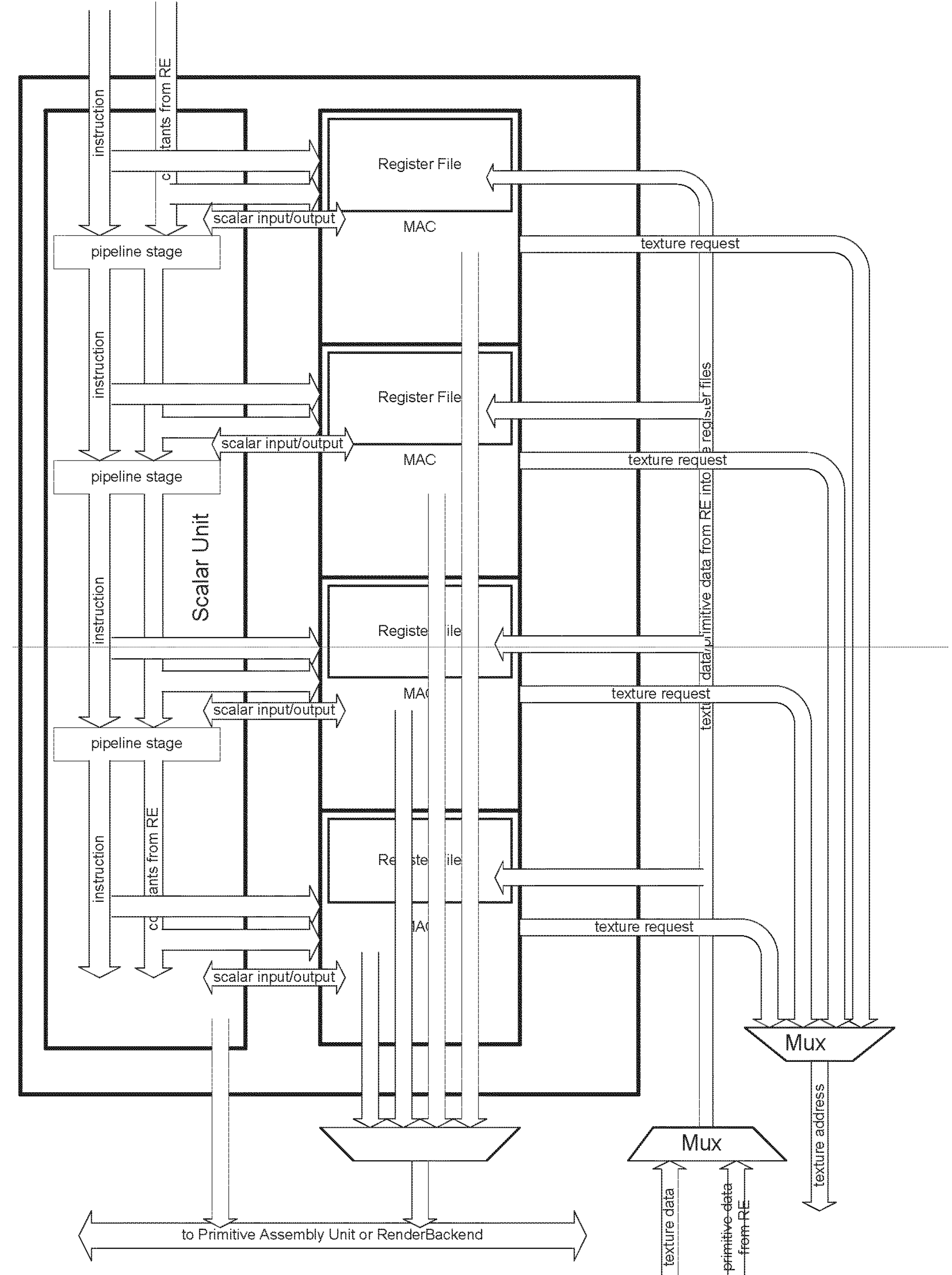
All other level process in the same way until the packet finally reaches the last ALU machine (8). On completion of the level 8 ALU clause, a valid bit is sent to the Render Backend which picks up the color data. This requires that the last instruction writes to the output register – a condition that 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~~, which picks up the data and puts it into the vertex store~~ so it can know that the data present in the parameter store is valid.

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

Each state machine maintains an address pointer specifying where the 16 ~~(or 32)~~ entries vector is located in the ~~SRAM~~register file (the texture machine has two pointers one for the read address and one for the write). Upon completion of its job, the address pointer is incremented by a predefined amount equal to the total number of registers required by the shading code. A comparison of the address pointer for the first state machine in the chain (the input state machine), and the last machine in the chain (the level 8 ALU machine), gives an indication of how much unallocated ~~SRAM~~register file memory is available

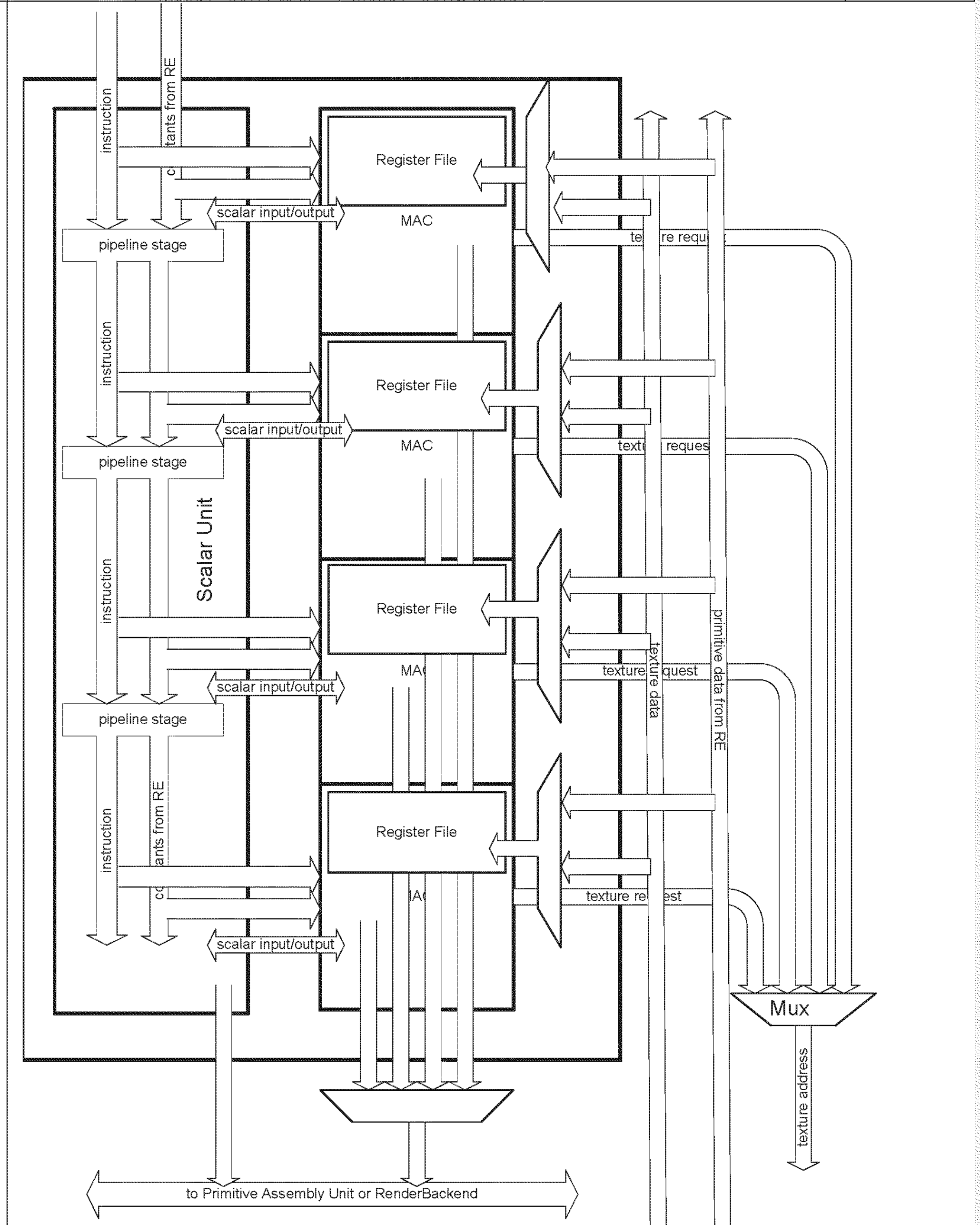


data returned from texture fetch
interpolated data from RE
Register File
512x128 (built as 4 128x128 or 16 128x32
control from RE
Address to texure
or vertex parameter data to RE through texture block
or pixel data to RB through texture block
4x32
128 bit data
constants from RE
Operand mux
4 32 bit MAC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

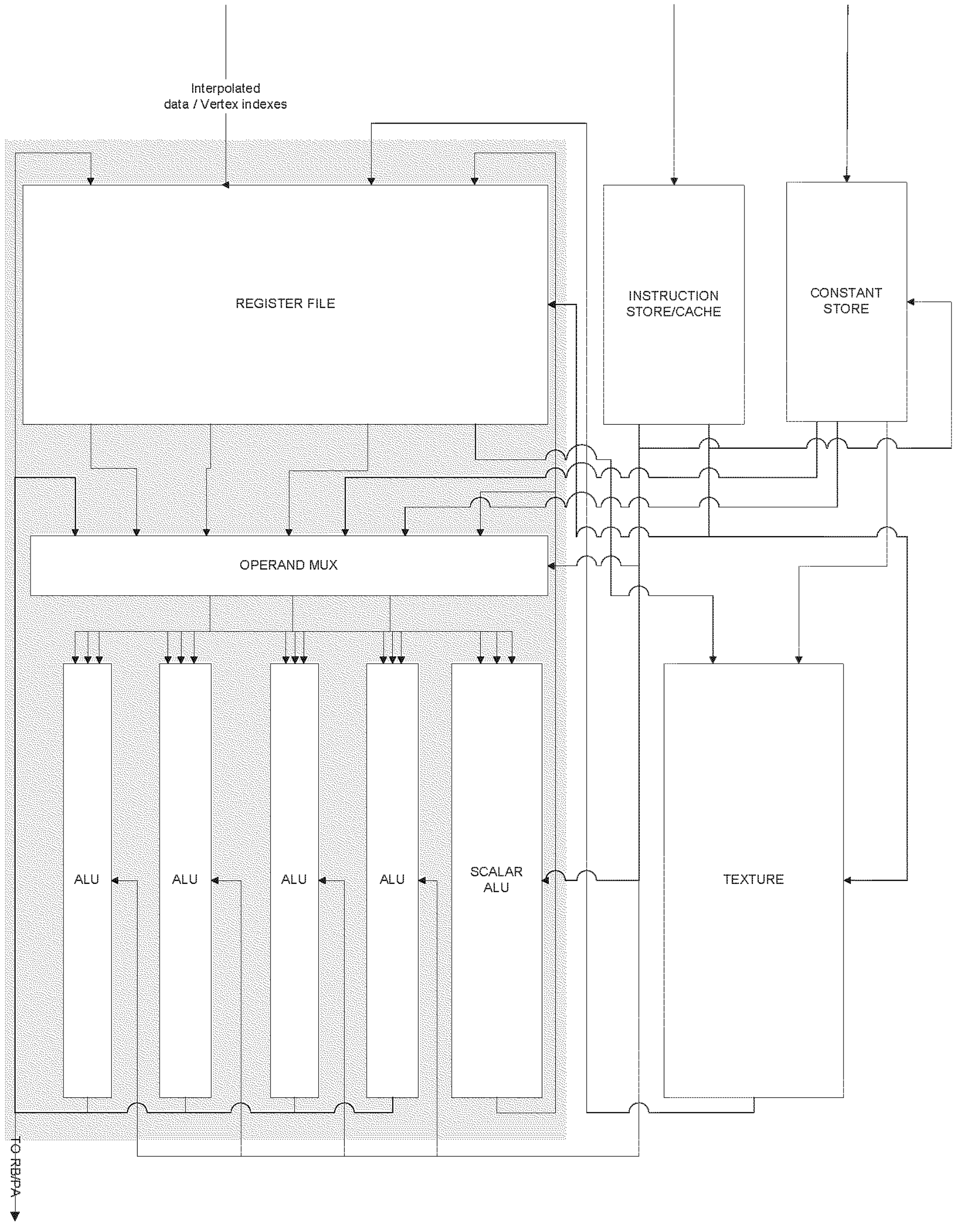


instruction
constants from RE
Register File
MAC
scalar input/output
pipeline stage
texture request
instruction
Register File
MAC
scalar input/output
pipeline stage
texture request
texture data/primitive data from RE into the register files
Scalar Unit
instruction
Register File
MAC
scalar input/output
pipeline stage
texture request
instruction
constants from RE
Register File
MAC
texture request
scalar input/output
Mux
texture address
Mux
texture data
primitive data from RE
to Primitive Assembly Unit or RenderBackend

1.2 Data Flow graph

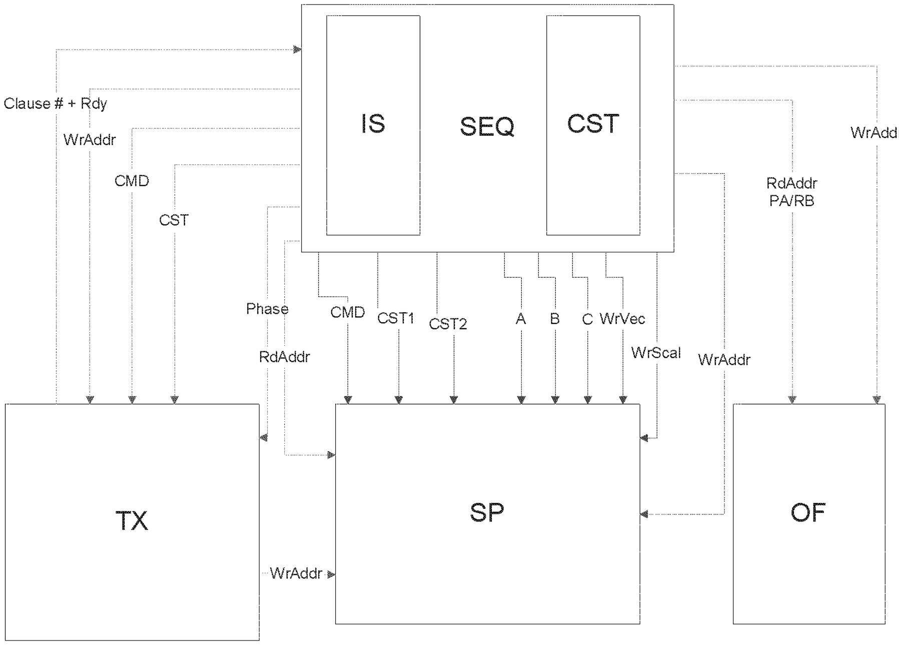


instruction
constants from RE
Register File
MAC
scalar input/output
pipeline stage
texture request
Scalar Unit
texture data
primitive data from RE
Mux
texture address
to Primitive Assembly Unit or RenderBackend



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 Texture control interface, in red the ALU control interface, in blue the Interpolated/Vector control interface and in purple is the output file control interface.

Formatted: Bullets and Numbering

2. Interpolated data bus

Since each of the register file is actually physically divided (one 128x128 per MAC) and we don't have the place to hold a maximum size vector of vertices in the parameter buffer, we need to interpolate on a parameter basis rather than on a quad basis. So the order to the register file will be:

Q0P0 Q1P0 Q2P0 Q3P0 Q0P1 Q1P1 Q2P1 Q3P1 Q0P2 Q1P2 ...

Formatted: Bullets and Numbering

3. Instruction Store

There is going to be only one instruction store for the whole chip. It may contain up to 2000 instructions of 96 bits each.

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

The read bandwith from this store is 24 bits/clock/pipe. To achieve this this instruction store is likely to be broken up into 4 blocks. An ALU instruction section (1R/1W) split in two and a texture section (1R/1W) also split in two. The bandwith out of those memories is 96 bits/clock.

4. Constant Store

Formatted: Bullets and Numbering

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 bits. The constant store is also planned to be shared. The read BW from the constant store is 512/4 bits/clock/pipe and the write bandwith is 32/4 bits/clock.

5. Looping and Branches

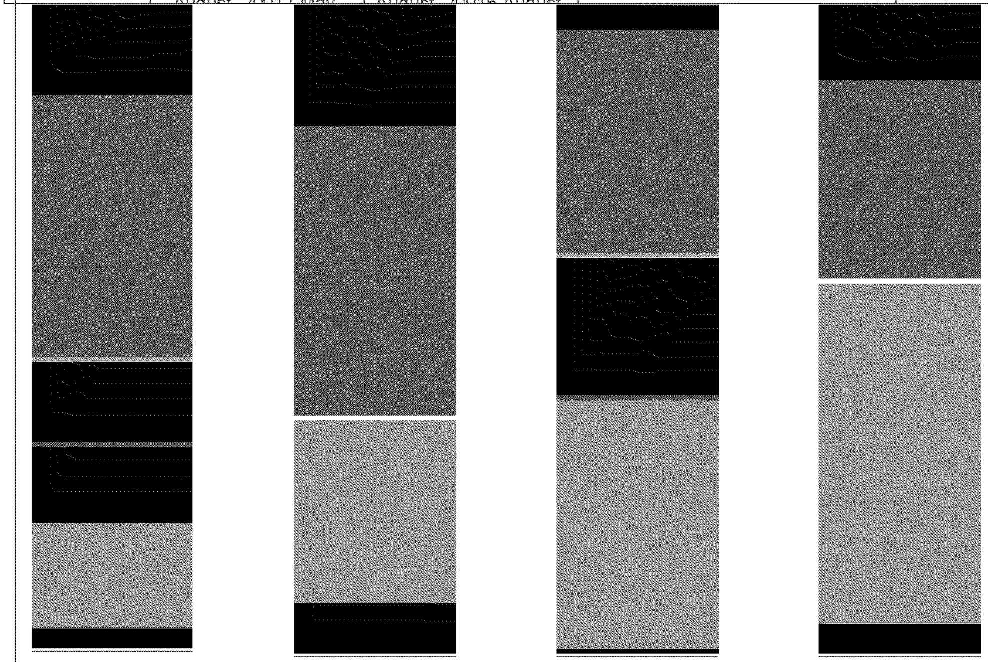
Formatted: Bullets and Numbering

Loops and branches are planned to be supported and will have to be dealt with at the sequencer level. However, it is still unclear if we plan on supporting data dependent branches or not.

6. Register file allocation

Formatted: Bullets and Numbering

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 as allowed moving again.

Formatted: Bullets and Numbering

~~2.~~7. Texture Arbitration

The texture arbitration logic chooses one of the 8 potentially pending texture 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 texture fetch per clock (or 4 fetches in one clock every 4 clocks) until all the texture fetch instructions of the clause are sent. This means that there cannot be any dependencies between two texture fetches of the same clause.

The arbitrator will not wait for the texture fetches to return prior to selecting another clause for execution. The texture pipe will be able to handle up to ~~100~~X(?) in flight texture fetches and thus there can be a fair number of active clauses waiting for their texture return data.

Formatted: Bullets and Numbering

~~3.~~8. ALU Arbitration

ALU arbitration proceeds in almost the same way than texture 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~~execute. If the packet chosen is a packet of vertices, the state machine issues one instruction every 4 clocks until the clause is finished. This means that the compiler has to insert nops between two dependent successive instructions. If the packet is a pixel packet it is made out of two sub-vectors of 16. Thus the state machine issues the first instruction for the first sub-vector and then, 4 clocks later, the first instruction of the second sub-vector and so on until the clause is finished.~~. 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):

Einst0 Oinst0 Einst1 Oinst1 Einst2 Oinst2 Einst0 Oinst3 Einst1 Oinst4 Einst2 Oinst0...
Proceeding this way hides the latency of 8 clocks of the ALUs.

Formatted: Bullets and Numbering

~~4.~~9. 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 ~~we have the ability to export at any clause~~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?). The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU arbitrer will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

Formatted: Bullets and Numbering

~~5.~~10. Content of the reservation station FIFOs

3 bits of Render State ~~and~~ 6-7 bits for the base address of the instruction store and some bits for LOD correction. Every other information (such as the coverage mask, quad address, etc.) is put in a FIFO and is retrieved when the quad exits the shader pipe to enter in the output file buffer. 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.

Formatted: Bullets and Numbering

~~6.~~11. The Output File (RB FIFO and Parameter Cache)

The output file is where program results are exported when the pixel/vertex shader finishes. It constists of a 512x128 memory cell that is statically divided between pixels and vertices. ~~Each section is a regular FIFO.~~ The output file has 1 write port and 1 read port. The sequencer is responsible for managing the addresses of this output file and for stalling the shader pipe should this output file fill up. The management is done by keeping the tail and head pointers of each sections (pixels and vertices) and incrementing them using a simple RoundRobin allocation policy. The sequencer must also arbitrate between the PA and the RB for the use of the read port. This arbitration will either be priority based or just interleaved evenly (1 read every 2 clocks for each of the blocks).

Formatted: Bullets and Numbering

~~7.~~12. Interfaces

~~7.1~~12.1 External Interfaces

~~7.1.1~~12.1.1 *Sequencer to Shader Engine Bus*

This is a bus that sends the instruction and constant data to all 4 Sub-Engines of the Shader. Because a new instruction is needed only every 4 clocks, the width of the bus is divided by 4 and both constants and instruction are sent over those 4 clocks.

Name	Direction	Bits	Description
Instruction Start	SEQ-> SP	1	High on first cycle of transfer
Constant 0	SEQ-> SP	32	128 bits transferred over 4 cycles, alpha first...blue last
Constant 1	SEQ-> SP	32	128 bits transferred over 4 cycles, alpha first...blue last
Instruction	SEQ-> SP	30	120 bits transferred over 4 cycles (order TBD) ?

Formatted: Bullets and Numbering

~~7.1.2~~12.1.2 *Shader Engine to Output File*

Every clock each Sub-Engine can output 128 bits of 'vector' data and 32 bits of 'scalar' data to an output file (?). This data will be compressed into 128 bits total prior to storage in output file.

Name	Direction	Bits	Description
UL_Vector_Out	SP-> OF	128	Vector Data out

UL_Scalar_Out	SP-> OF	32	Vector Data out
UR_Vector_Out	SP-> OF	128	Vector Data out
UR_Scalar_Out	SP-> OF	32	Vector Data out

Name	Direction	Bits	Description
LL_Vector_Out	SP-> OF	128	Vector Data out
LL_Scalar_Out	SP-> OF	32	Vector Data out
LR_Vector_Out	SP-> OF	128	Vector Data out
LR_Scalar_Out	SP-> OF	32	Vector Data out

~~7.1.3~~12.1.3 Shader Engine to Texture Unit Bus (Fast Bus)

Formatted: Bullets and Numbering

One quad's worth of addresses is transferred to Texture Unit every clock. These are sourced from a different pixel within each of the sub-engines repeating every 4 clocks. The ~~register file~~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.

One Quad's worth of Texture Data may be written to the ~~Register File~~Register file every clock. These are directed to a different pixel of the sub-engines repeating every 4 clocks. The ~~register file~~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_Read_Register_Index	SEQ->SP	8	Index into ~~Register File~~Register files for reading Texture Address
Tex_RegFile_Read_Data	SP->TEX	512	4 Texture Addresses read from the ~~Register File~~Register file
Tex_Write_Register_Index	SEQ->TEX	8	Index into ~~Register file~~Register file for write of returned Texture Data

~~7.1.4~~12.1.4 Sequencer to Texture Unit bus (Slow Bus)

Formatted: Bullets and Numbering

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

Name	Direction	Bits	Description
Tex_Ready	TEX→ SEQ	1	Data ready
Tex_Clause_Num	TEX→ SEQ	3	Clause number
Tex_cst	SEQ→TEX	?	Texture constants X bits sent over 4 clocks
Tex_Inst	SEQ→TEX	?	Texture fetch instruction X bits sent over 4 clocks

~~7.1.5~~12.1.5 Shader Engine to RE/PA Bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Interpolator_Register_Index	SEQ->SP	8	Index into ~~Register File~~Register files for write of Interpolator/Index Data
Interpolator_Write_Mask	SEQ->SP	1	Write Mask. The same write mask is used for all 4 pixels
Interpolator_Write_Data	RE/PA->SP	512	4 interpolated vectors or vectors of indices

12.1.6 PA? to sequencer

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Adress	PA→SEQ	?	Dealocation adress sent by the PA telling the Sequencer that it is now possible to free this space in the parameter buffer. This token is a pointer in the parameter cache and 4 bits to tell the size wich is to be freed up.

Formatted

Formatted

Formatted

Formatted

Formatted: Bullets and Numbering

13. Examples of program executions

13.1.1 *Sequencer Control of a Vector of Vertices*

1. PA sends a vector of 16 vertices (actually vertex indices – 32 bits/index for 512 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 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 512 bits/cycle)
   - the 16 vertex indices are sent to the 16 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 texture 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 texture clause 0 from the global instruction store
   - TSM0 was first selected by the TSM arbiter before it could start
7. all instructions of texture 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 Texture Unit to complete; it passes the register file write index for the texture 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 texture 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

13.1.2 *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 Parameter Buffer is loaded from the Parameter Cache before the SEQ takes control of the vector
    - after the HZ culling stage a request is made by the RE to send parameter data to the Parameter buffer
    - the Parameter buffer is wide enough to source 3 vertices worth of a particular parameter in one cycle
    - **at this moment the right sequencer will free up the parameter store locations not used anymore using the token provided by the PA.**
3. 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 one quad's worth of barycentrics per cycle
4. 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
5. 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
6. SEQ control starts with the interpolation of parameters (up to 16 per thread) by sending the barycentric coordinates from the Pixel FIFO and the parameters from the Parameter Buffer to the interpolator
    - P0i, P0j, and P0k (the value of P0 at each vertex) are loaded into the interpolator from the Parameter buffer
    - Q0 i, j, and k are loaded into the interpolator from the Pixel FIFO
    - The interpolator then generates the parameter value for each pixel in Q0 (Q0P0)
    - **P0i, P0j, and P0k are sent to the interpolator for Q1 only if Q1 is from a different primitive; if Q1 is from the same primitive as Q0, then the P0i, P0j, and P0k values loaded for Q0 are held by the interpolator and reused for Q1**
        - **a "different_prim" control bit is passed with the barycentric data for each quad in the Pixel FIFO that indicates whether new parameter data needs to be loaded into the interpolator**
    - Q1 i, j, and k are then loaded into the interpolator from the Pixel FIFO
    - The interpolator then generates the parameter value for each pixel in Q1 (Q1P0)
    - Q2P0 and Q3P0 are generated in a similar manner
    - The next set of parameter data - P1i, P1j, and P1k - is then loaded into the interpolator
    - Q0 i, j, and k now must be re-read from the Pixel FIFO – this means that the output of the Pixel FIFO loops through the top four entries on each read command until at the end a final "block_pop" signal is asserted, causing the top four sets of barycentric coordinates to finally be removed
    - so the order of parameter info generated is Q0P0, Q1P0, Q2P0, Q3P0, Q0P1, Q1P1, etc.
7. SEQ controls the transfer of interpolated data to the SP register file over the RE_SP interface (which has a bandwidth of 512 bits/cycle)
    - 16 pixels worth of interpolated parameter data is sent to the 16 register files over 4 cycles
        - RF0 of SU0, SU1, SU2, and SU3 is written with Q0P0 the first cycle
        - RF1 of SU0, SU1, SU2, and SU3 is written with Q1P0 second cycle
        - RF2 of SU0, SU1, SU2, and SU3 is written with Q2P0 third cycle
        - RF3 of SU0, SU1, SU2, and SU3 is written with Q3P0 fourth cycle

8. SEQ constructs a control packet for the vector and sends it to the first reservation station (the FIFO in front of texture 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 (such as quad address for example) travels in a separate FIFO
9. TSM0 accepts the control packet and fetches the instructions for texture clause 0 from the global instruction store
   - TSM0 was first selected by the TSM arbiter before it could start
10. all instructions of texture clause 0 are issued by TSM0
11. 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 texture requests made to the Texture Unit to complete; it passes the register file write index for the texture 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
12. 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
13. 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 texture state machine 1, or TSM1 FIFO)
14. 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
15. after the shader program has completed, the SEQ will free up the GPRs so that they can be used by another shader program

13.1.3 *Notes*

Formatted: Bullets and Numbering

16. the state machines and arbitrers will operate ahead of time so that they will be able to immediately start the real threads or stall.
17. 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
18. Waterfalling, parameter buffer allocation, loops and branches and parameter cache de-allocation still needs to be specked out.

~~8.~~14. 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 from the texture store to feed the ALUs. Two solutions exists for this problem:

1) Let the compiler handle the case and put those instructions in a texture clause so we can use the bandwith 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. 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~~register file as well as on the ~~register file~~register file allocation method (dynamic VS static).

~~Ability to export at any clause?~~

Saving power?

~~Are we working on 32 vertices at a time or 16?~~

Size of the fifo containing the information of a vector of pixels/vertices. And size of the fifos before the reservation stations.

Sequencer Instruction memory, and constant memory.

Arbitration policy for the output file.

Loops and branches.

The parameter cache may end up in the PA rather than in the RS. Parameter cache management thus may change.

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	14 August, 2001~~14 August, 20017 May~~	4 September, 2015~~7 September, 20016~~	GEN-CXXXXX-REVA	1 of 26

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SEQ

Version 0.5~~2~~

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.

ATI 2011
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 0.4 (Laurent Lefebvre) Date : September 7, 2001	Added timing diagrams (Vic)

1. Overview

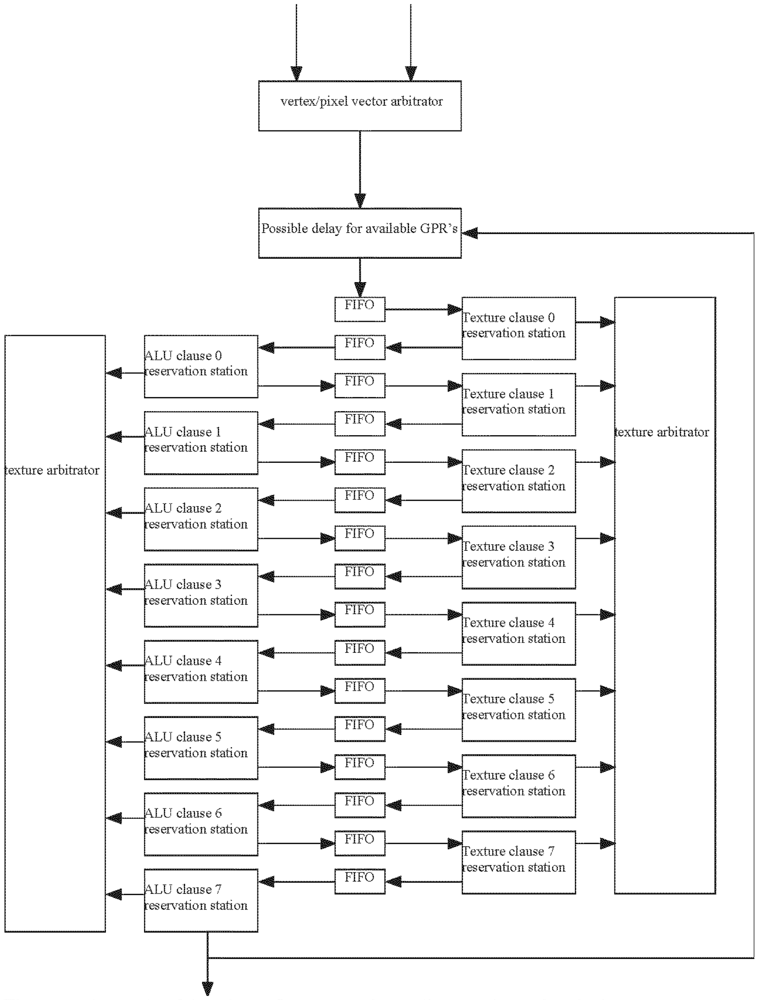
The sequencer first arbitrates between vectors of 16 ~~(maybe 32)~~ vertices that arrive directly from primitive assembly and vectors of ~~8~~4 quads (16 pixels) ~~(32 pixels)~~ that are generated in the raster engine.

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

The sequencer is based on the R300 design. It chooses ~~an~~ two ALU clauses and a texture clause to execute, and executes all of the instructions in ~~a~~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 texture and eight ALU clauses, but clauses do not need to contain instructions. A vector of pixels or vertices ping-pongs along the sequencer FIFO, bouncing from texture reservation station to alu reservation station. A FIFO exists between each reservation stage, holding up vectors until the vector currently occupying a reservation station has left. A vector at a reservation station can be chosen to execute. The sequencer looks at all eight alu reservation stations to choose an alu clause to execute and all eight texture stations to choose a texture clause to execute. The arbitrator will give priority to clauses/reservation stations closer to the ~~top~~ bottom of the pipeline. It will not execute an alu clause until the texture fetches initiated by the previous texture 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.

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.

The rasterizer always checks the vertices FIFO first and if allowed by the sequencer sends the data to the shader. If the vertex FIFO is empty then, the rasterizer takes the first entry of the pixel FIFO (a vector of ~~32~~ 16 pixels) and sends it to the interpolators. Then the sequencer takes control of the packet. The packet consists of 3 bits of state, 6-7 bits for the base address of the Shader program and some information on the coverage to determine texture LOD. All other information (2x2 adresses) is put in a FIFO (one for the pixels and one for the vertices) and retrieved when the packet finishes its last clause.

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 texture machine issues a texure request and corresponding register address for the texture address (ta). A small command (tcmd) is passed to the texture system identifying the current level number (0) as well as the register ~~set being used~~write address for the texture return data. One texture request is sent every 4 clocks causing the texturing of four 2x2s worth of data (or 16 vertices). Once all the requests are sent the packet is put in FIFO 1.

Upon recept of the return data~~ (identified by the tcmd containing the level number 0), the level 0 texture machine issues a register address for the return value (td)~~, the texture unit writes the data to the register file using the write address that was provided by the level 0 texture machine and sends the clause number (0) to the level 0 texture state machine to signify that the write is done and thus the data is ready. Then, the level 0 texture machine~~ it~~ increments the counter of FIFO ~~one~~ 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 (~~2~~ 3 cycles later) and an instruction~~ id wich is used to index into the instruction store~~. Once the last instruction as been issued, the packet is put into FIFO 2.~~ Note that in the case of a pixel packet, the two vectors of 16 pixels are consecutive in order to hide the latency of the ALUs (8 cycles).~~

There will always be two active ALU clauses at any given time (and two arbitrers)~~ In this case, the instructions of a vector are interleaved with the instructions of the other vector~~. One arbitrer will arbitrate over the odd clock cycles and the other one will arbitrate over the even clock cycles. The only constraints between the two arbitrers is that they are not allowed to pick the same clause number as they other one is currently working on if the packet os of the same type.

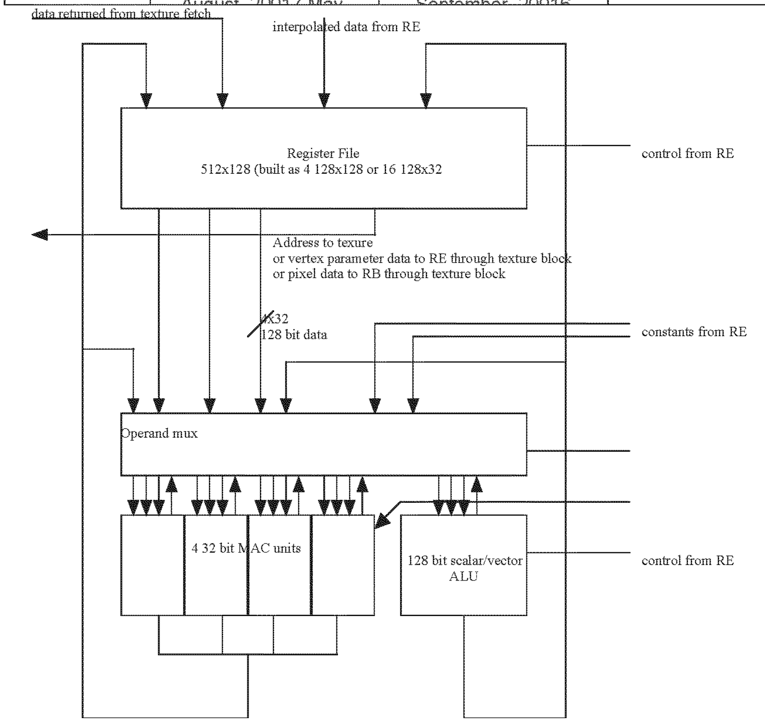
Formatted

If the packet is a vertex packet, upon reaching ALU clause 4, it can export the position if the position is ready. So the arbitrer must prevent ALU clause 4 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).

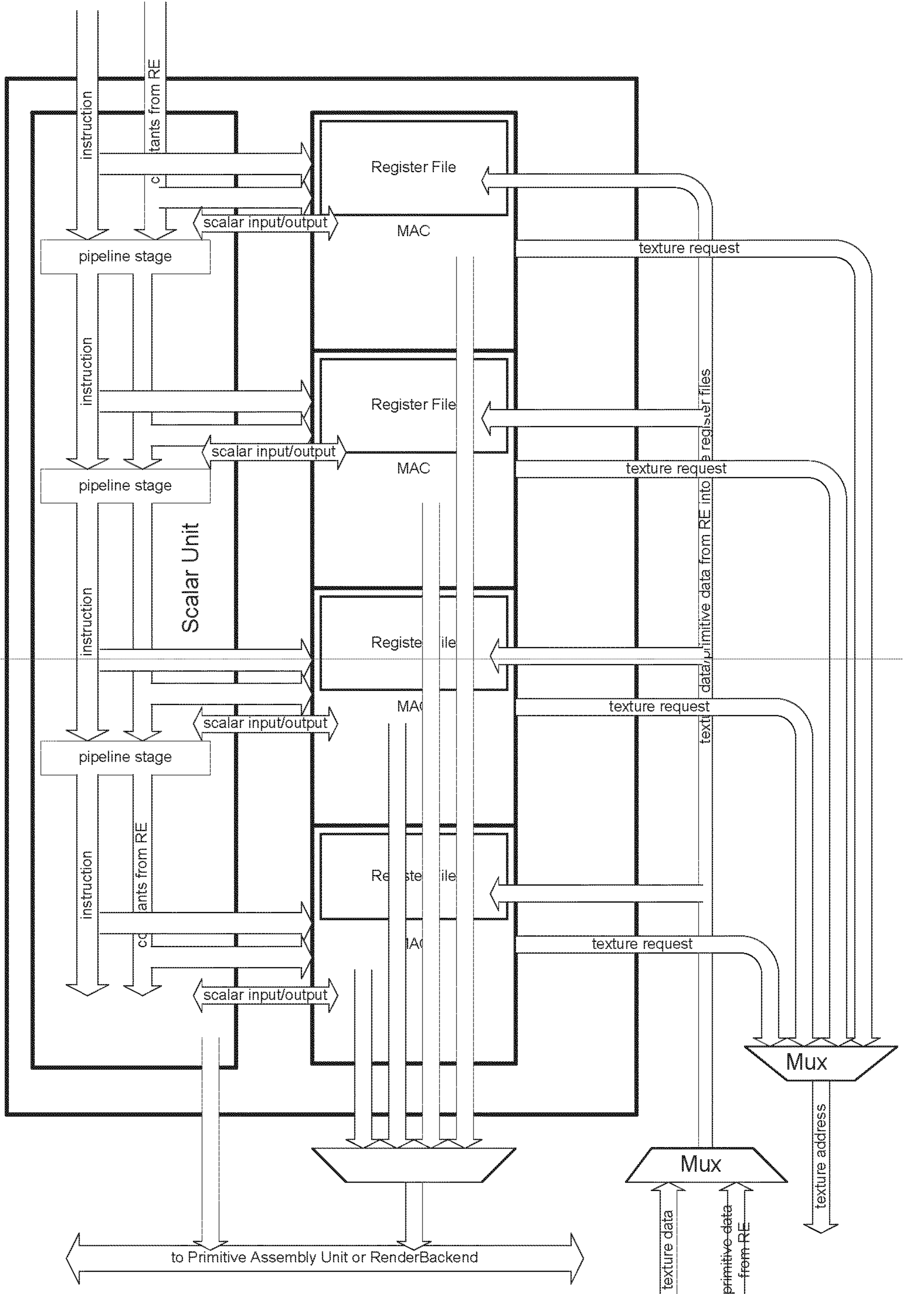
All other level process in the same way until the packet finally reaches the last ALU machine (8). On completion of the level 8 ALU clause, a valid bit is sent to the Render Backend which picks up the color data. This requires that the last instruction writes to the output register – a condition that 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~~, which picks up the data and puts it into the vertex store~~ so it can know that the data present in the parameter store is valid.

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

Each state machine maintains an address pointer specifying where the 16~~ (or 32)~~ entries vector is located in the ~~SRAM~~register file (the texture machine has two pointers one for the read address and one for the write). Upon completion of its job, the address pointer is incremented by a predefined amount equal to the total number of registers required by the shading code. A comparison of the address pointer for the first state machine in the chain (the input state machine), and the last machine in the chain (the level 8 ALU machine), gives an indication of how much unallocated ~~SRAM~~register file memory is available

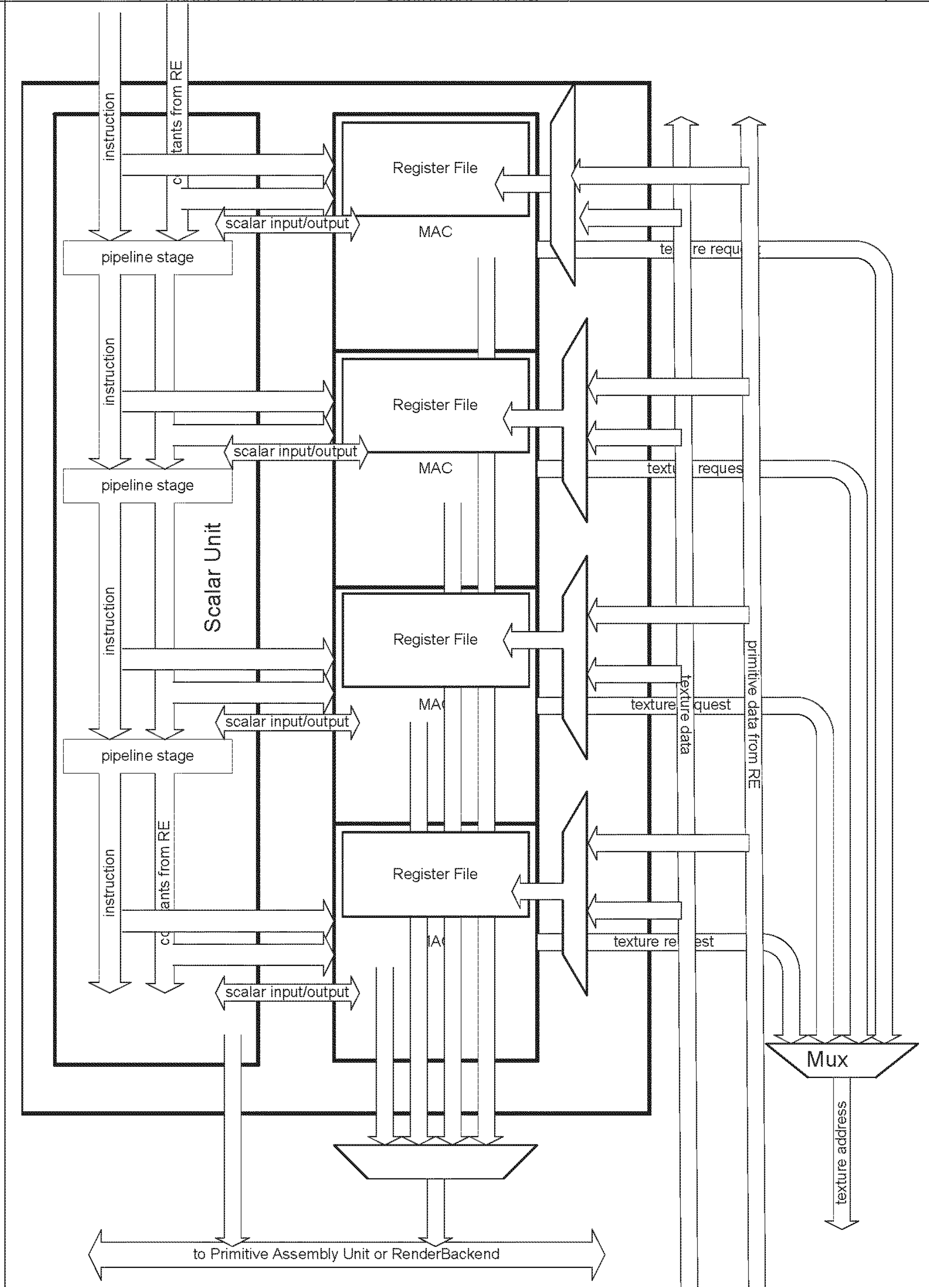


data returned from texture fetch
interpolated data from RE
Register File
512x128 (built as 4 128x128 or 16 128x32
control from RE
Address to texure
or vertex parameter data to RE through texture block
or pixel data to RB through texture block
4x32
128 bit data
constants from RE
Operand mux
4 32 bit MAC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

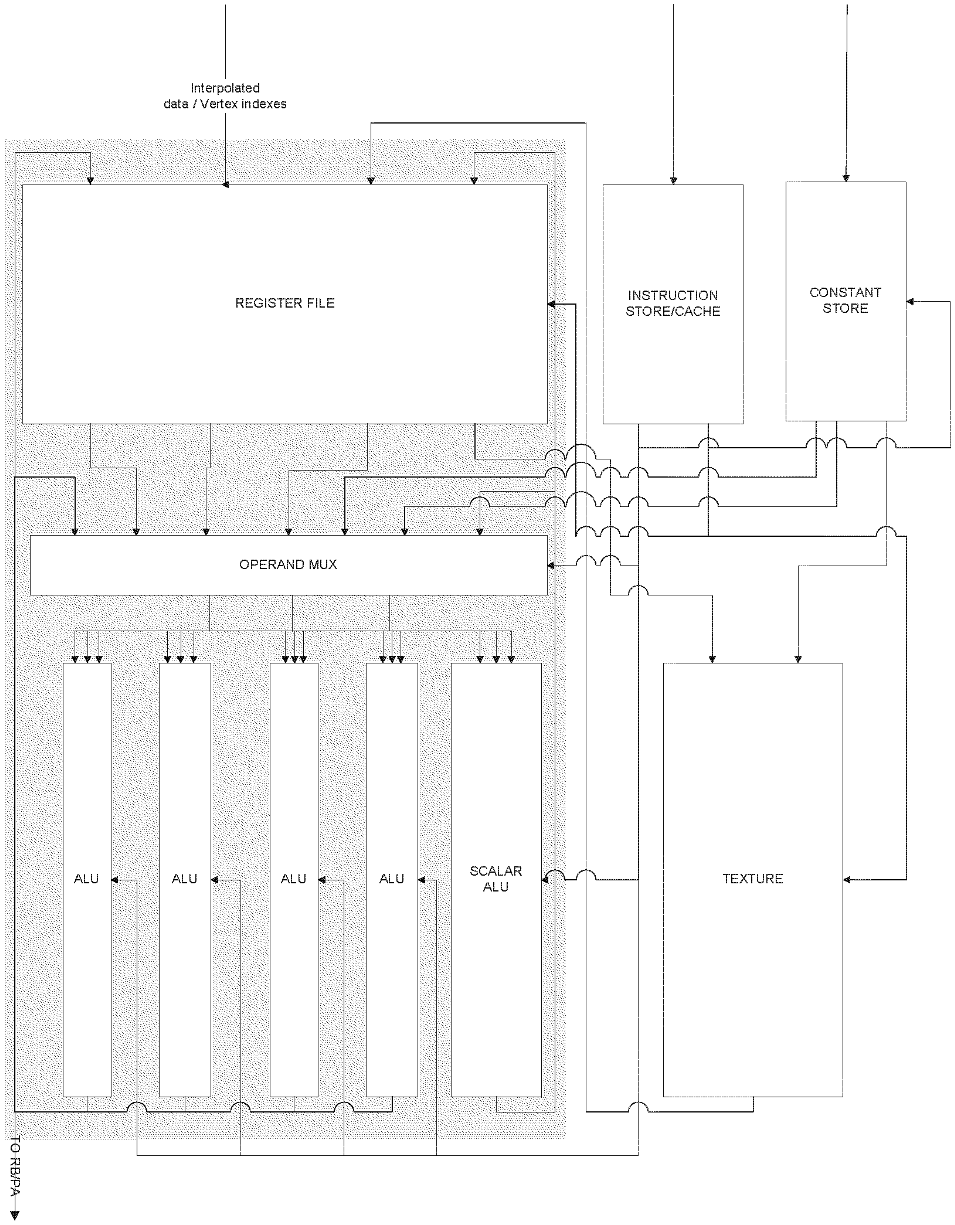


instruction
constants from RE
Register File
scalar input/output
MAC
texture request
pipeline stage
instruction
Register File
scalar input/output
MAC
texture request
pipeline stage
Scalar Unit
instruction
Register File
scalar input/output
MAC
texture request
texture data/primitive data from RE into the register files
pipeline stage
instruction
constants from RE
Register File
MAC
texture request
scalar input/output
Mux
texture address
Mux
texture data
primitive data from RE
to Primitive Assembly Unit or RenderBackend

1.2 Data Flow graph

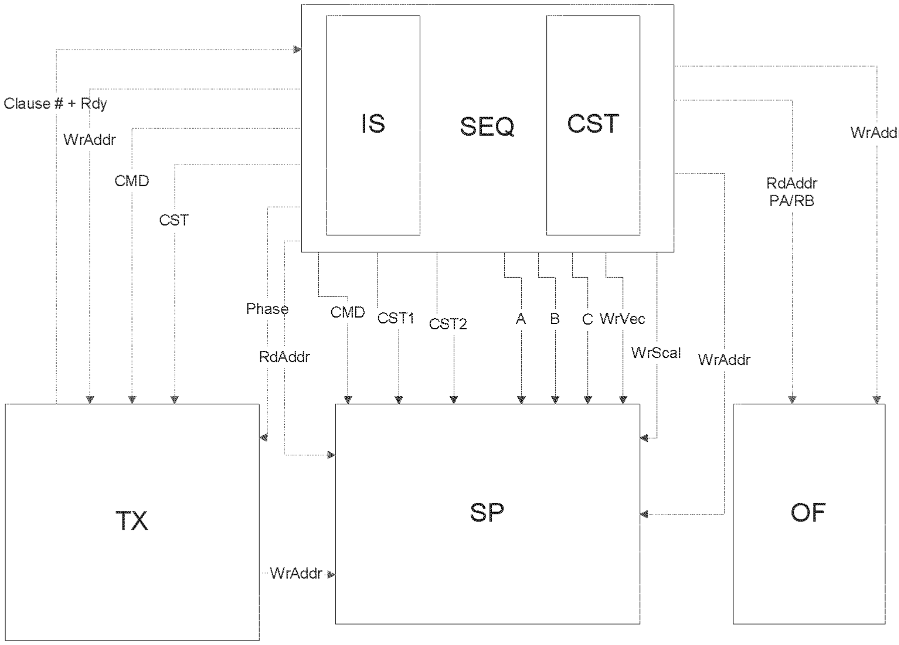


instruction
constants from RE
Register File
MAC
scalar input/output
pipeline stage
texture request
Register File
MAC
scalar input/output
pipeline stage
texture request
Scalar Unit
instruction
Register File
MAC
scalar input/output
pipeline stage
texture request
texture data
primitive data from RE
instruction
constants from RE
Register File
MAC
texture request
scalar input/output
Mux
texture address
to Primitive Assembly Unit or RenderBackend



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 Texture 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

Since each of the register file is actually physically divided (one 128x128 per MAC) and we don't have the place to hold a maximum size vector of vertices in the parameter buffer, we need to interpolate on a parameter basis rather than on a quad basis. So the order to the register file will be:

Q0P0 Q1P0 Q2P0 Q3P0 Q0P1 Q1P1 Q2P1 Q3P1 Q0P2 Q1P2 ...

3. Instruction Store

There is going to be only one instruction store for the whole chip. It may contain up to 2000 instructions of 96 bits each.

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

The read bandwith from this store is 24 bits/clock/pipe. To achieve this this instruction store is likely to be broken up into 4 blocks. An ALU instruction section (1R/1W) split in two and a texture section (1R/1W) also split in two. The bandwith out of those memories is 96 bits/clock.

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

4. Constant Store

Formatted: Bullets and Numbering

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 bits. The constant store is also planned to be shared. The read BW from the constant store is 512/4 bits/clock/pipe and the write bandwith is 32/4 bits/clock.

5. Looping and Branches

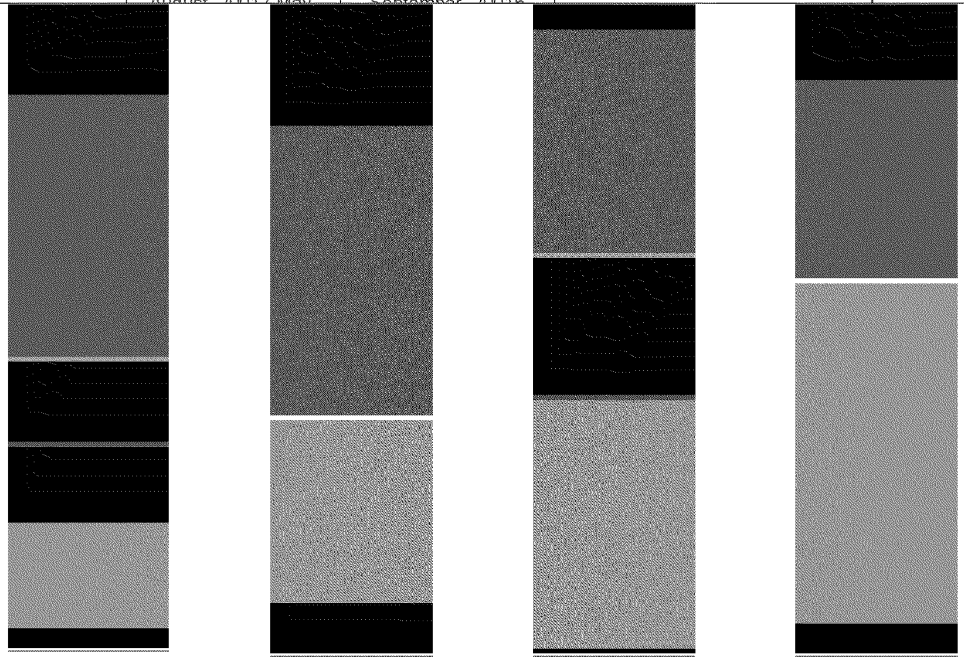
Formatted: Bullets and Numbering

Loops and branches are planned to be supported and will have to be dealt with at the sequencer level. However, it is still unclear if we plan on supporting data dependent branches or not.

6. Register file allocation

Formatted: Bullets and Numbering

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 as allowed moving again.

Formatted: Bullets and Numbering

~~2.~~7. Texture Arbitration

The texture arbitration logic chooses one of the 8 potentially pending texture 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 texture fetch per clock (or 4 fetches in one clock every 4 clocks) until all the texture fetch instructions of the clause are sent. This means that there cannot be any dependencies between two texture fetches of the same clause.

The arbitrator will not wait for the texture fetches to return prior to selecting another clause for execution. The texture pipe will be able to handle up to ~~100~~X(?) in flight texture fetches and thus there can be a fair number of active clauses waiting for their texture return data.

Formatted: Bullets and Numbering

~~3.~~8. ALU Arbitration

ALU arbitration proceeds in almost the same way than texture 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~~execute. If the packet chosen is a packet of vertices, the state machine issues one instruction every 4 clocks until the clause is finished. This means that the compiler has to insert nops between two dependent successive instructions. If the packet is a pixel packet it is made out of two sub-vectors of 16. Thus the state machine issues the first instruction for the first sub-vector and then, 4 clocks later, the first instruction of the second sub-vector and so on until the clause is finished.~~. 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):

Einst0 Oinst0 Einst1 Oinst1 Einst2 Oinst2 Einst0 Oinst3 Einst1 Oinst4 Einst2 Oinst0...
Proceeding this way hides the latency of 8 clocks of the ALUs.

Formatted: Bullets and Numbering

~~4.~~9. 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 ~~we have the ability to export at any clause~~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?). The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU arbitrer will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

Formatted: Bullets and Numbering

~~5.~~10. Content of the reservation station FIFOs

3 bits of Render State ~~and~~ 6-7 bits for the base address of the instruction store and some bits for LOD correction. Every other information (such as the coverage mask, quad address, etc.) is put in a FIFO and is retrieved when the quad exits the shader pipe to enter in the output file buffer. 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.

Formatted: Bullets and Numbering

~~6.~~11. The Output File (RB FIFO and Parameter Cache)

The output file is where program results are exported when the pixel/vertex shader finishes. It constists of a 512x128 memory cell that is statically divided between pixels and vertices. ~~Each section is a regular FIFO.~~ The output file has 1 write port and 1 read port. The sequencer is responsible for managing the addresses of this output file and for stalling the shader pipe should this output file fill up. The management is done by keeping the tail and head pointers of each sections (pixels and vertices) and incrementing them using a simple RoundRobin allocation policy. The sequencer must also arbitrate between the PA and the RB for the use of the read port. This arbitration will either be priority based or just interleaved evenly (1 read every 2 clocks for each of the blocks).

Formatted: Bullets and Numbering

~~7.~~12. Interfaces

~~7.1~~12.1 External Interfaces

~~7.1.1~~12.1.1 *Sequencer to Shader Engine Bus*

This is a bus that sends the instruction and constant data to all 4 Sub-Engines of the Shader. Because a new instruction is needed only every 4 clocks, the width of the bus is divided by 4 and both constants and instruction are sent over those 4 clocks.

Name	Direction	Bits	Description
Instruction Start	SEQ-> SP	1	High on first cycle of transfer
Constant 0	SEQ-> SP	32	128 bits transferred over 4 cycles, alpha first…blue last
Constant 1	SEQ-> SP	32	128 bits transferred over 4 cycles, alpha first…blue last
Instruction	SEQ-> SP	30	120 bits transferred over 4 cycles (order TBD) ?

Formatted: Bullets and Numbering

~~7.1.2~~12.1.2 *Shader Engine to Output File*

Every clock each Sub-Engine can output 128 bits of 'vector' data and 32 bits of 'scalar' data to an output file (?). This data will be compressed into 128 bits total prior to storage in output file.

Name	Direction	Bits	Description
UL_Vector_Out	SP-> OF	128	Vector Data out

UL_Scalar_Out	SP-> OF	32	Vector Data out
UR_Vector_Out	SP-> OF	128	Vector Data out
UR_Scalar_Out	SP-> OF	32	Vector Data out

Name	Direction	Bits	Description
LL_Vector_Out	SP-> OF	128	Vector Data out
LL_Scalar_Out	SP-> OF	32	Vector Data out
LR_Vector_Out	SP-> OF	128	Vector Data out
LR_Scalar_Out	SP-> OF	32	Vector Data out

~~7.1.3~~12.1.3 *Shader Engine to Texture Unit Bus (Fast Bus)*

Formatted: Bullets and Numbering

One quad's worth of addresses is transferred to Texture Unit every clock. These are sourced from a different pixel within each of the sub-engines repeating every 4 clocks. The ~~register file~~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.

One Quad's worth of Texture Data may be written to the ~~Register File~~Register file every clock. These are directed to a different pixel of the sub-engines repeating every 4 clocks. The ~~register file~~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_Read_Register_Index	SEQ->SP	8	Index into ~~Register File~~Register files for reading Texture Address
Tex_RegFile_Read_Data	SP->TEX	512	4 Texture Addresses read from the ~~Register File~~Register file
Tex_Write_Register_Index	SEQ->TEX	8	Index into ~~Register file~~Register file for write of returned Texture Data

~~7.1.4~~12.1.4 *Sequencer to Texture Unit bus (Slow Bus)*

Formatted: Bullets and Numbering

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

Name	Direction	Bits	Description
Tex_Ready	TEX→ SEQ	1	Data ready
Tex_Clause_Num	TEX→ SEQ	3	Clause number
Tex_cst	SEQ→TEX	?	Texture constants X bits sent over 4 clocks
Tex_Inst	SEQ→TEX	?	Texture fetch instruction X bits sent over 4 clocks

~~7.1.5~~12.1.5 *Shader Engine to RE/PA Bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Interpolator_Register_Index	SEQ->SP	8	Index into ~~Register File~~Register files for write of Interpolator/Index Data
Interpolator_Write_Mask	SEQ->SP	1	Write Mask. The same write mask is used for all 4 pixels
Interpolator_Write_Data	RE/PA->SP	512	4 interpolated vectors or vectors of indices

12.1.6 *PA? to sequencer*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Adress	PA→SEQ	?	Dealocation adress sent by the PA telling the Sequencer that it is now possible to free this space in the parameter buffer. This token is a pointer in the parameter cache and 4 bits to tell the size wich is to be freed up.

Formatted

Formatted

Formatted

Formatted

Formatted: Bullets and Numbering

13. Examples of program executions

13.1.1 *Sequencer Control of a Vector of Vertices*

1. PA sends a vector of 16 vertices (actually vertex indices – 32 bits/index for 512 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 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 512 bits/cycle)
   - the 16 vertex indices are sent to the 16 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 texture 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 texture clause 0 from the global instruction ~~store~~TSM0 was first selected by the TSM arbiter before it could start
7. all instructions of texture 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 Texture Unit to complete; it passes the register file write index for the texture 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 texture 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

13.1.2 *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~~ Parameter Buffer is loaded from the Parameter Cache before the SEQ takes control of the vector
   - after the HZ culling stage a request is made by the RE to send parameter data to the Parameter buffer
   - the Parameter buffer is wide enough to source 3 vertices worth of a particular parameter in one cycle
   - **at this moment the right sequencer will free up the parameter store locations not used anymore using the token provided by the PA**.
3. 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 one quad's worth of barycentrics per cycle
4. 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
5. 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
6. SEQ control starts with the interpolation of parameters (up to 16 per thread) by sending the barycentric coordinates from the Pixel FIFO and the parameters from the Parameter Buffer to the interpolator
   - P0i, P0j, and P0k (the value of P0 at each vertex) are loaded into the interpolator from the Parameter buffer
   - Q0 i, j, and k are loaded into the interpolator from the Pixel FIFO
   - The interpolator then generates the parameter value for each pixel in Q0 (Q0P0)
   - **P0i, P0j, and P0k are sent to the interpolator for Q1 only if Q1 is from a different primitive; if Q1 is from the same primitive as Q0, then the P0i, P0j, and P0k values loaded for Q0 are held by the interpolator and reused for Q1**
     - **a "different_prim" control bit is passed with the barycentric data for each quad in the Pixel FIFO that indicates whether new parameter data needs to be loaded into the interpolator**
   - Q1 i, j, and k are then loaded into the interpolator from the Pixel FIFO
   - The interpolator then generates the parameter value for each pixel in Q1 (Q1P0)
   - Q2P0 and Q3P0 are generated in a similar manner
   - The next set of parameter data - P1i, P1j, and P1k - is then loaded into the interpolator
   - Q0 i, j, and k now must be re-read from the Pixel FIFO – this means that the output of the Pixel FIFO loops through the top four entries on each read command until at the end a final "block_pop" signal is asserted, causing the top four sets of barycentric coordinates to finally be removed
   - so the order of parameter info generated is Q0P0, Q1P0, Q2P0, Q3P0, Q0P1, Q1P1, etc.
7. SEQ controls the transfer of interpolated data to the SP register file over the RE_SP interface (which has a bandwidth of 512 bits/cycle)
   - 16 pixels worth of interpolated parameter data is sent to the 16 register files over 4 cycles
     - RF0 of SU0, SU1, SU2, and SU3 is written with Q0P0 the first cycle
     - RF1 of SU0, SU1, SU2, and SU3 is written with Q1P0 second cycle
     - RF2 of SU0, SU1, SU2, and SU3 is written with Q2P0 third cycle
     - RF3 of SU0, SU1, SU2, and SU3 is written with Q3P0 fourth cycle

8. SEQ constructs a control packet for the vector and sends it to the first reservation station (the FIFO in front of texture 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 (such as quad address for example) travels in a separate FIFO

9. TSM0 accepts the control packet and fetches the instructions for texture clause 0 from the global instruction ~~store~~TSM0 was first selected by the TSM arbiter before it could start

10. all instructions of texture clause 0 are issued by TSM0

11. 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 texture requests made to the Texture Unit to complete; it passes the register file write index for the texture 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

12. 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

13. 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 texture state machine 1, or TSM1 FIFO)

14. 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

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

13.1.3 *Notes*

Formatted: Bullets and Numbering

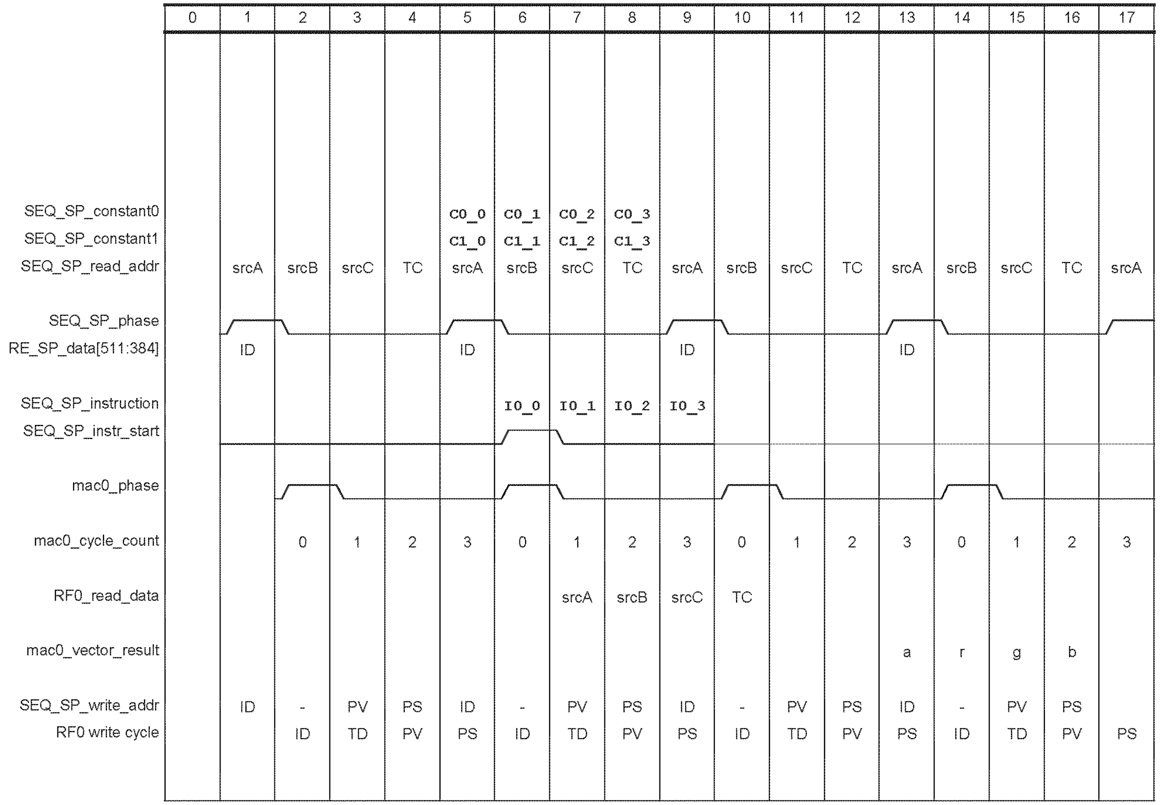
16. the state machines and arbitrers will operate ahead of time so that they will be able to immediately start the real threads or stall.

17. 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

18. Waterfalling, parameter buffer allocation, loops and branches and parameter cache de-allocation still needs to be specked out.

14. Timing Diagrams

14.1 MAC 0

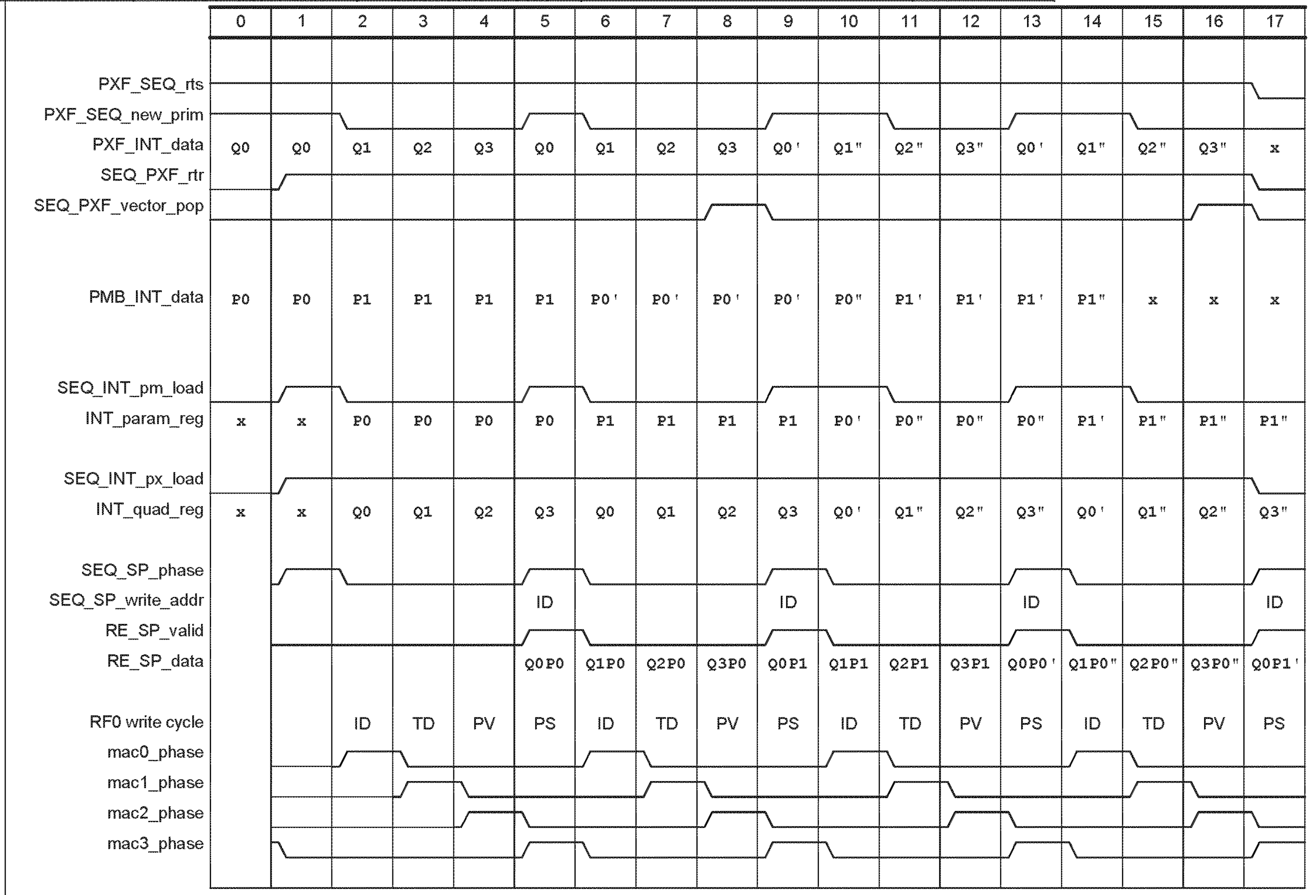


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
SEQ_SP_constant0						C0_0	C0_1	C0_2	C0_3									
SEQ_SP_constant1						C1_0	C1_1	C1_2	C1_3									
SEQ_SP_read_addr		srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA
SEQ_SP_phase																		
RE_SP_data[511:384]		ID				ID				ID				ID				
SEQ_SP_instruction							I0_0	I0_1	I0_2	I0_3								
SEQ_SP_instr_start																		
mac0_phase																		
mac0_cycle_count			0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
RF0_read_data								srcA	srcB	srcC	TC							
mac0_vector_result														a	r	g	b	
SEQ_SP_write_addr		ID	-	PV	PS	ID	-	PV	PS	ID	-	PV	PS	ID	-	PV	PS	
RF0 write cycle			ID	TD	PV	PS	ID	TD	PV	PS	ID	TD	PV	PS	ID	TD	PV	PS

Timing Diagram 1: Sequencer to Shader Pipe 0, Shader Unit 0, MAC 0

14.2 Sequencer to Shader Pipe

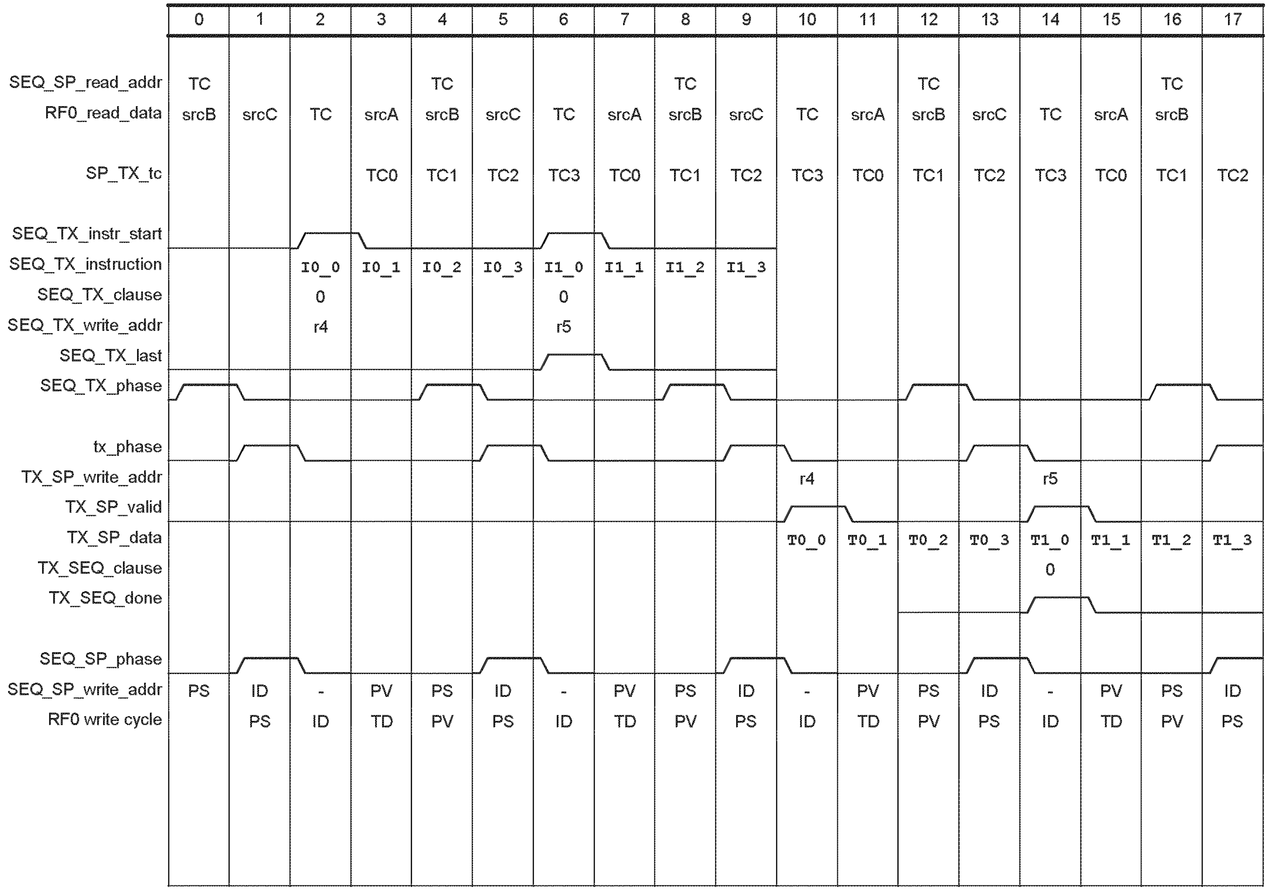
Formatted: Bullets and Numbering


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
PXF_SEQ_rts
PXF_SEQ_new_prim
PXF_INT_data
Q0 Q0 Q1 Q2 Q3 Q0 Q1 Q2 Q3 Q0' Q1" Q2" Q3" Q0' Q1" Q2" Q3" x
SEQ_PXF_rtr
SEQ_PXF_vector_pop
PMB_INT_data
P0 P0 P1 P1 P1 P1 P0' P0' P0' P0' P0" P1' P1' P1' P1" x x x
SEQ_INT_pm_load
INT_param_reg
x x P0 P0 P0 P0 P1 P1 P1 P1 P0' P0" P0" P0" P1' P1" P1" P1"
SEQ_INT_px_load
INT_quad_reg
x x Q0 Q1 Q2 Q3 Q0 Q1 Q2 Q3 Q0' Q1" Q2" Q3" Q0' Q1" Q2" Q3"
SEQ_SP_phase
SEQ_SP_write_addr
ID ID ID ID
RE_SP_valid
RE_SP_data
Q0P0 Q1P0 Q2P0 Q3P0 Q0P1 Q1P1 Q2P1 Q3P1 Q0P0' Q1P0" Q2P0" Q3P0" Q0P1'
RF0 write cycle
ID TD PV PS ID TD PV PS ID TD PV PS ID TD PV PS
mac0_phase
mac1_phase
mac2_phase
mac3_phase

Timing Diagram 2: RE Interpolator to Shader Pipe Data Transfer

Formatted: Bullets and Numbering

14.3 Sequencer to Texture Pipe



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
SEQ_SP_read_addr	TC				TC				TC				TC				TC	
RF0_read_data	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	
SP_TX_tc				TC0	TC1	TC2	TC3	TC0	TC1	TC2	TC3	TC0	TC1	TC2	TC3	TC0	TC1	TC2
SEQ_TX_instr_start																		
SEQ_TX_instruction			I0_0	I0_1	I0_2	I0_3	I1_0	I1_1	I1_2	I1_3								
SEQ_TX_clause			0				0											
SEQ_TX_write_addr			r4				r5											
SEQ_TX_last																		
SEQ_TX_phase																		
tx_phase																		
TX_SP_write_addr											r4				r5			
TX_SP_valid																		
TX_SP_data											T0_0	T0_1	T0_2	T0_3	T1_0	T1_1	T1_2	T1_3
TX_SEQ_clause															0			
TX_SEQ_done																		
SEQ_SP_phase																		
SEQ_SP_write_addr	PS	ID	-	PV	PS	ID	-	PV	PS	ID	-	PV	PS	ID	-	PV	PS	ID
RF0 write cycle		PS	ID	TD	PV	PS	ID	TD	PV	PS	ID	TD	PV	PS	ID	TD	PV	PS

Timing Diagram 3: Sequencer - Texture Unit Interface and Texture Unit - Shader Pipe Data Transfer

14.4 Timing diagrams explanations

Formatted: Bullets and Numbering

The numbering of the four shader pipes, the four shader units, and the four MACs is from left to right and from 0 to 3. So for example the most significant 512 bits of a SP goes to SU0 and the least significant 512 bits go to SU3; within SU0, the most significant 128 bits go to MAC0 and the least significant 128 bits go to MAC3.
The following assumptions are made:

1. all block to block signals are register to register
2. for register file reads, the RF read data is available in the MAC one clock after a RF read address is registered into the MAC (this is the same as saying the read data is valid out of the RF two clocks after the address is asserted on the SEQ to SP interface)

Formatted: Bullets and Numbering

14.4.1 *Timing Diagram 1: Sequencer to Shader Pipe 0, Shader Unit 0, MAC 0*

This diagram shows the basics of the Sequencer to Shader Pipe interface. For simplicity only the timing relative to MAC0 is shown. The timing for MAC1 is one clock later than MAC0, MAC2 one clock later than MAC1, etc. This means that most of the signals need to be delayed in the SP by one cycle for MAC1, two cycles for MAC2, and three cycles for MAC3.
SEQ_SP_constant0: Constant 0 (128 bits over 4 cycles). Pipelined in SP for other MACs.
SEQ_SP_constant1: Constant 1 (128 bits over 4 cycles). Pipelined in SP for other MACs.
SEQ_SP_read_addr: Register File Read Address (8 bits).   Pipelined in SP for other MACs.
SEQ_SP_phase: This signal syncs the data transfer to the RF from the RE, as well as defining the order of all writes into the RF. It is asserted during the cycle that interpolated data (ID) is valid on the RE_SP_ID bus. Pipelined in SP for other MACs.
RE_SP_ID[511:384]: This is the most significant 128 bits of the RE_SP_data interface (meaning that this MAC0 is in SU0).
SEQ_SP_instruction: 96 bits of instruction are sent over 4 cycles. Pipelined in SP for other MACs.
SEQ_SP_instr_start: control bit that signals the first cycle of the instruction transfer. Pipelined in SP for other MACs.
mac0_phase: registered version of SEQ_SP_phase used in MAC0 (this may not be he actual signal name).
mac0_cycle_count: a counter inside the MAC that keeps track of the RF write cycles; 0 here corresponds to the cycle RE interpolated data is written (this may not be he actual signal name).
RF0_read_data: data that is read out of MAC0's register file (this may not be he actual signal name).
mac0_vector_result: the 32-bit output of the vector ALU (PV is built up over 4 cycles) (this may not be he actual signal name).
SEQ_SP_write_addr: Register File Write Address (8 bits).   Note that the SEQ does not send the Texture Data write address over this bus. Pipelined in SP for other MACs.
RF0 write cycle: the cycles allocated to the different write sources (ID = Interpolated Data, TD = Texture Data, PV = Previous Vector, PS = Previous Scalar) (not a signal – just a reference point on the diagram).

14.4.2 *Timing Diagram 2: RE Interpolator to Shader Pipe Data Transfer*

Formatted: Bullets and Numbering

This diagram shows how pixel data (barycentric coordinates i, j, and k) is sent from the Pixel FIFO to the interpolator under SEQ control, and how parameter data (for each vertex) is also sent to the interpolator under SEQ control. The output of the Interpolator is then shown being sent over the RE_SP interface.
PXF_SEQ_rts: Indicates that the output of the pixel FIFO is valid.
PXF_SEQ_new_prim: The current output of the Pixel FIFO is from a different primitive that the previous output. Tells the SEQ that new parameter info must be fetched (if its not from a new prim, then new parameter data is not needed).
PXF_INT_data: Data output of the Pixel FIFO – goes to the Interpolator.
SEQ_PXF_rtr: Indicates that the current Pixel FIFO output will be taken by the Interpolator (driven by SEQ). Then next quad of data will be driven the next cycle.
SEQ_PXF_vector_pop: SEQ tells the Pixel FIFO to pop a vector of pixels (otherwise RTRs cause the data to be cycled between the four quads).

PMB_INT_data: Data from the Parameter Buffer to the Interpolator. (Note that the control of the parameter buffer is TBD).

SEQ_INT_pm_load: controls the loading of parameter data into the Interpolator.

INT_param_reg: register in the Interpolator that holds the per-vertex parameter data while the per-pixel parameters are generated for one or more quads (may not be the actual signal name).
SEQ_INT_px_load: controls the loading of pixel data into the Interpolator.
INT_quad_reg: : register in the Interpolator that holds one quad's worth of pixel data(may not be the actual signal name).

SEQ_SP_phase: see above under TD1.
SEQ_SP_write_addr: see above under TD1.
RE_SP_valid: Interpolator Data Valid – indicates that the SP should write the ID on the appropriate cycle.
RE_SP_data: Data from the RE interpolator to the SP.
RF0 write cycle: see above under TD1.
mac*_phase: see above under TD1. These phase signals help to show the timing offset between the MACs. Note also that each Shader Unit has a set of these signals (all with the same timing).

14.4.3 *Timing Diagram 3: Sequencer - Texture Unit Interface and Texture Unit - Shader Pipe Data Transfer*

Formatted: Bullets and Numbering

This diagram starts with the texture coordinate read from the register file and its transfer to the TX. The instruction transfer is then shown, followed by the texture data transfer to the shader pipe.
SEQ_SP_read_addr: see above. Here shows the cycle that the texture coordinate read address is asserted.
RF0_read_addr: see above.
SP_TX_tc: Texture coordinate data sent from the shader pipe to the texture unit.
SEQ_TX_instr_start: Asserted on the first cycle of a SEQ to TX instruction transfer.
SEQ_TX_instruction: 96 bits of texture instruction transferred over 4 cycles.
SEQ_TX_clause: the clause number associated with this instruction.
SEQ_TX_write_addr: RF write index used by TX for returned texture data.
SEQ_TX_last: indicates that this is the last texture instruction of a clause.
SEQ_TX_phase: syncs the texture data write. Note that it is asserted early enough to be registered into TX and still allow TX to source the texture data to the SP on the correct cycle.

tx_phase: the phase signal after being registered into TX.
TX_SP_write_addr: RF write index for texture data.
TX_SP_valid: indicates that valid texture data is being driven to the SP.
TX_SP_data: the texture data.
TX_SEQ_clause: the clause number associated with the texture data.
TX_SEQ_done: indicates to the SEQ that the texture data transfer is complete for the clause number that is on the TX_SEQ_clause bus.

SEQ_SP_phase: see above under TD1 - shown here for reference.
SEQ_SP_write_addr: see above under TD1- shown here for reference.
RF0 write cycle: see above under TD1- shown here for reference.

Formatted: Bullets and Numbering

~~8.~~15. 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 from the texture store to feed the ALUs. Two solutions exists for this problem:

1) Let the compiler handle the case and put those instructions in a texture clause so we can use the bandwith 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. 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~~register file as well as on the ~~register file~~register file allocation method (dynamic VS static).

~~Ability to export at any clause?~~

Saving power?

~~Are we working on 32 vertices at a time or 16?~~

Size of the fifo containing the information of a vector of pixels/vertices. And size of the fifos before the reservation stations.

Sequencer Instruction memory, and constant memory.

Arbitration policy for the output file.

Loops and branches.

The parameter cache may end up in the PA rather than in the RS. Parameter cache management thus may change.

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 200124 September, 200114	4 September, 20153 October, 200124	GEN-CXXXXX-REVA	1 of 31

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SEQ

Version 0.6~~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\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.

ATI 2012
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.~~4~~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.

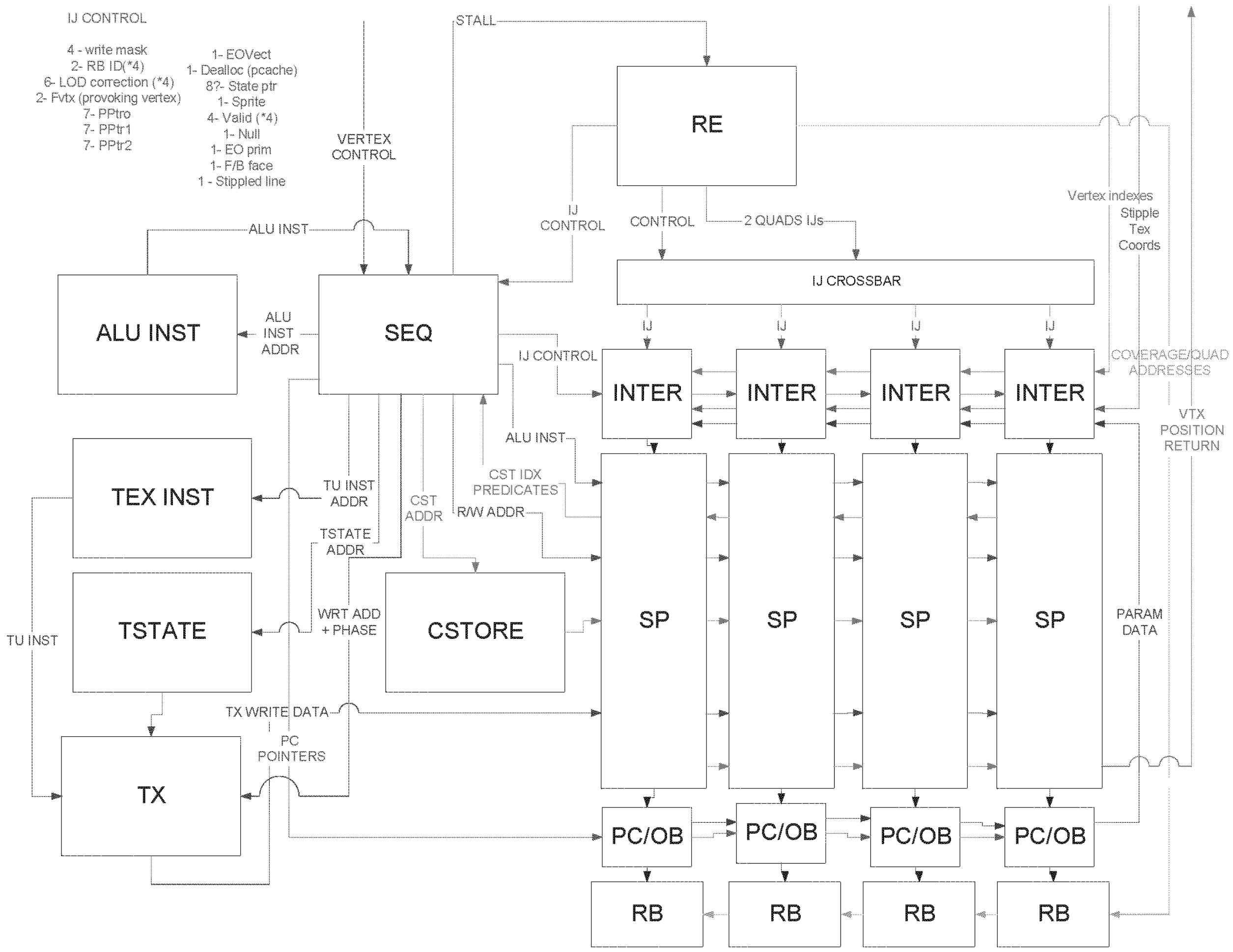
1. Overview

The sequencer first arbitrates between vectors of ~~16~~ 64 vertices that arrive directly from primitive assembly and vectors of ~~4~~ 16 quads (~~16~~ 64 pixels) that are generated in the raster engine.

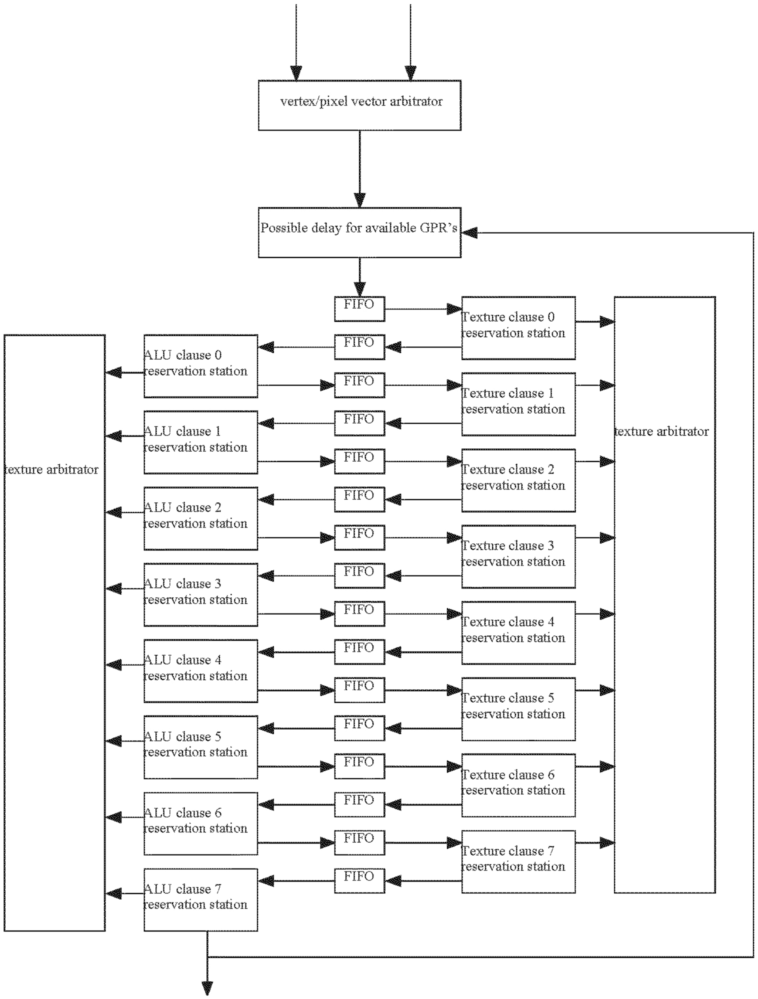
The vertex or pixel program specifies how many GPR's it needs to execute. The sequencer will not start the next vector until the needed space is available.

The sequencer is based on the R300 design. It chooses two ALU clauses and a texture 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 texture and eight ALU clauses, but clauses do not need to contain instructions. A vector of pixels or vertices ping-pongs along the sequencer FIFO, bouncing from texture reservation station to alu reservation station. A FIFO exists between each reservation stage, holding up vectors until the vector currently occupying a reservation station has left. A vector at a reservation station can be chosen to execute. The sequencer looks at all eight alu reservation stations to choose an alu clause to execute and all eight texture stations to choose a texture 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 texture fetches initiated by the previous texture 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 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
Vertex indexes
Stipple
Tex
Coords
ALU INST
ALU INST
ALU INST ADDR
SEQ
IJ CROSSBAR
IJ
IJ
IJ
IJ
IJ CONTROL
INTER
INTER
INTER
INTER
COVERAGE/QUAD ADDRESSES
VTX POSITION RETURN
ALU INST
CST IDX PREDICATES
TEX INST
TU INST ADDR
CST ADDR
R/W ADDR
TSTATE ADDR
SP
SP
SP
SP
PARAM DATA
TU INST
TSTATE
WRT ADD + PHASE
CSTORE
TX WRITE DATA
PC POINTERS
TX
PC/OB
PC/OB
PC/OB
PC/OB
RB
RB
RB
RB

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.

The rasterizer always checks the vertices FIFO first and if allowed by the sequencer sends the data to the shader. If the vertex FIFO is empty then, the rasterizer takes the first entry of the pixel FIFO (a vector of ~~16~~ 64 pixels) and sends it to the interpolators. Then the sequencer takes control of the packet. The packet consists of ~~3~~ 20 bits of state, 6-7 bits for the base address of the Shader program and some information on the coverage to determine texture LOD. All other information (2x2 adresses) is put in a FIFO (one for the pixels and one for the vertices) and retrieved when the packet finishes its last clause.

{Issue: How many bits of state exactly?}

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 texture machine issues a texure request and corresponding register address for the texture address (ta). A small command (tcmd) is passed to the texture system identifying the current level number (0) as well as the register write address for the texture return data. One texture request is sent every 4 clocks causing the texturing of ~~four~~ sixteen 2x2s worth of data (or ~~16~~ 64 vertices). Once all the requests are sent the packet is put in FIFO 1.

Upon recept of the return data, the texture unit writes the data to the register file using the write address that was provided by the level 0 texture machine and sends the clause number (0) to the level 0 texture state machine to signify that the write is done and thus the data is ready. Then, the level 0 texture 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.

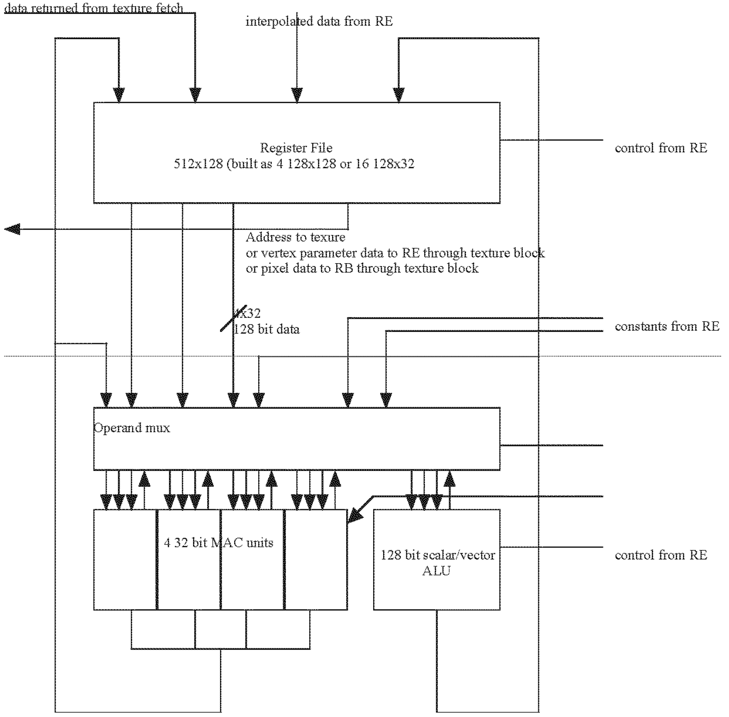
There will always be two active ALU clauses at any given time (and two arbitrers). One arbitrer will arbitrate over the odd clock cycles and the other one will arbitrate over the even clock cycles. The only constraints between the two arbitrers is that they are not allowed to pick the same clause number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

If the packet is a vertex packet, upon reaching ALU clause 4, it can export the position if the position is ready. So the arbitrer must prevent ALU clause 4 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).

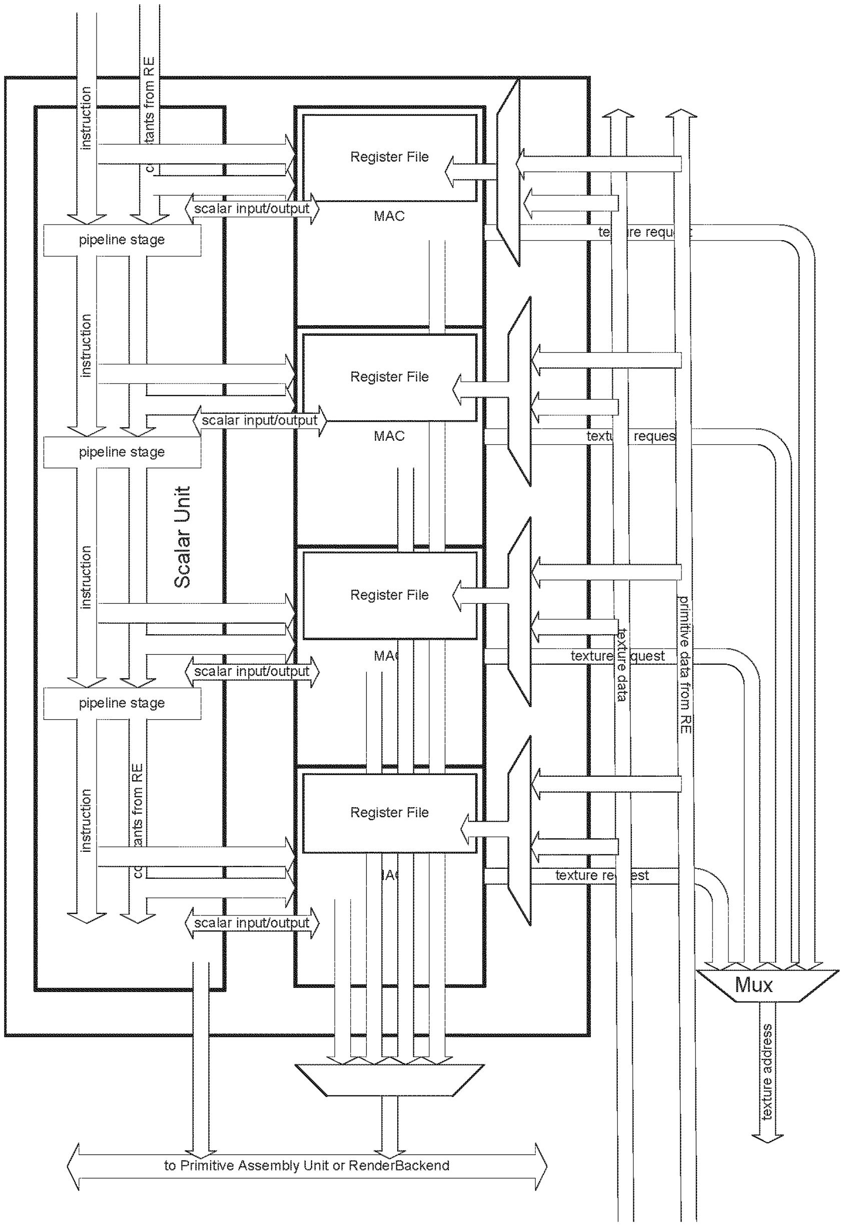
All other level process in the same way until the packet finally reaches the last ALU machine (8). On completion of the level 8 ALU clause, a valid bit is sent to the Render Backend which picks up the color data. This requires that the last instruction writes to the output register – a condition that 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.

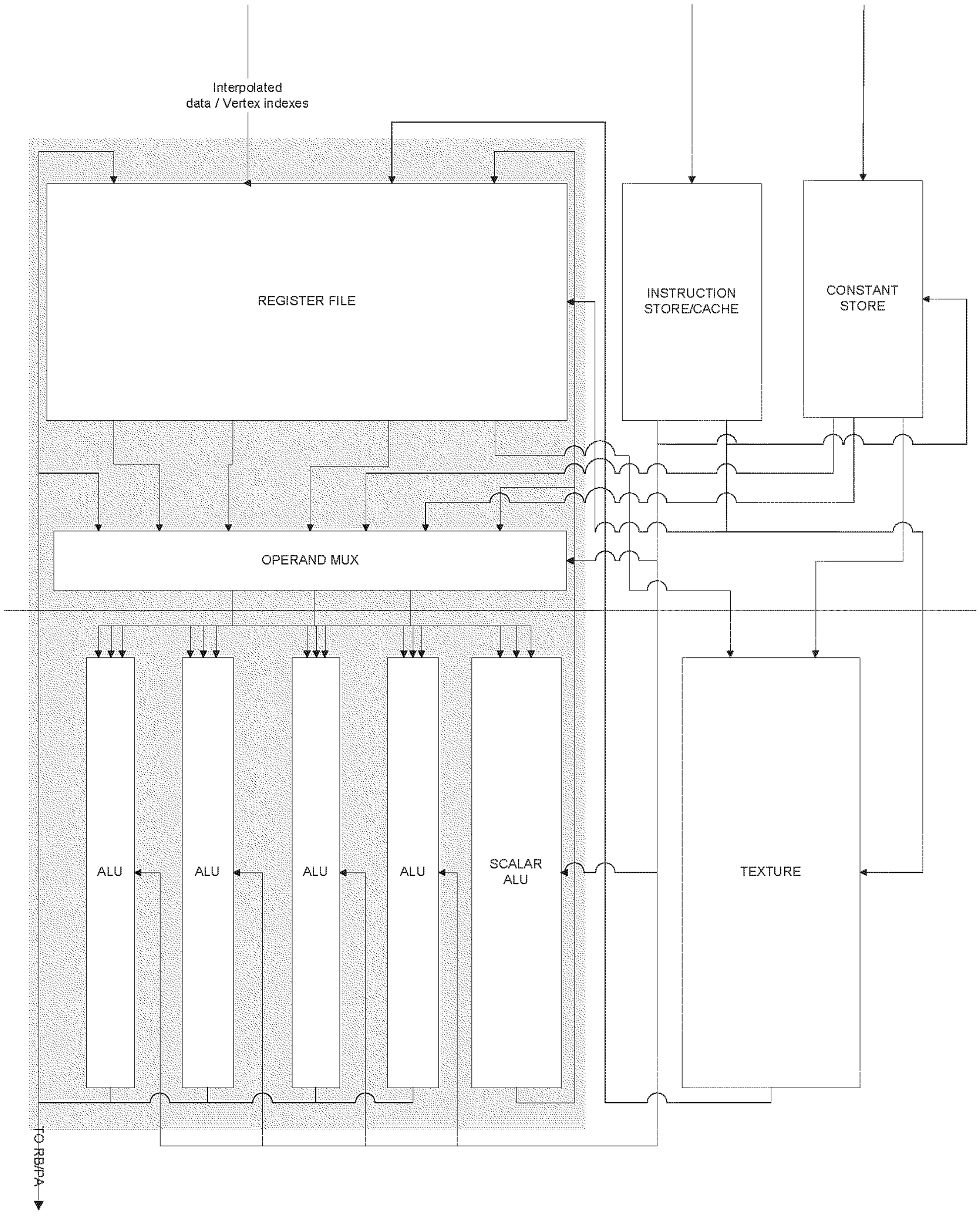
Only two ALU state machine may have access to the register file address bus or the instruction decode bus at one time. Similarly, only one texture state machine may have access to the register file address bus at one time. Arbitration is performed by three arbitrer blocks (two for the ALU state machines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

~~Each state machine maintains an address pointer specifying where the 16 entries vector is located in the register file (the texture machine has two pointers one for the read address and one for the write). Upon completion of its job, the address pointer is incremented by a predefined amount equal to the total number of registers required by the shading code. A comparison of the address pointer for the first state machine in the chain (the input state machine), and the last machine in the chain (the level 8 ALU machine), gives an indication of how much unallocated register file memory is available~~


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

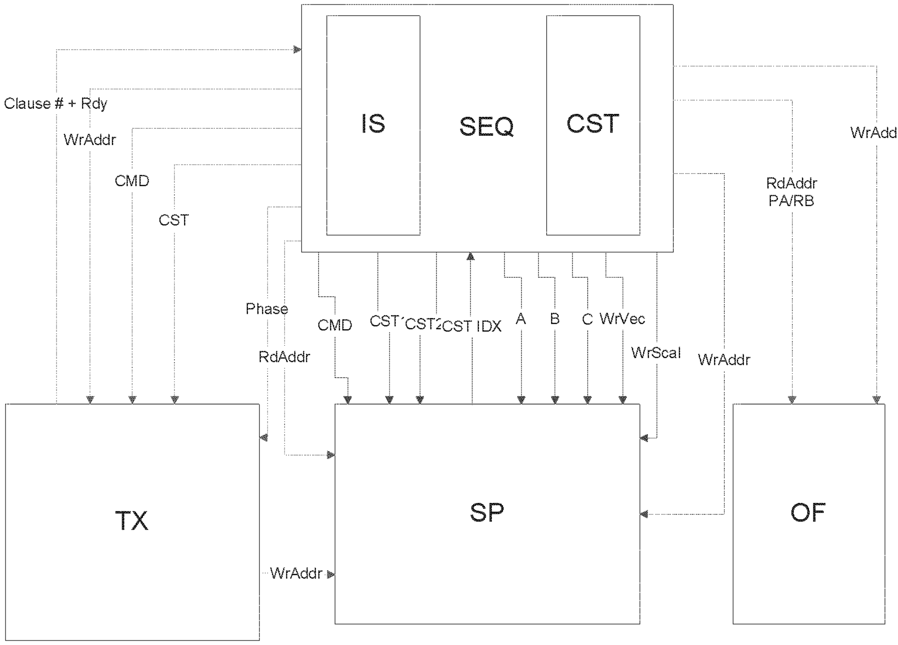
1.2 Data Flow graph





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

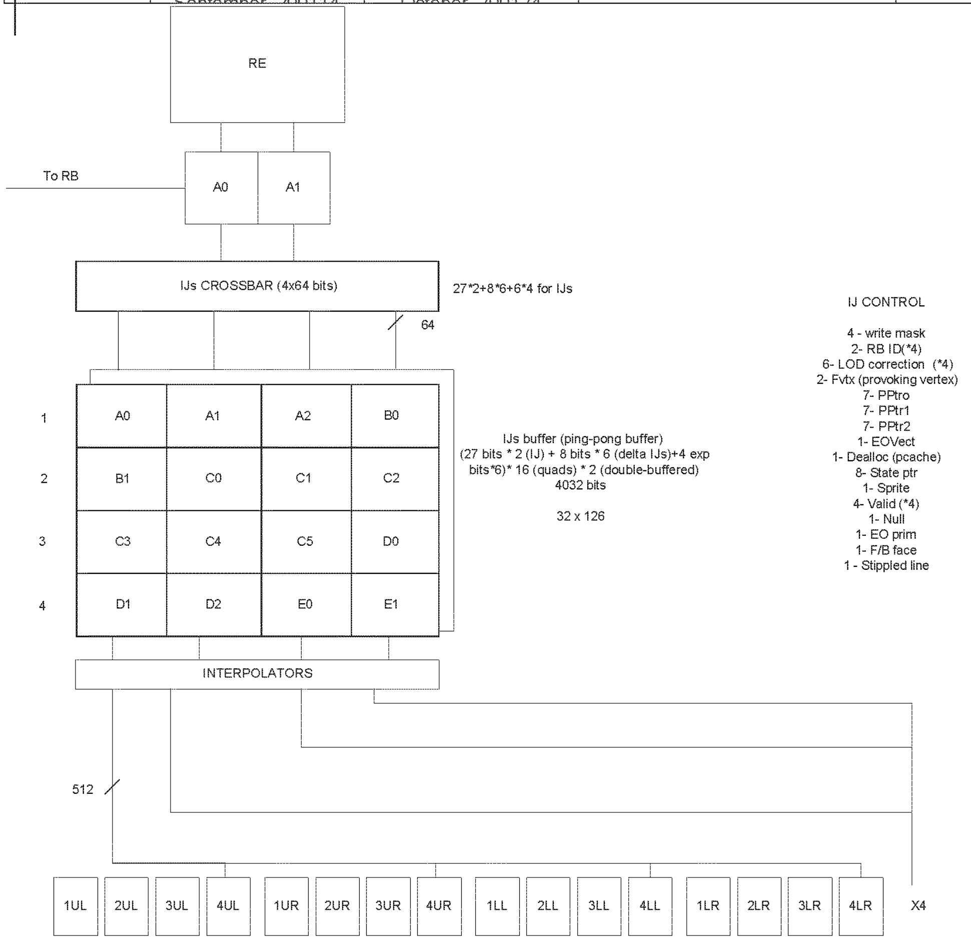
1.3 Control Graph



In green is represented the Texture 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.



1	A0	A1	A2	B0
2	B1	C0	C1	C2
3	C3	C4	C5	D0
4	D1	D2	E0	E1

IJs buffer (ping-pong buffer)
(27 bits * 2 (IJ) + 8 bits * 6 (delta IJs)+4 exp bits*6)* 16 (quads) * 2 (double-buffered)
4032 bits

32 x 126

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

ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
24 September, 2001~~2~~4 September, 200114	4 September, 2015~~3~~ October, 200124	GEN-CXXXXX-REVA	13 of 31

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19
SP0	A0	B1	C3	D1					A0	B1	C3	D1					V 0-3	V 16-19	V 32-35	V 48-51
SP1	A1		C4	D2		C0			A1		C4	D2		C0			V 4-7	V 20-23	V 36-39	V 52-55
SP2	A2		C5			C1		E0	A2		C5			C1		E0	V 8-11	V 24-27	V 40-43	V 56-59
SP3					B0	C2	D0	E1					B0	C2	D0	E1	V 12-15	V 28-31	V 44-47	V 60-63
				P0								P1								

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.~~Since each of the register file is actually physically divided (one 128x128 per MAC) and we don't have the place to hold a maximum size vector of vertices in the parameter buffer, we need to interpolate on a parameter basis rather than on a quad basis. So the order to the register file will be:~~

~~Q0P0 Q1P0 Q2P0 Q3P0 Q0P1 Q1P1 Q2P1 Q3P1 Q0P2 Q1P2 …~~

3. Instruction Store

There is going to be only one instruction store for the whole chip. It ~~may~~ will contain ~~up to 200~~40960 instructions of 96 bits each. There is also going to be a control instruction store of 256x32.

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

The read bandwith from this store is ~~24~~ 96*2 bits/ 4 clocks (48 bits/clock)~~/pipe~~. ~~To achieve this this instruction store is likely to be broken up into 4 blocks. An ALU instruction section (1R/1W) split in two and a texture section (1R/1W) also split in two.~~ The bandwith out of those memories is ~~96~~ 48 bits/clock. It is likely to be a 1R/1W port memory; we use 2 clocks to load the ALU instruction and 2 clocks to load the Texture instruction.

Formatted: Bullets and Numbering

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).

Formatted: Bullets and Numbering

~~4.~~5. Constant Store

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 bits. The constant store is also planned to be shared. The read BW from the constant store is 128~~512/4~~ bits/clock~~/pipe~~ 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.

Formatted: Bullets and Numbering

~~5.~~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. The control program has 4 instructions:

Formatted: Bullets and Numbering

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[31:0]
Loop_count[7:0][15:0]
Loop_Start[7:0] [15:0]
Loop_End[7:0] [15:0]

6.2 The Control Flow Program

Formatted: Bullets and Numbering

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. I believe that this method has increased area and complexity when the program size is increased.

The Method I prefer is a "control program"
The control program has four basic instructions:
Execute
Conditional_execute
Loop_start
Loop_end

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.
Loop_end increments (decrements?) the loop counter and jumps back the specified number of instructions if the loop end condition is not met.

if we try and fit the control flow instructions into 32 bit words, the following instructions are possible choices:

Execute																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	instruction_count												Reserved			Address												

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

Conditional Execute																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	Boolean						0: = 0 1: = 1 2,3:NA			Instruction_count						Address												

if the specified boolean (6 bits can address 64 booleans) meets the specified condition then execute the specified instructions (up to 64 instructions)

Conditional Execute Predicates																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	Reserved =0						0: = 0 1: = 1 2,3:NA			Instruction_count						Address												

Check the OR of all current predicate bits. If OR matches the condition execute the specified number of instructions.

Loop_Start																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

0	0	1	0																								Loop ID				

Initialize the specified loop

Loop_End																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	Reserved (must = 0)												Start Address								Reserved (must = 0)			Loop ID				

if the loop condition of the current loop is not met, then branch back to the specified address in the control flow program. Note that jumping back to the loop_start results in an infinite loop, the jump should be to loop_start+1.

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

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

The control program can be up to 256 instructions in size. (There is an offset added to the address from the render state before accessing the control flow program memory to allow for multiple programs resident at the same time)

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 addresses from the control program are added to another offset to allow for multiple programs resident at the same time.

Under this model, all subroutine calls must be inlined into the control program.

6.3 Data dependant predicate instructions

Formatted: Bullets and Numbering

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

PRED_SETE - similar to SETE except that the result is 'exported' to the sequencer.
PRED_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 the 64 bit predicate vector and use it to control the write masking (two sets for interleaved operation). This predicate is not maintained across clause boundaries.

Then we have two conditional execute bits. The first bit is a conditional execute "on" bit and the second bit tells us if we execute on 1 or 0. For exemple, 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 Register file indexing

Formatted: Bullets and Numbering

Because we can have loops in texture clause, we need to be able to index into the register file in order to retrieve the data created in a texture 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 :

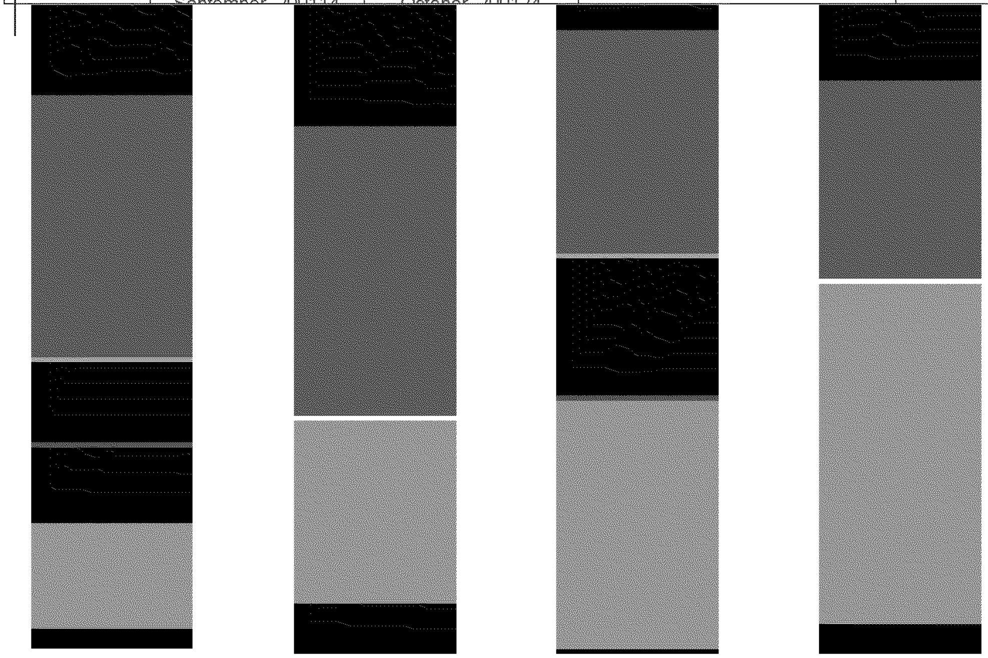
Bit7	Bit 6	
0	0	'absolute register'
0	1	'relative register'
1	0	'previous vector'
1	1	'previous scalar'

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_counter and this becomes our new address that we give to the shader pipe.~~However, it is still unclear if we plan on supporting data dependent branches or not.~~

~~6.~~7. Register file allocation

Formatted: Bullets and Numbering

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 ~~as~~ is allowed to moving~~e~~ again.

Formatted: Bullets and Numbering

~~7.~~8. Texture Arbitration

The texture arbitration logic chooses one of the 8 potentially pending texture 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 texture fetch per clock (or 4 fetches in one clock every 4 clocks) until all the texture fetch instructions of the clause are sent. This means that there cannot be any dependencies between two texture fetches of the same clause.

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

Formatted: Bullets and Numbering

~~8.~~9. ALU Arbitration

ALU arbitration proceeds in almost the same way than texture 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):

Einst0 Oinst0 Einst1 Oinst1 Einst2 Oinst2 Einst0 Oinst3 Einst1 Oinst4 Einst2 Oinst0...
Proceeding this way hides the latency of 8 clocks of the ALUs.

~~9.~~10. Handling Stalls

Formatted: Bullets and Numbering

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?). The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU arbitrer will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

~~10.~~11. Content of the reservation station FIFOs

Formatted: Bullets and Numbering

3 bits of Render State 6-7 bits for the base address of the instruction store, ~~and~~ some bits for LOD correction and coverage mask information in order to fetch texture for only valid pixels. Every other information (such as the coverage mask, quad address, etc.) is put in a FIFO and is retrieved when the quad exits the shader pipe to enter in the output file buffer. 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.

~~11.~~12. The Output File (RB FIFO and Parameter Cache)

Formatted: Bullets and Numbering

The output file is where program results are exported when the pixel/vertex shader finishes. It constists of a 512x128 memory cell that is statically divided between pixels and vertices. The output file has 1 write port and 1 read port. The sequencer is responsible for managing the addresses of this output file and for stalling the shader pipe should this output file fill up. The management is done by keeping the tail and head pointers of each sections (pixels and vertices) and incrementing them using a simple RoundRobin allocation policy. The sequencer must also arbitrate between the PA and the RB for the use of the read port. This arbitration will either be priority based or just interleaved evenly (1 read every 2 clocks for each of the blocks).

13. Registers

Formatted: Bullets and Numbering

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)
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
PCNTLSHADER	base pointer for the pixel control program
VCNTLSHADER	base pointer for the vertex control program
VWRAP	wrap point for the vertex shader instruction store
PWRAP	wrap point for the pixel shader instruction store
REG_ALLOC_PIX	number of registers to allocate for pixel shader programs
REG_ALLOC_VERT	number of registers to allocate for vertex shader programs
PARAM_MASK[0…16]	parameter mask to specify wich parameters the pixel shader
FLAT_GOUR[0…16]	wich parameters are to be gouraud shaded
GEN_TEX[0….16]	for wich parameters do we need to generate tex coords.
CYL_WRAP[0…64]	for wich vertices do we do the cyl wrapping.
P_EXPORT	number of exports for pixel shader
V_EXPORT	number of exports for vertex shader (also the number of interpolated parameters)
V_EXPORT_LOC	Vertex shader exporting to RB or the PCACHE

Formatted: Bullets and Numbering

~~12.~~14. Interfaces

~~12.1~~14.1 External Interfaces

~~12.1.1~~14.1.1 *~~Sequencer to Shader Engine Bus~~PA/SC to RE : IJ bus*

~~This is a bus that sends the instruction and constant data to all 4 Sub-Engines of the Shader. Because a new instruction is needed only every 4 clocks, the width of the bus is divided by 4 and both constants and instruction are sent over those 4 clocks.~~ 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

Name	Direction	Bits	Description
~~Instruction Start~~IJs	~~SEQ-> SP~~PA→RE	~~1~~64	~~High on first cycle of transfer~~IJ information sent over 2 clocks

Formatted: Bullets and Numbering

~~12.1.2~~

~~*12.1.2Shader Engine to Output File*~~

~~Every clock each Sub-Engine can output 128 bits of 'vector' data and 32 bits of 'scalar' data to an output file (?). This data will be compressed into 128 bits total prior to storage in output file.~~

Formatted: Bullets and Numbering

14.1.2 *PA/SC to SEQ : 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.

Name	Direction	Bits	Description
Write Mask	PA→SEQ(RE)	4	Quad Write mask left to right
RB_ID	PA→SEQ(RE)	8	RB id for each quad sent 2 bits per quad
LOD_CORRECT	PA→SEQ(RE)	24	LOD correction per quad (6 bits per quad)
FVTX	PA→SEQ(RE)	2	Provoking vertex for flat shading
PPTR0	PA→SEQ(RE)	11	P Store pointer for vertex 0
PPRT1	PA→SEQ(RE)	11	P Store pointer for vertex 1
PPTR2	PA→SEQ(RE)	11	P Store pointer for vertex 2
E_OFF_VECTOR	PA→SEQ(RE)	1	End of the vector
DEALLOC	PA→SEQ(RE)	1	Deallocation token for the P Store
STATE	PA→SEQ(RE)	21	State/constant pointer (6*3+3)
SPRITE	PA→SEQ(RE)	1	Need to generate tex cords
VALID	PA→SEQ(RE)	16	Valid bits for all pixels
NULL	PA→SEQ(RE)	1	Null Primitive (for deallocation purposes)
E_OFF_PRIM	PA→SEQ(RE)	1	End Of the primitive
FBFACE	PA→SEQ(RE)	1	Front face = 1, back face = 0
STIPPLE_LINE	PA→SEQ(RE)	1	Stippled line need to load tex cords from alternate buffer
RTRn	SEQ→PA	1	Stalls the PA in n clocks
RTS	PA→SEQ(RE)	1	PA ready to send data

Formatted: Bullets and Numbering

14.1.3 *PA/SC to RE : Vertex Bus*

Name	Direction	Bits	Description

14.1.4 PA/SC to SEQ : Vertex Control Bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
STATE	PA→SEQ	21	Render State (6*3+3 for constants)

14.1.5 CP to SEQ : Constant store load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Constant Address	CP→SEQ	8	Address of the block of 4 constants
Constant Data	CP→SEQ	512	Data sent over X clocks
Remap Address	CP→SEQ	10	Remaping address write address
Remap Data pointer	CP→SEQ	8	Remaping pointer

14.1.6 CP to SEQ : Texture State store load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Constant Address	CP→SEQ	8	Address of the block of 4 state constants
Constant Data	CP→SEQ	512	Data sent over X clocks
Remap Address	CP→SEQ	10	Remaping address write address
Remap Data pointer	CP→SEQ	8	Remaping pointer

14.1.7 CP to SEQ : Control State store load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description

14.1.8 MH to SEQ: Instruction store Load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Instruction address	MH→SEQ	12	Instruction address
Instruction	MH→SEQ	96	Instruction X times
Control Instruction address	MH→SEQ	8	Pointer to the instruction store
Control Instruction	MH→SEQ	32	Control Instruction X times

14.1.9 OB to RB : Pixel read from RBs

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Pixel Data	OB→RB	128	2 pixels (or ½ quad)
Quad Address	OB→RB	20	XY address 10 bits per

14.1.10 SP to PA/SC : Position return bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Position return	SP→PA	128	Position data or sprite size
Position Buffer pointer	SP→PA	7?	Pointer to the position cache
Parameter cache pointer	SP→PA	11	Pointer where the data will be in the parameter cache

~~12.1.3~~14.1.11 Shader Engine to Texture Unit Bus (Fast Bus)

Formatted: Bullets and Numbering

~~One~~ Four quad's worth of addresses is transferred to Texture 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.

~~One~~Four Quad's worth of Texture 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_Read_Register_Index	SEQ->SP	8	Index into Register files for reading Texture Address
Tex_RegFile_Read_Data	SP->TEX	~~512~~2048	~~4~~16 Texture Addresses read from the Register file
Tex_Write_Register_Index	SEQ->TEX	8	Index into Register file for write of returned Texture Data

~~12.1.4~~14.1.12 *Sequencer to Texture Unit bus (Slow Bus)*

Formatted: Bullets and Numbering

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

Name	Direction	Bits	Description
Tex_Ready	TEX→ SEQ	1	Data ready
Tex_Clause_Num	TEX→ SEQ	3	Clause number
Tex_cst	SEQ→TEX	~~?~~10	Texture ~~constants~~ ~~X~~state address 10 bits sent over 4 clocks
Tex_Inst	SEQ→TEX	~~?~~12	Texture fetch instruction ~~X~~address 12 bits sent over 4 clocks

~~12.1.5 *Shader Engine to RE/PA Bus*~~

Formatted: Bullets and Numbering

~~12.1.6 *PA? to sequencer*~~

~~13.~~15. Examples of program executions

~~13.1.1~~15.1.1 *Sequencer Control of a Vector of Vertices*

1. PA sends a vector of ~~16~~ 64 vertices (actually vertex indices – 32 bits/index for ~~512~~ 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 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 ~~512~~ 2048 bits/cycle)
   - the ~~16~~ 64 vertex indices are sent to the ~~16~~ 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 texture 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 texture clause 0 from the global instruction store
   - TSM0 was first selected by the TSM arbiter before it could start
7. all instructions of texture 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 Texture Unit to complete; it passes the register file write index for the texture 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 texture 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

Formatted: Bullets and Numbering

~~13.1.2~~15.1.2 *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.

Formatted: Bullets and Numbering

~~2.the RE's Parameter Buffer is loaded from the Parameter Cache before the SEQ takes control of the vector~~
- ~~after the HZ culling stage a request is made by the RE to send parameter data to the Parameter buffer~~
- ~~the Parameter buffer is wide enough to source 3 vertices worth of a particular parameter in one cycle~~
- ~~**at this moment the right sequencer will free up the parameter store locations not used anymore using the token provided by the PA**.~~

~~3.~~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 ~~one~~ four quad's worth of barycentrics per cycle

Formatted: Bullets and Numbering

~~4.~~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

~~5.~~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

~~6.SEQ control starts with the interpolation of parameters (up to 16 per thread) by sending the barycentric coordinates from the Pixel FIFO and the parameters from the Parameter Buffer to the interpolator~~

Formatted: Bullets and Numbering

- ~~P0i, P0j, and P0k (the value of P0 at each vertex) are loaded into the interpolator from the Parameter buffer~~
- ~~Q0 i, j, and k are loaded into the interpolator from the Pixel FIFO~~
- ~~The interpolator then generates the parameter value for each pixel in Q0 (Q0P0)~~
- ~~**P0i, P0j, and P0k are sent to the interpolator for Q1 only if Q1 is from a different primitive; if Q1 is from the same primitive as Q0, then the P0i, P0j, and P0k values loaded for Q0 are held by the interpolator and reused for Q1**~~
  - ~~**a "different_prim" control bit is passed with the barycentric data for each quad in the Pixel FIFO that indicates whether new parameter data needs to be loaded into the interpolator**~~
- ~~Q1 i, j, and k are then loaded into the interpolator from the Pixel FIFO~~
- ~~The interpolator then generates the parameter value for each pixel in Q1 (Q1P0)~~
- ~~Q2P0 and Q3P0 are generated in a similar manner~~
- ~~The next set of parameter data - P1i, P1j, and P1k - is then loaded into the interpolator~~
- ~~Q0 i, j, and k now must be re-read from the Pixel FIFO – this means that the output of the Pixel FIFO loops through the top four entries on each read command until at the end a final "block_pop" signal is asserted, causing the top four sets of barycentric coordinates to finally be removed~~
- ~~so the order of parameter info generated is Q0P0, Q1P0, Q2P0, Q3P0, Q0P1, Q1P1, etc.~~

~~7.~~5. SEQ controls the transfer of interpolated data to the SP register file over the RE_SP interface (which has a bandwidth of ~~512~~ 2048 bits/cycle). See interpolated data bus diagrams for details.

- ~~16 pixels worth of interpolated parameter data is sent to the 16 register files over 4 cycles~~
  - ~~RF0 of SU0, SU1, SU2, and SU3 is written with Q0P0 the first cycle~~
  - ~~RF1 of SU0, SU1, SU2, and SU3 is written with Q1P0 second cycle~~
  - ~~RF2 of SU0, SU1, SU2, and SU3 is written with Q2P0 third cycle~~
  - ~~RF3 of SU0, SU1, SU2, and SU3 is written with Q3P0 fourth cycle~~

~~8.~~6. SEQ constructs a control packet for the vector and sends it to the first reservation station (the FIFO in front of texture 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 (such as quad address for example) travels in a separate FIFO

~~9.~~7. TSM0 accepts the control packet and fetches the instructions for texture clause 0 from the global instruction store

Formatted: Bullets and Numbering

- TSM0 was first selected by the TSM arbiter before it could start

~~10.~~8. all instructions of texture clause 0 are issued by TSM0

Formatted: Bullets and Numbering

~~11.~~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 texture requests made to the Texture Unit to complete; it passes the register file write index for the texture 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

~~12.~~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

Formatted: Bullets and Numbering

~~13.~~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 texture state machine 1, or TSM1 FIFO)

~~14.~~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

~~15.~~13. after the shader program has completed, the SEQ will free up the GPRs so that they can be used by another shader program

Formatted: Bullets and Numbering

~~13.1.3~~15.1.3 *Notes*

Formatted: Bullets and Numbering

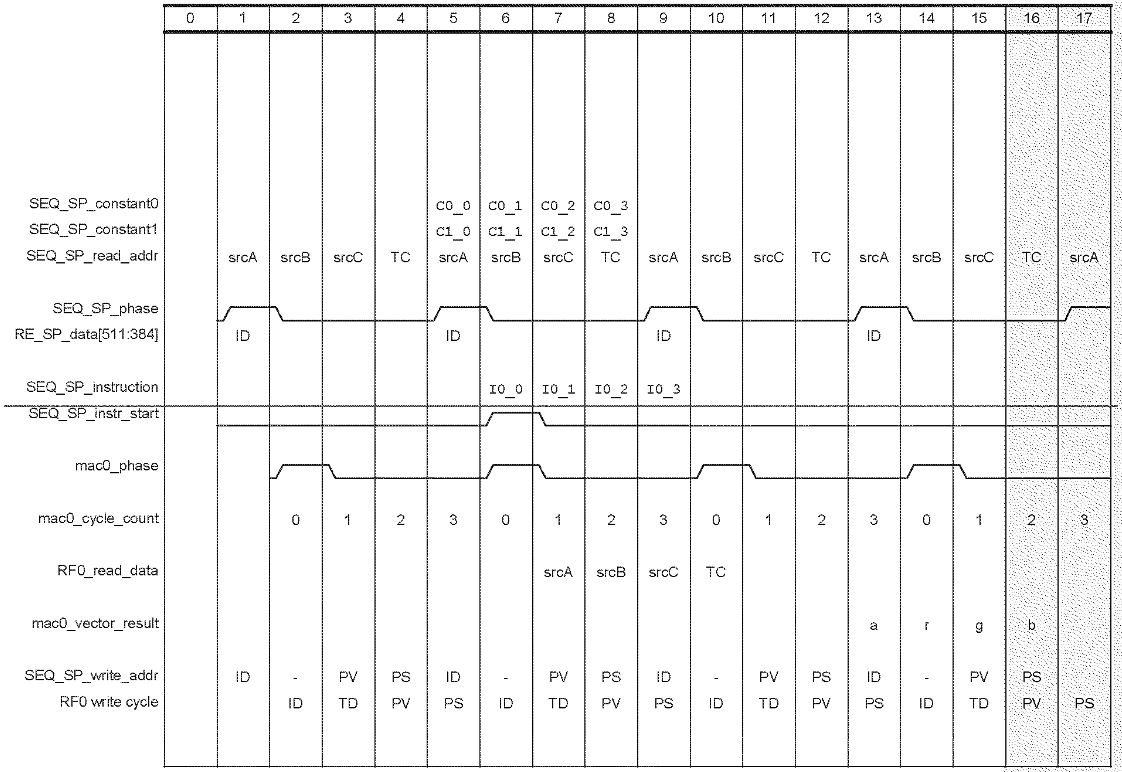
~~16.~~14. the state machines and arbitrers will operate ahead of time so that they will be able to immediately start the real threads or stall.

~~17.~~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, parameter buffer allocation, loops and branches and parameter cache de-allocation still needs to be specked out.

~~14. Timing Diagrams~~

~~14.1 MAC 0~~

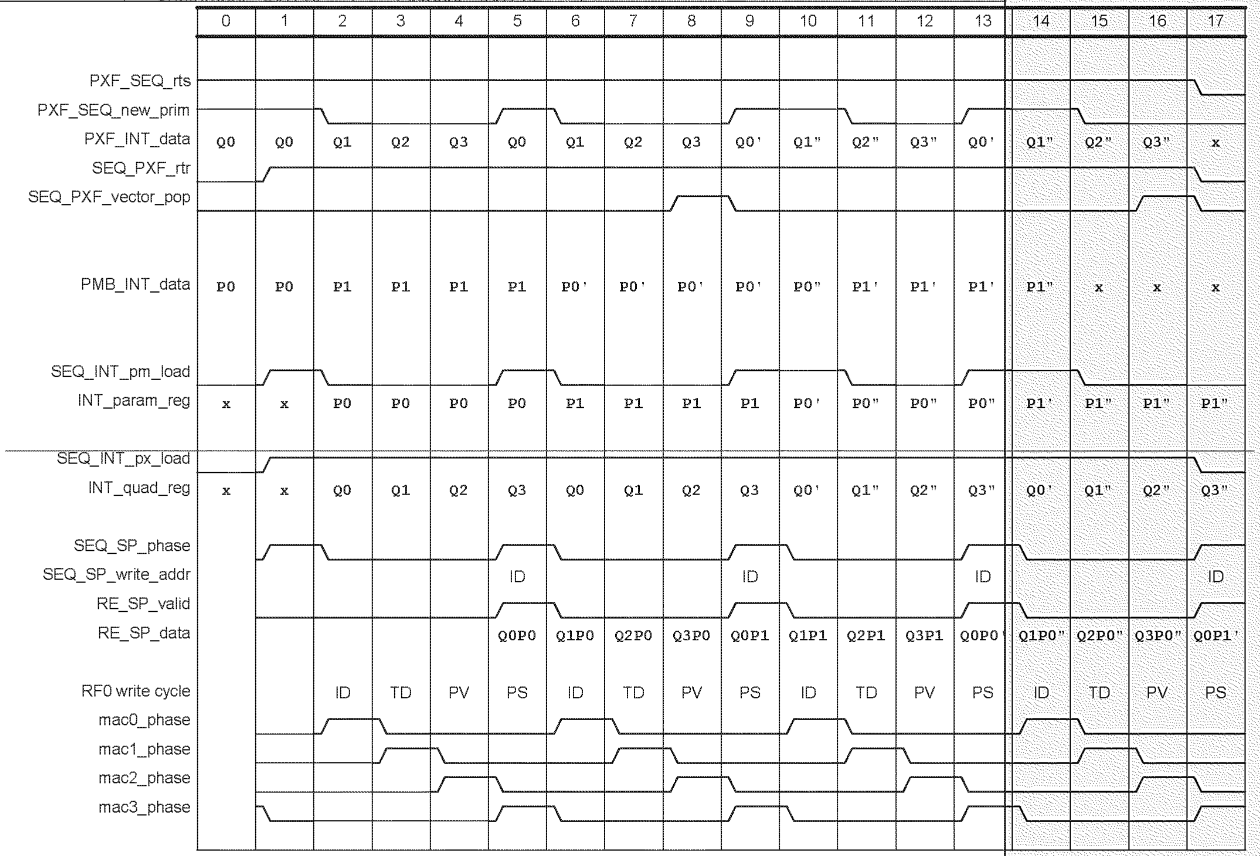


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
SEQ_SP_constant0						C0_0	C0_1	C0_2	C0_3									
SEQ_SP_constant1						C1_0	C1_1	C1_2	C1_3									
SEQ_SP_read_addr		srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA
SEQ_SP_phase																		
RE_SP_data[511:384]		ID				ID				ID				ID				
SEQ_SP_instruction							I0_0	I0_1	I0_2	I0_3								
SEQ_SP_instr_start																		
mac0_phase																		
mac0_cycle_count			0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
RF0_read_data								srcA	srcB	srcC	TC							
mac0_vector_result														a	r	g	b	
SEQ_SP_write_addr		ID	-	PV	PS	ID	-	PV	PS	ID	-	PV	PS	ID	-	PV	PS	
RF0 write cycle			ID	TD	PV	PS	ID	TD	PV	PS	ID	TD	PV	PS	ID	TD	PV	PS

Timing Diagram 1: Sequencer to Shader Pipe 0, Shader Unit 0, MAC 0

~~14.2 Sequencer to Shader Pipe~~

Formatted: Bullets and Numbering

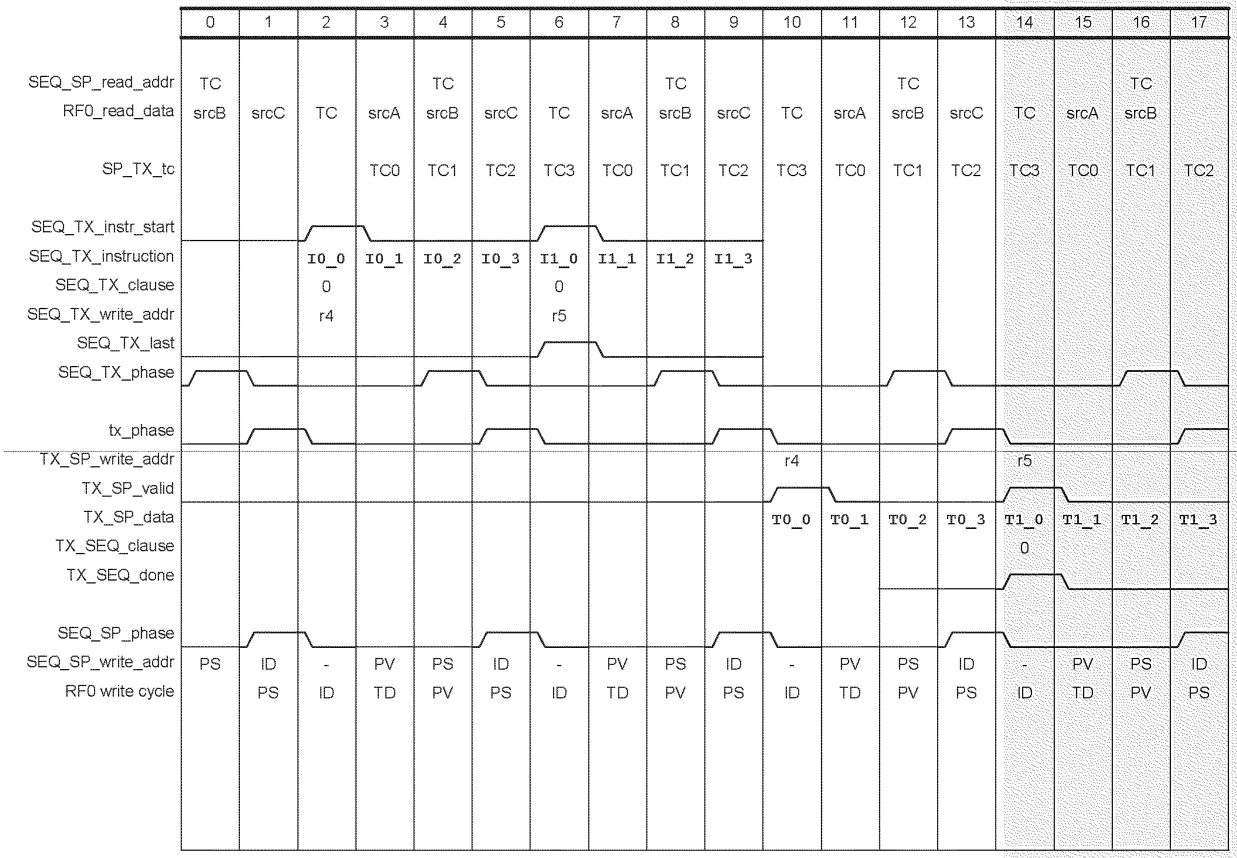


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
PXF_SEQ_rts
PXF_SEQ_new_prim
PXF_INT_data
Q0 Q0 Q1 Q2 Q3 Q0 Q1 Q2 Q3 Q0' Q1" Q2" Q3" Q0' Q1" Q2" Q3" x
SEQ_PXF_rtr
SEQ_PXF_vector_pop
PMB_INT_data
P0 P0 P1 P1 P1 P1 P0' P0' P0' P0' P0" P1' P1' P1' P1" x x x
SEQ_INT_pm_load
INT_param_reg
x x P0 P0 P0 P0 P1 P1 P1 P1 P0' P0" P0" P0" P1' P1" P1" P1"
SEQ_INT_px_load
INT_quad_reg
x x Q0 Q1 Q2 Q3 Q0 Q1 Q2 Q3 Q0' Q1" Q2" Q3" Q0' Q1" Q2" Q3"
SEQ_SP_phase
SEQ_SP_write_addr
ID ID ID ID
RE_SP_valid
RE_SP_data
Q0P0 Q1P0 Q2P0 Q3P0 Q0P1 Q1P1 Q2P1 Q3P1 Q0P0' Q1P0" Q2P0" Q3P0" Q0P1'
RF0 write cycle
ID TD PV PS ID TD PV PS ID TD PV PS ID TD PV PS
mac0_phase
mac1_phase
mac2_phase
mac3_phase

Timing Diagram 2: RE Interpolator to Shader Pipe Data Transfer

Formatted: Bullets and Numbering

~~14.3 Sequencer to Texture Pipe~~



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
SEQ_SP_read_addr	TC				TC				TC				TC				TC	
RF0_read_data	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	
SP_TX_tc				TC0	TC1	TC2	TC3	TC0	TC1	TC2	TC3	TC0	TC1	TC2	TC3	TC0	TC1	TC2
SEQ_TX_instr_start																		
SEQ_TX_instruction			I0_0	I0_1	I0_2	I0_3	I1_0	I1_1	I1_2	I1_3								
SEQ_TX_clause			0				0											
SEQ_TX_write_addr			r4				r5											
SEQ_TX_last																		
SEQ_TX_phase																		
tx_phase																		
TX_SP_write_addr											r4				r5			
TX_SP_valid																		
TX_SP_data											T0_0	T0_1	T0_2	T0_3	T1_0	T1_1	T1_2	T1_3
TX_SEQ_clause															0			
TX_SEQ_done																		
SEQ_SP_phase																		
SEQ_SP_write_addr	PS	ID	-	PV	PS	ID	-	PV	PS	ID	-	PV	PS	ID	-	PV	PS	ID
RF0 write cycle		PS	ID	TD	PV	PS	ID	TD	PV	PS	ID	TD	PV	PS	ID	TD	PV	PS

Timing Diagram 3: Sequencer - Texture Unit Interface and Texture Unit - Shader Pipe Data Transfer

~~14.4 Timing diagrams explanations~~

Formatted: Bullets and Numbering

~~The numbering of the four shader pipes, the four shader units, and the four MACs is from left to right and from 0 to 3. So for example the most significant 512 bits of a SP goes to SU0 and the least significant 512 bits go to SU3; within SU0, the most significant 128 bits go to MAC0 and the least significant 128 bits go to MAC3.~~
~~The following assumptions are made:~~

~~1.all block to block signals are register to register~~

~~2.for register file reads, the RF read data is available in the MAC one clock after a RF read address is registered into the MAC (this is the same as saying the read data is valid out of the RF two clocks after the address is asserted on the SEQ to SP interface)~~

Formatted: Bullets and Numbering

~~14.4.1 *Timing Diagram 1: Sequencer to Shader Pipe 0, Shader Unit 0, MAC 0*~~

~~This diagram shows the basics of the Sequencer to Shader Pipe interface. For simplicity only the timing relative to MAC0 is shown. The timing for MAC1 is one clock later than MAC0, MAC2 one clock later than MAC1, etc. This means that most of the signals need to be delayed in the SP by one cycle for MAC1, two cycles for MAC2, and three cycles for MAC3.~~
~~**SEQ_SP_constant0**: Constant 0 (128 bits over 4 cycles). Pipelined in SP for other MACs.~~
~~**SEQ_SP_constant1**: Constant 1 (128 bits over 4 cycles). Pipelined in SP for other MACs.~~
~~**SEQ_SP_read_addr**: Register File Read Address (8 bits). Pipelined in SP for other MACs.~~
~~**SEQ_SP_phase**: This signal syncs the data transfer to the RF from the RE, as well as defining the order of all writes into the RF. It is asserted during the cycle that interpolated data (ID) is valid on the RE_SP_ID bus. Pipelined in SP for other MACs.~~
~~**RE_SP_ID[511:384]**: This is the most significant 128 bits of the RE_SP_data interface (meaning that this MAC0 is in SU0).~~
~~**SEQ_SP_instruction**: 96 bits of instruction are sent over 4 cycles. Pipelined in SP for other MACs.~~
~~**SEQ_SP_instr_start**: control bit that signals the first cycle of the instruction transfer. Pipelined in SP for other MACs.~~
~~**mac0_phase**: registered version of SEQ_SP_phase used in MAC0 (this may not be he actual signal name).~~
~~**mac0_cycle_count**: a counter inside the MAC that keeps track of the RF write cycles; 0 here corresponds to the cycle RE interpolated data is written (this may not be he actual signal name).~~
~~**RF0_read_data**: data that is read out of MAC0's register file (this may not be he actual signal name).~~
~~**mac0_vector_result**: the 32-bit output of the vector ALU (PV is built up over 4 cycles) (this may not be he actual signal name).~~
~~**SEQ_SP_write_addr**: Register File Write Address (8 bits). Note that the SEQ does not send the Texture Data write address over this bus. Pipelined in SP for other MACs.~~
~~**RF0 write cycle**: the cycles allocated to the different write sources (ID = Interpolated Data, TD = Texture Data, PV = Previous Vector, PS = Previous Scalar) (not a signal – just a reference point on the diagram).~~

~~14.4.2 *Timing Diagram 2: RE Interpolator to Shader Pipe Data Transfer*~~

Formatted: Bullets and Numbering

~~This diagram shows how pixel data (barycentric coordinates i, j, and k) is sent from the Pixel FIFO to the interpolator under SEQ control, and how parameter data (for each vertex) is also sent to the interpolator under SEQ control. The output of the Interpolator is then shown being sent over the RE_SP interface.~~
~~**PXF_SEQ_rts**: Indicates that the output of the pixel FIFO is valid.~~
~~**PXF_SEQ_new_prim**: The current output of the Pixel FIFO is from a different primitive that the previous output. Tells the SEQ that new parameter info must be fetched (if its not from a new prim, then new parameter data is not needed).~~
~~**PXF_INT_data**: Data output of the Pixel FIFO – goes to the Interpolator.~~
~~**SEQ_PXF_rtr**: Indicates that the current Pixel FIFO output will be taken by the Interpolator (driven by SEQ). Then next quad of data will be driven the next cycle.~~
~~**SEQ_PXF_vector_pop**: SEQ tells the Pixel FIFO to pop a vector of pixels (otherwise RTRs cause the data to be cycled between the four quads).~~

~~**PMB_INT_data**: Data from the Parameter Buffer to the Interpolator. (Note that the control of the parameter buffer is TBD).~~

~~**SEQ_INT_pm_load**: controls the loading of parameter data into the Interpolator.~~

~~**INT_param_reg**: register in the Interpolator that holds the per-vertex parameter data while the per-pixel parameters are generated for one or more quads (may not be the actual signal name).~~
~~**SEQ_INT_px_load**: controls the loading of pixel data into the Interpolator.~~
~~**INT_quad_reg**: : register in the Interpolator that holds one quad's worth of pixel data(may not be the actual signal name).~~

~~**SEQ_SP_phase**: see above under TD1.~~
~~**SEQ_SP_write_addr**: see above under TD1.~~
~~**RE_SP_valid**: Interpolator Data Valid – indicates that the SP should write the ID on the appropriate cycle.~~
~~**RE_SP_data**: Data from the RE interpolator to the SP.~~
~~**RF0 write cycle**: see above under TD1.~~
~~**mac*_phase**: see above under TD1. These phase signals help to show the timing offset between the MACs. Note also that each Shader Unit has a set of these signals (all with the same timing).~~

~~14.4.3 *Timing Diagram 3: Sequencer - Texture Unit Interface and Texture Unit - Transfer*~~

Formatted: Bullets and Numbering

~~This diagram starts with the texture coordinate read from the register file and its transfer to the TX. The instruction transfer is then shown, followed by the texture data transfer to the shader pipe.~~
~~**SEQ_SP_read_addr**: see above. Here shows the cycle that the texture coordinate read address is asserted.~~
~~**RF0_read_addr**: see above.~~
~~**SP_TX_tc**: Texture coordinate data sent from the shader pipe to the texture unit.~~
~~**SEQ_TX_instr_start**: Asserted on the first cycle of a SEQ to TX instruction transfer.~~
~~**SEQ_TX_instruction**: 96 bits of texture instruction transferred over 4 cycles.~~
~~**SEQ_TX_clause**: the clause number associated with this instruction.~~
~~**SEQ_TX_write_addr**: RF write index used by TX for returned texture data.~~
~~**SEQ_TX_last**: indicates that this is the last texture instruction of a clause.~~
~~**SEQ_TX_phase**: syncs the texture data write. Note that it is asserted early enough to be registered into TX and still allow TX to source the texture data to the SP on the correct cycle.~~

~~**tx_phase**: the phase signal after being registered into TX.~~
~~**TX_SP_write_addr**: RF write index for texture data.~~
~~**TX_SP_valid**: indicates that valid texture data is being driven to the SP.~~
~~**TX_SP_data**: the texture data.~~
~~**TX_SEQ_clause**: the clause number associated with the texture data.~~
~~**TX_SEQ_done**: indicates to the SEQ that the texture data transfer is complete for the clause number that is on the TX_SEQ_clause bus.~~

~~**SEQ_SP_phase**: see above under TD1 - shown here for reference.~~
~~**SEQ_SP_write_addr**: see above under TD1- shown here for reference.~~
~~**RF0 write cycle**: see above under TD1- shown here for reference.~~

Formatted: Bullets and Numbering

~~15.~~16. 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 from the texture store to feed the ALUs. Two solutions exists for this problem:

1) Let the compiler handle the case and put those instructions in a texture clause so we can use the bandwith 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. 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?

Size of the fifo containing the information of a vector of pixels/vertices. And size of the fifos before the reservation stations.

Sequencer Instruction memory, and constant memory.

Arbitration policy for the output file.

Loops and branches.

The parameter cache may end up in the PA rather than in the RS. Parameter cache management thus may change.

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001~~24~~ ~~September, 200114~~	4 September, 2015~~5~~ ~~October, 200124~~	GEN-CXXXXX-REVA	1 of 33

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SEQ

Version 0.7~~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\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.

ATI 2013
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.~~4~~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.

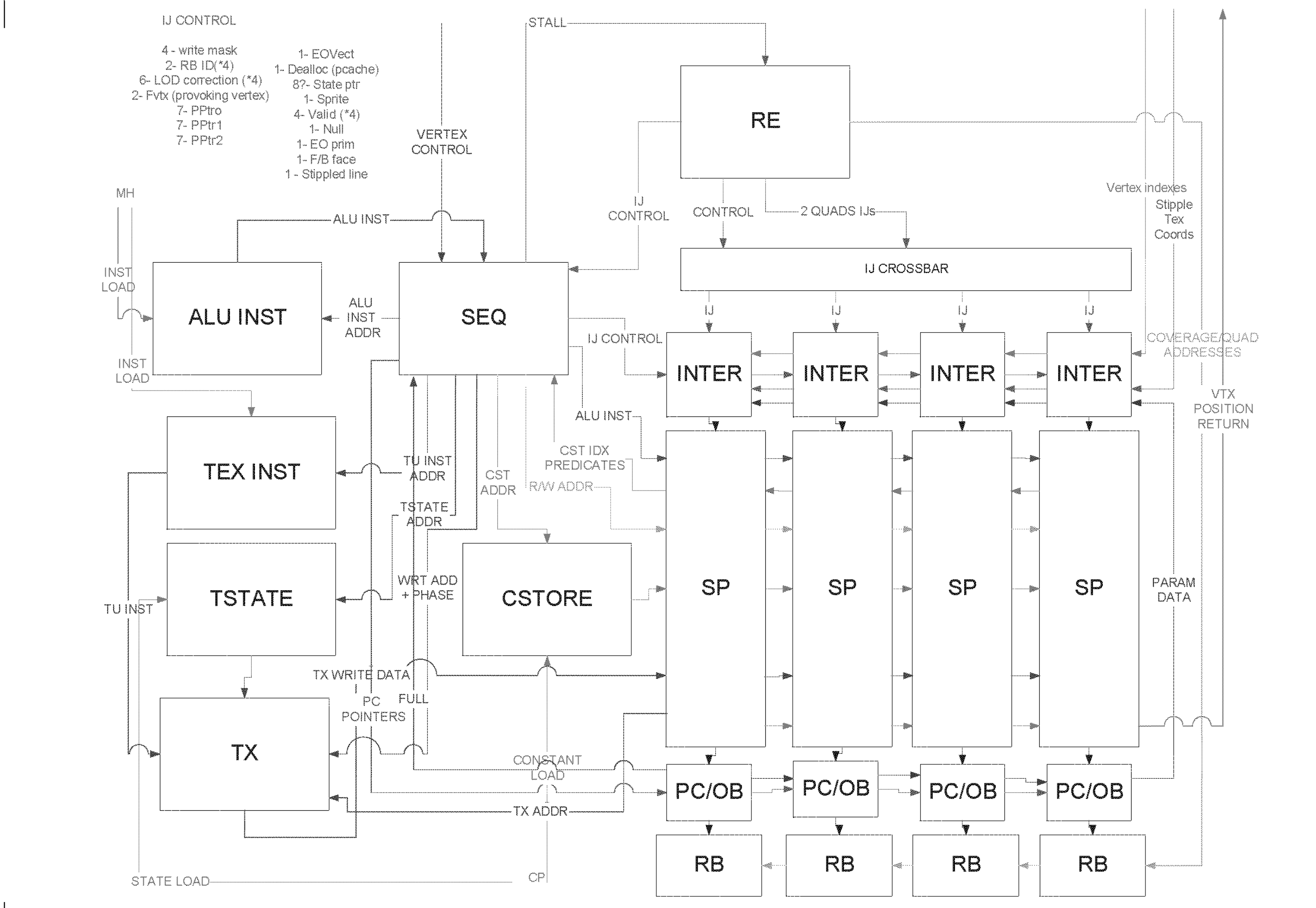
1. Overview

The sequencer first arbitrates between vectors of ~~16~~ 64 vertices that arrive directly from primitive assembly and vectors of ~~4~~ 16 quads (~~16~~ 64 pixels) that are generated in the raster engine.

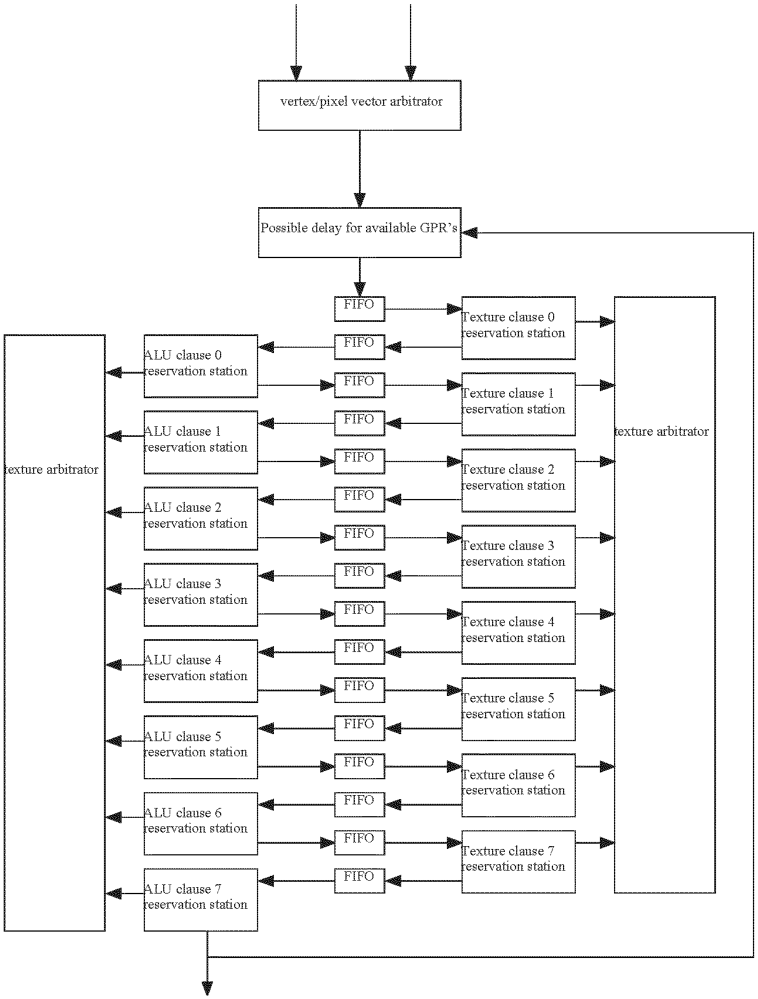
The vertex or pixel program specifies how many GPR's it needs to execute. The sequencer will not start the next vector until the needed space is available.

The sequencer is based on the R300 design. It chooses two ALU clauses and a texture 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 texture and eight ALU clauses, but clauses do not need to contain instructions. A vector of pixels or vertices ping-pongs along the sequencer FIFO, bouncing from texture reservation station to alu reservation station. A FIFO exists between each reservation stage, holding up vectors until the vector currently occupying a reservation station has left. A vector at a reservation station can be chosen to execute. The sequencer looks at all eight alu reservation stations to choose an alu clause to execute and all eight texture stations to choose a texture 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 texture fetches initiated by the previous texture 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 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
Vertex indexes
Stipple
Tex
Coords
MH
INST LOAD
ALU INST
ALU INST
ALU INST ADDR
SEQ
IJ CROSSBAR
IJ
IJ
IJ
IJ
IJ CONTROL
INTER
INTER
INTER
INTER
COVERAGE/QUAD ADDRESSES
INST LOAD
ALU INST
VTX POSITION RETURN
TEX INST
TU INST ADDR
CST ADDR
CST IDX PREDICATES
R/W ADDR
TSTATE ADDR
SP
SP
SP
SP
PARAM DATA
TU INST
TSTATE
WRT ADD + PHASE
CSTORE
TX WRITE DATA
PC POINTERS
FULL
TX
CONSTANT LOAD
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 arbitrator
ALU clause 0 reservation station
ALU clause 1 reservation station
ALU clause 2 reservation station
ALU clause 3 reservation station
ALU clause 4 reservation station
ALU clause 5 reservation station
ALU clause 6 reservation station
ALU clause 7 reservation station
Texture clause 0 reservation station
Texture clause 1 reservation station
Texture clause 2 reservation station
Texture clause 3 reservation station
Texture clause 4 reservation station
Texture clause 5 reservation station
Texture clause 6 reservation station
Texture clause 7 reservation station
texture arbitrator

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

The rasterizer always checks the vertices FIFO first and if allowed by the sequencer sends the data to the shader. If the vertex FIFO is empty then, the rasterizer takes the first entry of the pixel FIFO (a vector of ~~16~~ 64 pixels) and sends it to the interpolators. Then the sequencer takes control of the packet. The packet consists of ~~3~~ 21 bits of state, 6-7 bits for the base address of the Shader program and some information on the coverage to determine texture LOD. All other information (2x2 adresses) is put in a FIFO (one for the pixels and one for the vertices) and retrieved when the packet finishes its last clause.

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 texture machine issues a texure request and corresponding register address for the texture address (ta). A small command (tcmd) is passed to the texture system identifying the current level number (0) as well as the register write address for the texture return data. One texture request is sent every 4 clocks causing the texturing of ~~four~~ sixteen 2x2s worth of data (or ~~16~~ 64 vertices). Once all the requests are sent the packet is put in FIFO 1.

Upon recept of the return data, the texture unit writes the data to the register file using the write address that was provided by the level 0 texture machine and sends the clause number (0) to the level 0 texture state machine to signify that the write is done and thus the data is ready. Then, the level 0 texture 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.

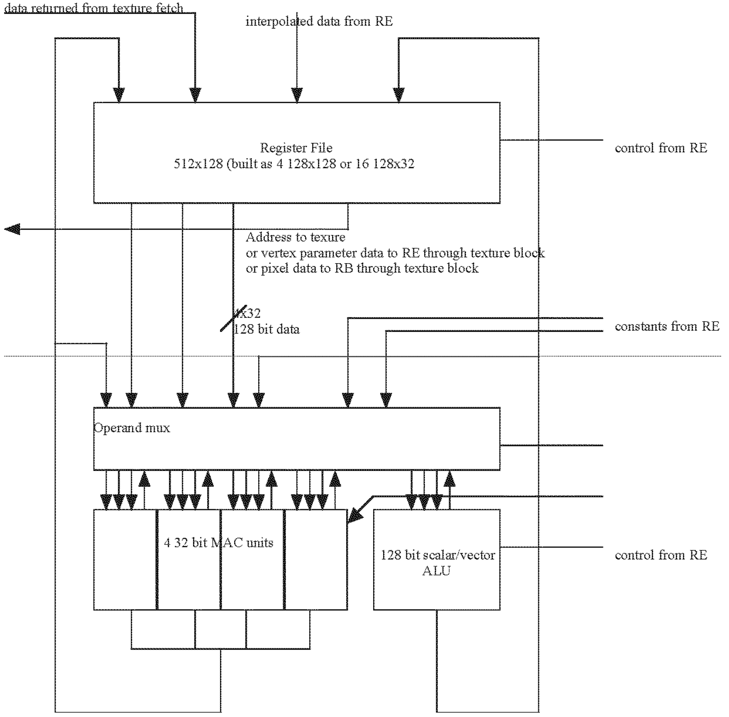
There will always be two active ALU clauses at any given time (and two arbitrers). One arbitrer will arbitrate over the odd clock cycles and the other one will arbitrate over the even clock cycles. The only constraints between the two arbitrers is that they are not allowed to pick the same clause number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

If the packet is a vertex packet, upon reaching ALU clause 4, it can export the position if the position is ready. So the arbitrer must prevent ALU clause 4 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).

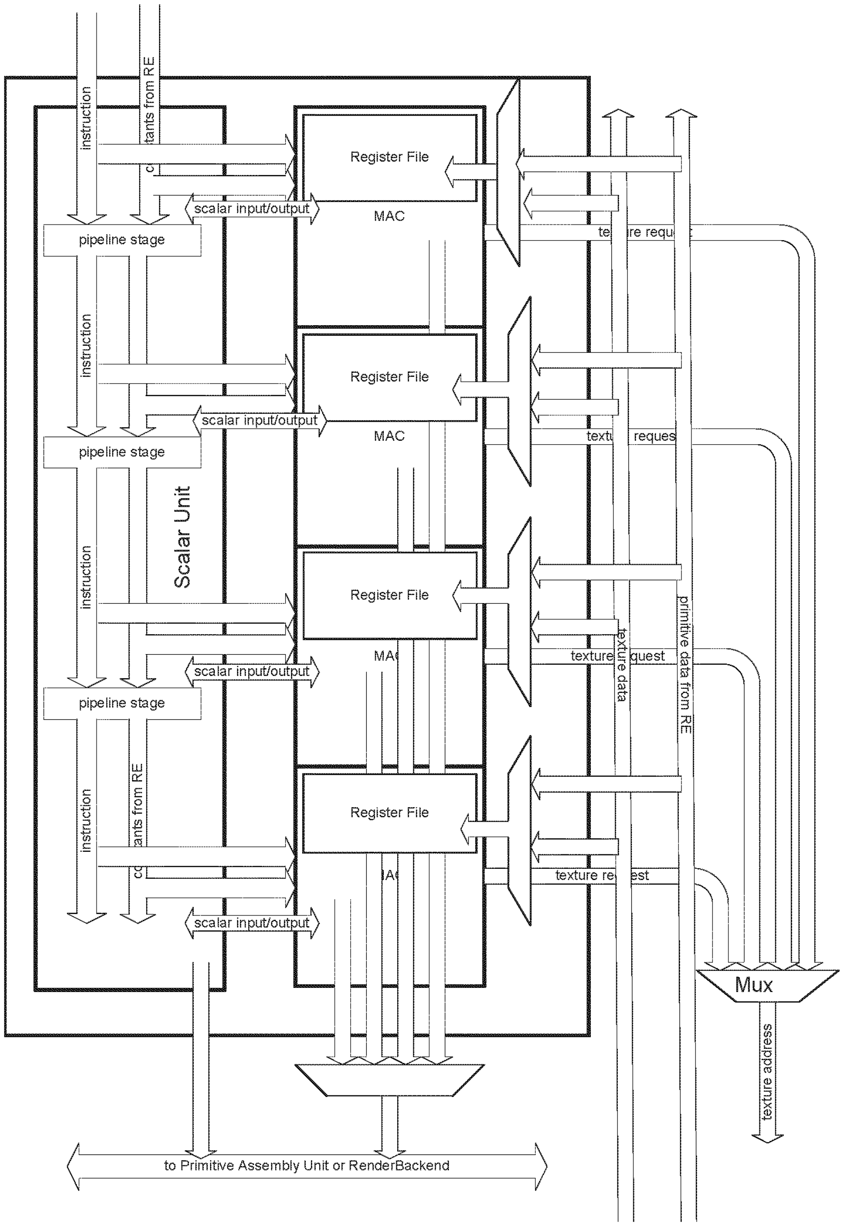
All other level process in the same way until the packet finally reaches the last ALU machine (8). On completion of the level 8 ALU clause, a valid bit is sent to the Render Backend which picks up the color data. This requires that the last instruction writes to the output register – a condition that 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.

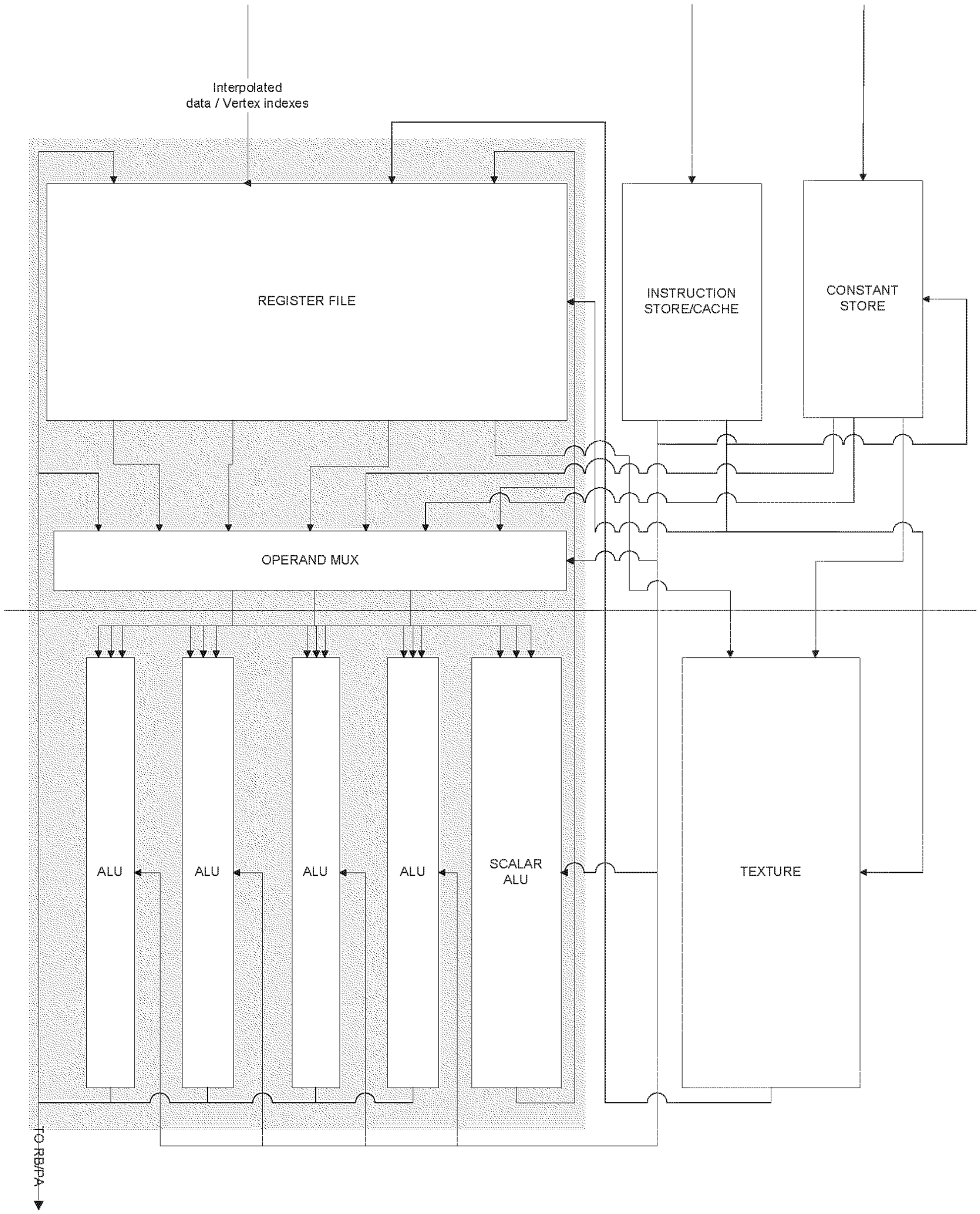
Only two ALU state machine may have access to the register file address bus or the instruction decode bus at one time. Similarly, only one texture state machine may have access to the register file address bus at one time. Arbitration is performed by three arbitrer blocks (two for the ALU state machines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

~~Each state machine maintains an address pointer specifying where the 16 entries vector is located in the register file (the texture machine has two pointers one for the read address and one for the write). Upon completion of its job, the address pointer is incremented by a predefined amount equal to the total number of registers required by the shading code. A comparison of the address pointer for the first state machine in the chain (the input state machine), and the last machine in the chain (the level 8 ALU machine), gives an indication of how much unallocated register file memory is available~~


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

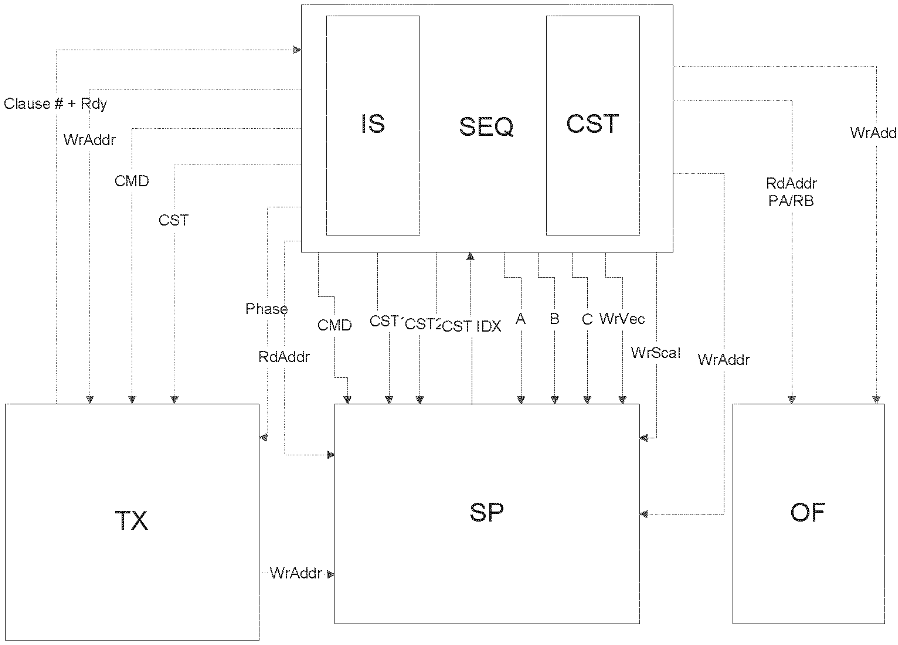
1.2 Data Flow graph





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

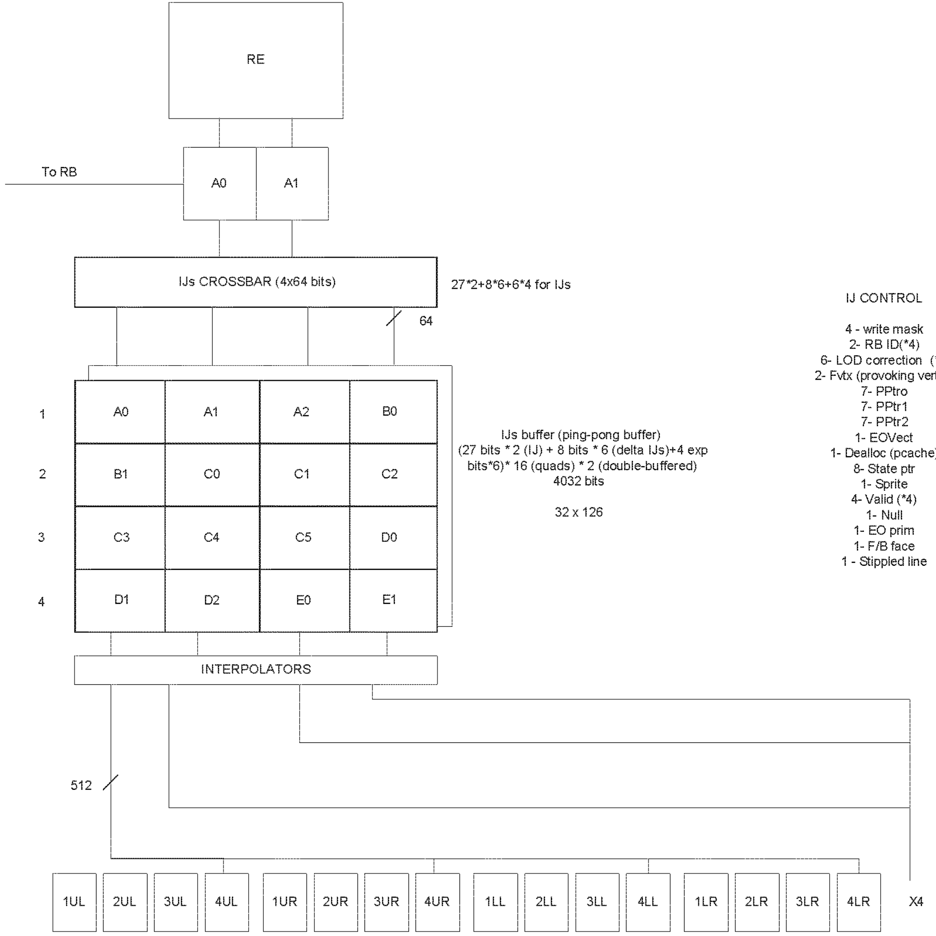
1.3 Control Graph



In green is represented the Texture 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.



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

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19
SP0	A0	B1	C3	D1					A0	B1	C3	D1					V 0-3	V 16-19	V 32-35	V 48-51
SP1	A1		C4	D2		C0			A1		C4	D2		C0			V 4-7	V 20-23	V 36-39	V 52-55
SP2	A2		C5			C1		E0	A2		C5			C1		E0	V 8-11	V 24-27	V 40-43	V 56-59
SP3					B0	C2	D0	E1					B0	C2	D0	E1	V 12-15	V 28-31	V 44-47	V 60-63
				P0									P1							

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.~~Since each of the register file is actually physically divided (one 128x128 per MAC) and we don't have the place to hold a maximum size vector of vertices in the parameter buffer, we need to interpolate on a parameter basis rather than on a quad basis. So the order to the register file will be:~~

~~Q0P0 Q1P0 Q2P0 Q3P0 Q0P1 Q1P1 Q2P1 Q3P1 Q0P2 Q1P2 ...~~

3. Instruction Store

There is going to be only one instruction store for the whole chip. It ~~may~~ will contain ~~up to 200~~4096~~0~~ 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 ~~24~~ 96*2 bits/ 4 clocks (48 bits/clock)~~/pipe~~. ~~To achieve this this instruction store is likely to be broken up into 4 blocks. An ALU instruction section (1R/1W) split in two and a texture section (1R/1W) also split in two. The bandwith out of those memories is 96 bits/clock.~~It is likely to be a 1R/1W port memory; we use 2 clocks to load the ALU instruction and 2 clocks to load the Texture instruction.

Formatted: Bullets and Numbering

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).

Formatted: Bullets and Numbering

~~4.~~5. Constant Store

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 bits. The constant store is also planned to be shared. The read BW from the constant store is 128~~512/4~~ bits/clock~~/pipe~~ 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.

Formatted: Bullets and Numbering

~~5.~~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. The control program has 4(5) instructions:

Formatted: Bullets and Numbering

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[31:0]
Loop_count[7:0][15:0]
Loop_Start[7:0] [15:0]
Loop_End[7:0] [15:0]

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

6.2 The Control Flow Program

Formatted: Bullets and Numbering

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 four basic instructions:

Execute
Conditional_execute (Conditional Execute Predicates)
Loop_start
Loop_end

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.
Loop_end increments (decrements?) the loop counter and jumps back the specified number of instructions if the loop end condition is not met.

if we try and fit the control flow instructions into 32 bit words, the following instructions are possible choices:

Execute																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	instruction_count												Reserved			Address												

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

Conditional Execute																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	Boolean						0: = 0 1: = 1 2,3:NA		Instruction_count							Address												

if the specified boolean (6 bits can address 64 booleans) meets the specified condition then execute the specified instructions (up to 64 instructions)

Conditional Execute Predicates																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	Reserved =0						0: = 0 1: = 1 2,3:NA		Instruction_count							Address												

Check the OR of all current predicate bits. If OR matches the condition execute the specified number of instructions.

Loop_Start																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0																								Loop ID				

Initialize the specified loop

Loop_End																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	Reserved (must = 0)												Start Address								Reserved (must = 0)			Loop ID				

If the loop condition of the current loop is not met, then branch back to the specified address in the control flow program. Note that jumping back to the loop_start results in an infinite loop, the jump should be to loop_start+1.

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

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
```

The control program can be up to 256 instructions in size. (There is an offset added to the address from the render state before accessing the control flow program memory to allow for multiple programs resident at the same time)

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 addresses from the control program are added to another offset to allow for multiple programs resident at the same time.

Under this model, all subroutine calls must be inlined into the control program.

6.3 Data dependant predicate instructions

Formatted: Bullets and Numbering

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

PRED_SETE - similar to SETE except that the result is 'exported' to the sequencer.
PRED_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 the 64 bit predicate vector and use it to control the write masking (two sets for interleaved operation). This predicate is not maintained across clause boundaries.

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

Formatted: Bullets and Numbering

6.4 Register file indexing

Because we can have loops in texture clause, we need to be able to index into the register file in order to retrieve the data created in a texture 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 :

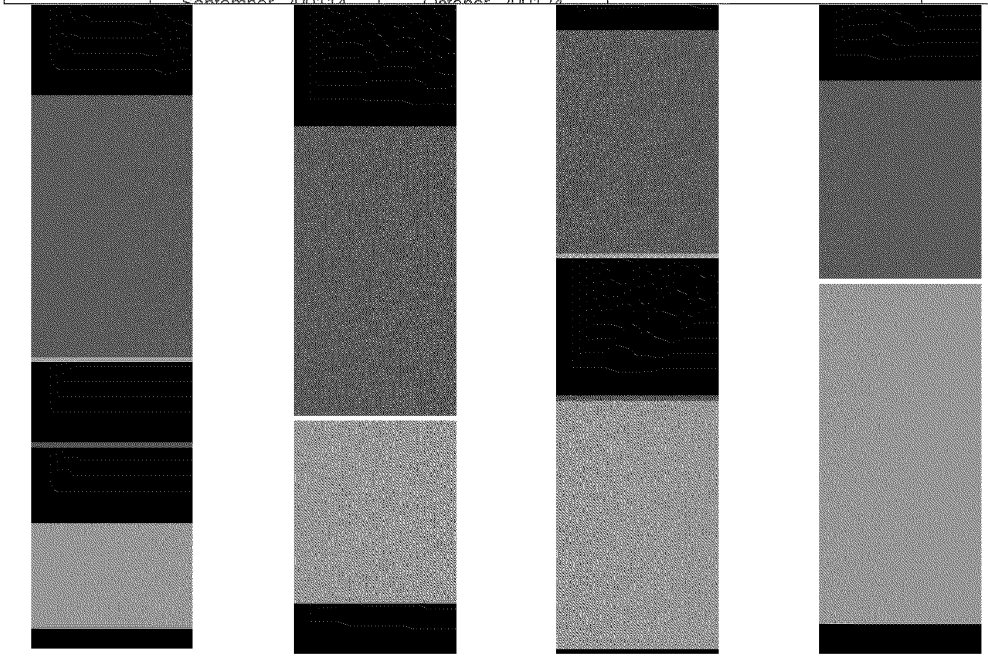
Bit7	Bit 6	
0	0	'absolute register'
0	1	'relative register'
1	0	'previous vector'
1	1	'previous scalar'

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_counter and this becomes our new address that we give to the shader pipe.~~However, it is still unclear if we plan on supporting data dependent branches or not.~~

Formatted: Bullets and Numbering

~~6.~~7. 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 ~~as~~ is allowed to movi~~ng~~e again.

Formatted: Bullets and Numbering

~~7.~~8. Texture Arbitration

The texture arbitration logic chooses one of the 8 potentially pending texture 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 texture fetch per clock (or 4 fetches in one clock every 4 clocks) until all the texture fetch instructions of the clause are sent. This means that there cannot be any dependencies between two texture fetches of the same clause.

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

Formatted: Bullets and Numbering

~~8.~~9. ALU Arbitration

ALU arbitration proceeds in almost the same way than texture 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):

Einst0 Oinst0 Einst1 Oinst1 Einst2 Oinst2 Einst0 Oinst3 Einst1 Oinst4 Einst2 Oinst0...
Proceeding this way hides the latency of 8 clocks of the ALUs.

Formatted: Bullets and Numbering

~~9.~~10. 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?). The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU arbitrer will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

Formatted: Bullets and Numbering

~~10.~~11. Content of the reservation station FIFOs

3 bits of Render State 6-7 bits for the base address of the instruction store, ~~and~~ some bits for LOD correction and coverage mask information in order to fetch texture for only valid pixels. Every other information (such as the coverage mask, quad address, etc.) is put in a FIFO and is retrieved when the quad exits the shader pipe to enter in the output file buffer. 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.

Formatted: Bullets and Numbering

~~11.~~12. 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).

Formatted: Bullets and Numbering

13. 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 24x24 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 full 24x24 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 24x24 yielding 8 bits (IJs): 6
Subtracts 24x24 (Parameters): 2
Adds: 8

FORMAT OF P0's IJ : Mantissa 23 Exp 4 for I
Mantissa 23 Exp 4 for J

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

Total number of bits : 23*2 + 8*6 + 4*8 = 126 (rounded up on the bus to 128)

Formatted: Bullets and Numbering

14. 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.~~(RB FIFO and Parameter Cache)~~

Formatted: Bullets and Numbering

15. ~~The output file is where program results are exported when the pixel/vertex shader finishes. It constists of a 512x128 memory cell that is statically divided between pixels and vertices. The output file has 1 write port and 1 read port. The sequencer is responsible for managing the addresses of this output file and for stalling the shader pipe should this output file fill up. The management is done by keeping the tail and head pointers of each sections (pixels and vertices) and incrementing them using a simple RoundRobin allocation policy. The sequencer must also arbitrate between the PA and the RB for the use of the read port. This arbitration will either be priority based or just interleaved evenly (1 read every 2 clocks for each of the blocks).~~Vertex position exporting

On clause 4 (or 5) the vertex shader can export to the PA both the vertex position and the point sprite. It can also do so at clause 8 if not done at clause 4. The export is done by putting the exported position back into the GPRs. Then using the texture port in an opportunistic manner, 16 positions are put into a FIFO (16x128) in order (left to right). This fifo drains 128 bits per clock to the PA and once empty is filled up again with sprite sizes (if any). The process is repeated 4 times. The sequencer must make sure that the program doesn't enter ALU clause 5 (it can enter texture clause 5) because the registers can be reused at this point. The sequencer must also make sure not to dealocate an exporting program before it is done exporting data. Along with the position is exported a pointer to the parameter cache where the data will be once the vertex shader exports.

Formatted: Bullets and Numbering

16. Registers

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)
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
PCNTLSHADER	base pointer for the pixel control program
VCNTLSHADER	base pointer for the vertex control program
VWRAP	wrap point for the vertex shader instruction store
PWRAP	wrap point for the pixel shader instruction store
REG_ALLOC_PIX	number of registers to allocate for pixel shader programs
REG_ALLOC_VERT	number of registers to allocate for vertex shader programs
PARAM_MASK[0…16]	parameter mask to specify wich parameters the pixel shader
FLAT_GOUR[0…16]	wich parameters are to be gouraud shaded
GEN_TEX[0….16]	for wich parameters do we need to generate tex coords.
CYL_WRAP[0…64]	for wich vertices do we do the cyl wrapping.
P_EXPORT	number of exports for pixel shader
V_EXPORT	number of exports for vertex shader (also the number of interpolated parameters for pixel shaders)
V_EXPORT_LOC	Vertex shader exporting to RB or the PCACHE

Formatted: Bullets and Numbering

~~12.~~17. Interfaces

~~12.1~~17.1 External Interfaces

~~12.1.1~~17.1.1 ~~*Sequencer to Shader Engine Bus*~~*PA/SC to RE : IJ bus*

~~This is a bus that sends the instruction and constant data to all 4 Sub-Engines of the Shader. Because a new instruction is needed only every 4 clocks, the width of the bus is divided by 4 and both constants and instruction are sent over those 4 clocks.~~ 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

Name	Direction	Bits	Description
~~Instruction Start~~IJs	~~SEQ-> SP~~PA→RE	~~1~~64	~~High on first cycle of transfer~~IJ information sent over 2 clocks

Formatted: Bullets and Numbering

~~12.1.2~~

~~*12.1.2Shader Engine to Output File*~~

~~Every clock each Sub-Engine can output 128 bits of 'vector' data and 32 bits of 'scalar' data to an output file (?). This data will be compressed into 128 bits total prior to storage in output file.~~

Formatted: Bullets and Numbering

17.1.2 *PA/SC to SEQ : 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.

Name	Direction	Bits	Description
Write Mask	PA→SEQ(RE)	4	Quad Write mask left to right
RB_ID	PA→SEQ(RE)	8	RB id for each quad sent 2 bits per quad
LOD_CORRECT	PA→SEQ(RE)	24	LOD correction per quad (6 bits per quad)
FVTX	PA→SEQ(RE)	2	Provoking vertex for flat shading
PPTR0	PA→SEQ(RE)	11	P Store pointer for vertex 0
PPRT1	PA→SEQ(RE)	11	P Store pointer for vertex 1
PPTR2	PA→SEQ(RE)	11	P Store pointer for vertex 2
E_OFF_VECTOR	PA→SEQ(RE)	1	End of the vector

DEALLOC	PA→SEQ(RE)	1	Deallocation token for the P Store
STATE	PA→SEQ(RE)	21	State/constant pointer (6*3+3)
SPRITE	PA→SEQ(RE)	1	Need to generate tex cords
VALID	PA→SEQ(RE)	16	Valid bits for all pixels
NULL	PA→SEQ(RE)	1	Null Primitive (for PC deallocation purposes)
E_OFF_PRIM	PA→SEQ(RE)	1	End Of the primitive
FBFACE	PA→SEQ(RE)	1	Front face = 1, back face = 0
STIPPLE_LINE	PA→SEQ(RE)	1	Stippled line need to load tex cords from alternate buffer
RTRn	SEQ→PA	1	Stalls the PA in n clocks
RTS	PA→SEQ(RE)	1	PA ready to send data
QuadX	PA→SEQ(RE)	40	Quad X address 10 bits per quad
QuadY	PA→SEQ(RE)	40	Quad Y address 10 bits per quad

17.1.3 *PA/SC to RE : Vertex Bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Vertex indexes	PA→RE	32	Pointers of indexes

17.1.4 *PA/SC to SEQ : Vertex Control Bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
STATE	PA→SEQ	21	Render State (6*3+3 for constants)
Position Cache Pointer	PA→SEQ	7	Pointer to the position cache
Write Mask	PA→SEQ	64?	Which vertices are valid
E_OFF_VECTOR	PA→SEQ	1	End of the vector

17.1.5 *CP to SEQ : Constant store load*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Constant Address	CP→SEQ	8	Address of the block of 4 constants
Constant Data	CP→SEQ	512	Data sent over 4 clocks
Remap Address	CP→SEQ	10	Remaping address write address
Remap Data pointer	CP→SEQ	8	Remaping pointer

17.1.6 *CP to SEQ : Texture State store load*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Constant Address	CP→SEQ	8	Address of the block of 4 state constants
Constant Data	CP→SEQ	512	Data sent over 4 clocks
Remap Address	CP→SEQ	10	Remaping address write address
Remap Data pointer	CP→SEQ	8	Remaping pointer

17.1.7 *CP to SEQ : Control State store load*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description

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

17.1.8 *MH to SEQ: Instruction store Load*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Instruction address	MH→SEQ	12	Instruction address
Instruction	MH→SEQ	96	Instruction X times
Control Instruction address	MH→SEQ	9	Pointer to the control instruction store
Control Instruction	MH→SEQ	32	Control Instruction X times

17.1.9 SP to RB : Pixel read from RBs

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Pixel Data	SP→RB	256	2 pixels (or ½ quad)
Quad Address	SP→RB	20	XY address 10 bits per

Only one exporting clause (7) can be selected at any given time.

17.1.10 SP to PA/SC : Position return bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Position return	SP→PA	128	Position data or sprite size
Position Buffer pointer	SP→PA	7	Pointer to the position cache
Parameter cache pointer	SP→PA	11	Pointer where the data will be in the parameter cache

For point sprites and position exports the size and position are interleaved on a 16 x 16 basis. We export 16 positions then 16 point sprite sizes. The registers are taken until the next ALU clause where they are going to be available again. Thus the sequencer has to make sure that we finished exporting data before allowing the program in the next ALU clause.

~~12.1.3~~17.1.11 Shader Engine to Texture Unit Bus (Fast Bus)

Formatted: Bullets and Numbering

~~One~~ Four quad's worth of addresses is transferred to Texture 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.

~~One~~Four Quad's worth of Texture 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_Read_Register_Index	SEQ->SP	~~8~~7	Index into Register files for reading Texture Address
Tex_RegFile_Read_Data	SP->TEX	~~512~~2048	~~4~~ 16 Texture Addresses read from the Register file
Tex_Write_Register_Index	SEQ->TEX	~~8~~7	Index into Register file for write of returned Texture Data

~~12.1.4~~17.1.12 Sequencer to Texture Unit bus (Slow Bus)

Formatted: Bullets and Numbering

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

Name	Direction	Bits	Description
Tex_Ready	TEX→ SEQ	1	Data ready
Tex_Clause_Num	TEX→ SEQ	3	Clause number
Tex_cst	SEQ→TEX	~~?~~10	Texture ~~constants~~ ~~X~~state address 10 bits sent over 4 clocks
Tex_Inst	SEQ→TEX	~~?~~12	Texture fetch instruction ~~X~~address 12 bits sent over 4 clocks
EO_CLAUSE	SEQ→TEX	1	Last instruction of the clause
PHASE	SEQ→TEX	1	Write phase signal

Formatted: Bullets and Numbering

18. Internal interfaces

~~12.1.5 Shader Engine to RE/PA Bus~~

~~12.1.6 PA? to sequencer~~

~~13.~~19. Examples of program executions

~~13.1.1~~19.1.1 *Sequencer Control of a Vector of Vertices*

1. PA sends a vector of ~~16~~ 64 vertices (actually vertex indices – 32 bits/index for ~~512~~ 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 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 ~~512~~ 2048 bits/cycle)
   - the ~~16~~ 64 vertex indices are sent to the ~~16~~ 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 texture 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 texture clause 0 from the global instruction store
   - TSM0 was first selected by the TSM arbiter before it could start
7. all instructions of texture 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 Texture Unit to complete; it passes the register file write index for the texture 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 texture 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

~~13.1.2~~19.1.2 *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 Parameter Buffer is loaded from the Parameter Cache before the SEQ takes control of the vector~~

Formatted: Bullets and Numbering

- ~~after the HZ culling stage a request is made by the RE to send parameter data to the Parameter buffer~~
- ~~the Parameter buffer is wide enough to source 3 vertices worth of a particular parameter in one cycle~~
- **~~at this moment the right sequencer will free up the parameter store locations not used anymore using the token provided by the PA.~~**

~~3.~~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 ~~one~~ four quad's worth of barycentrics per cycle

~~4.~~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

Formatted: Bullets and Numbering

~~5.~~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

~~6.SEQ control starts with the interpolation of parameters (up to 16 per thread) by sending the barycentric coordinates from the Pixel FIFO and the parameters from the Parameter Buffer to the interpolator~~

Formatted: Bullets and Numbering

- ~~P0i, P0j, and P0k (the value of P0 at each vertex) are loaded into the interpolator from the Parameter buffer~~
- ~~Q0 i, j, and k are loaded into the interpolator from the Pixel FIFO~~
- ~~The interpolator then generates the parameter value for each pixel in Q0 (Q0P0)~~
- **~~P0i, P0j, and P0k are sent to the interpolator for Q1 only if Q1 is from a different primitive; if Q1 is from the same primitive as Q0, then the P0i, P0j, and P0k values loaded for Q0 are held by the interpolator and reused for Q1~~**
    - **~~a "different_prim" control bit is passed with the barycentric data for each quad in the Pixel FIFO that indicates whether new parameter data needs to be loaded into the interpolator~~**
- ~~Q1 i, j, and k are then loaded into the interpolator from the Pixel FIFO~~
- ~~The interpolator then generates the parameter value for each pixel in Q1 (Q1P0)~~
- ~~Q2P0 and Q3P0 are generated in a similar manner~~
- ~~The next set of parameter data - P1i, P1j, and P1k - is then loaded into the interpolator~~

- ~~Q0 i, j, and k now must be re-read from the Pixel FIFO – this means that the output of the Pixel FIFO loops through the top four entries on each read command until at the end a final "block_pop" signal is asserted, causing the top four sets of barycentric coordinates to finally be removed~~
- ~~so the order of parameter info generated is Q0P0, Q1P0, Q2P0, Q3P0, Q0P1, Q1P1, etc.~~

~~7.~~5. SEQ controls the transfer of interpolated data to the SP register file over the RE_SP interface (which has a bandwidth of ~~512~~ 2048 bits/cycle). See interpolated data bus diagrams for details.

- ~~16 pixels worth of interpolated parameter data is sent to the 16 register files over 4 cycles~~
  - ~~RF0 of SU0, SU1, SU2, and SU3 is written with Q0P0 the first cycle~~
  - ~~RF1 of SU0, SU1, SU2, and SU3 is written with Q1P0 second cycle~~
  - ~~RF2 of SU0, SU1, SU2, and SU3 is written with Q2P0 third cycle~~
  - ~~RF3 of SU0, SU1, SU2, and SU3 is written with Q3P0 fourth cycle~~

~~8.~~6. SEQ constructs a control packet for the vector and sends it to the first reservation station (the FIFO in front of texture 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 (such as quad address for example) travels in a separate FIFO

~~9.~~7. TSM0 accepts the control packet and fetches the instructions for texture clause 0 from the global instruction store

Formatted: Bullets and Numbering

- TSM0 was first selected by the TSM arbiter before it could start

~~10.~~8. all instructions of texture clause 0 are issued by TSM0

Formatted: Bullets and Numbering

~~11.~~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 texture requests made to the Texture Unit to complete; it passes the register file write index for the texture 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

~~12.~~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

Formatted: Bullets and Numbering

~~13.~~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 texture state machine 1, or TSM1 FIFO)

~~14.~~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

~~15.~~13. after the shader program has completed, the SEQ will free up the GPRs so that they can be used by another shader program

Formatted: Bullets and Numbering

~~13.1.3~~19.1.3 *Notes*

Formatted: Bullets and Numbering

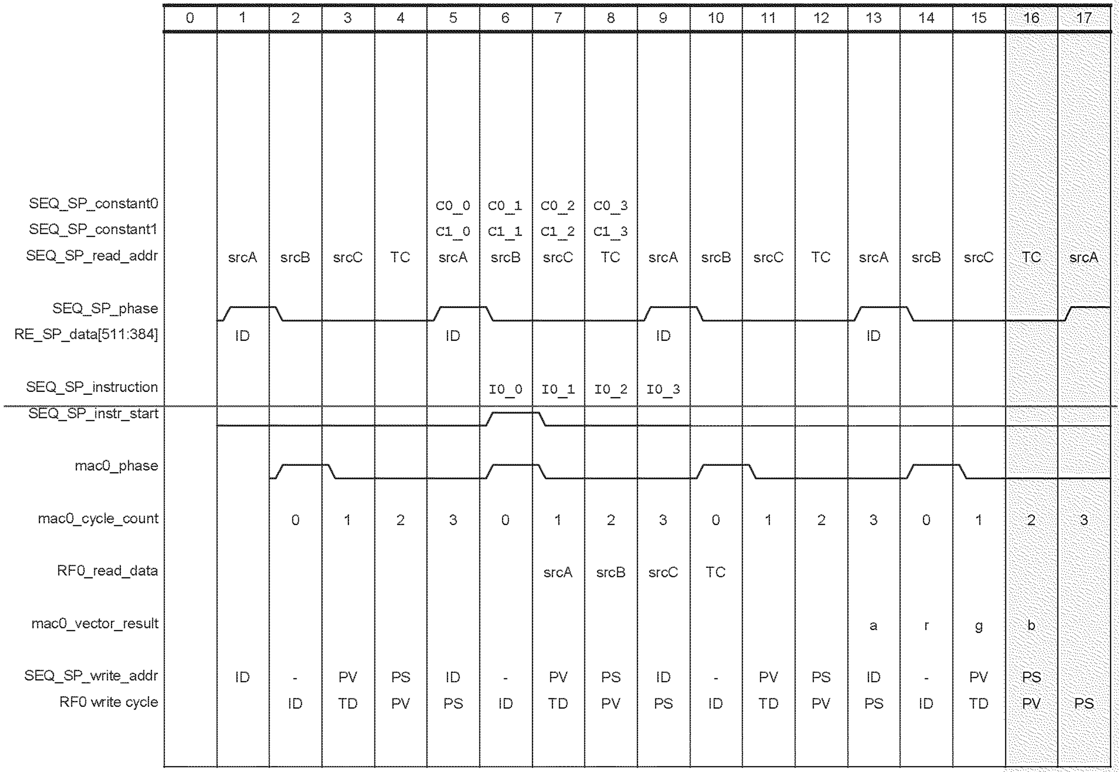
~~16.~~14. the state machines and arbitrers will operate ahead of time so that they will be able to immediately start the real threads or stall.

~~17.~~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, parameter buffer allocation, loops and branches and parameter cache de-allocation still needs to be specked out.

~~14. Timing Diagrams~~

~~14.1 MAC 0~~

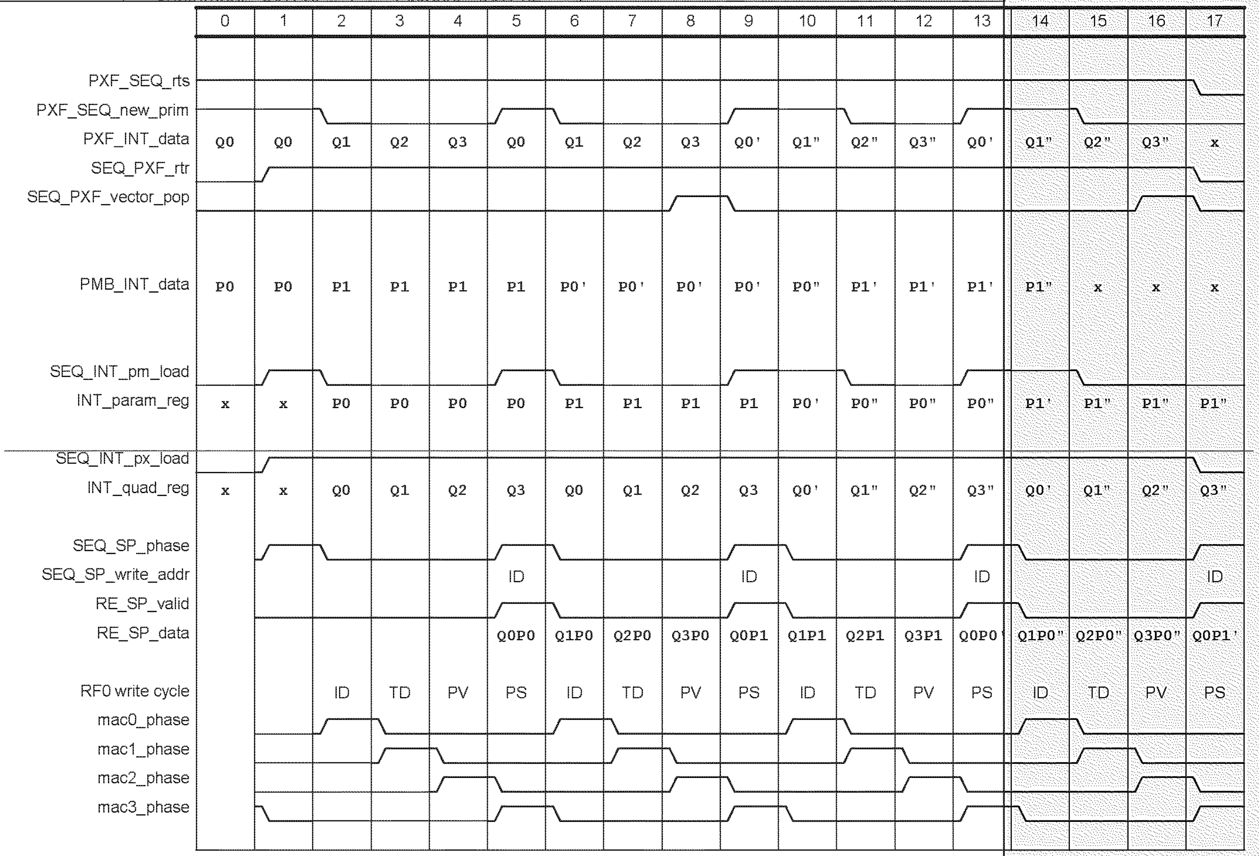


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
SEQ_SP_constant0						C0_0	C0_1	C0_2	C0_3									
SEQ_SP_constant1						C1_0	C1_1	C1_2	C1_3									
SEQ_SP_read_addr		srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA
SEQ_SP_phase																		
RE_SP_data[511:384]		ID				ID				ID				ID				
SEQ_SP_instruction							I0_0	I0_1	I0_2	I0_3								
SEQ_SP_instr_start																		
mac0_phase																		
mac0_cycle_count			0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
RF0_read_data								srcA	srcB	srcC	TC							
mac0_vector_result														a	r	g	b	
SEQ_SP_write_addr		ID	-	PV	PS	ID	-	PV	PS	ID	-	PV	PS	ID	-	PV	PS	
RF0 write cycle			ID	TD	PV	PS	ID	TD	PV	PS	ID	TD	PV	PS	ID	TD	PV	PS

Timing Diagram 1: Sequencer to Shader Pipe 0, Shader Unit 0, MAC 0

~~14.2 Sequencer to Shader Pipe~~

Formatted: Bullets and Numbering

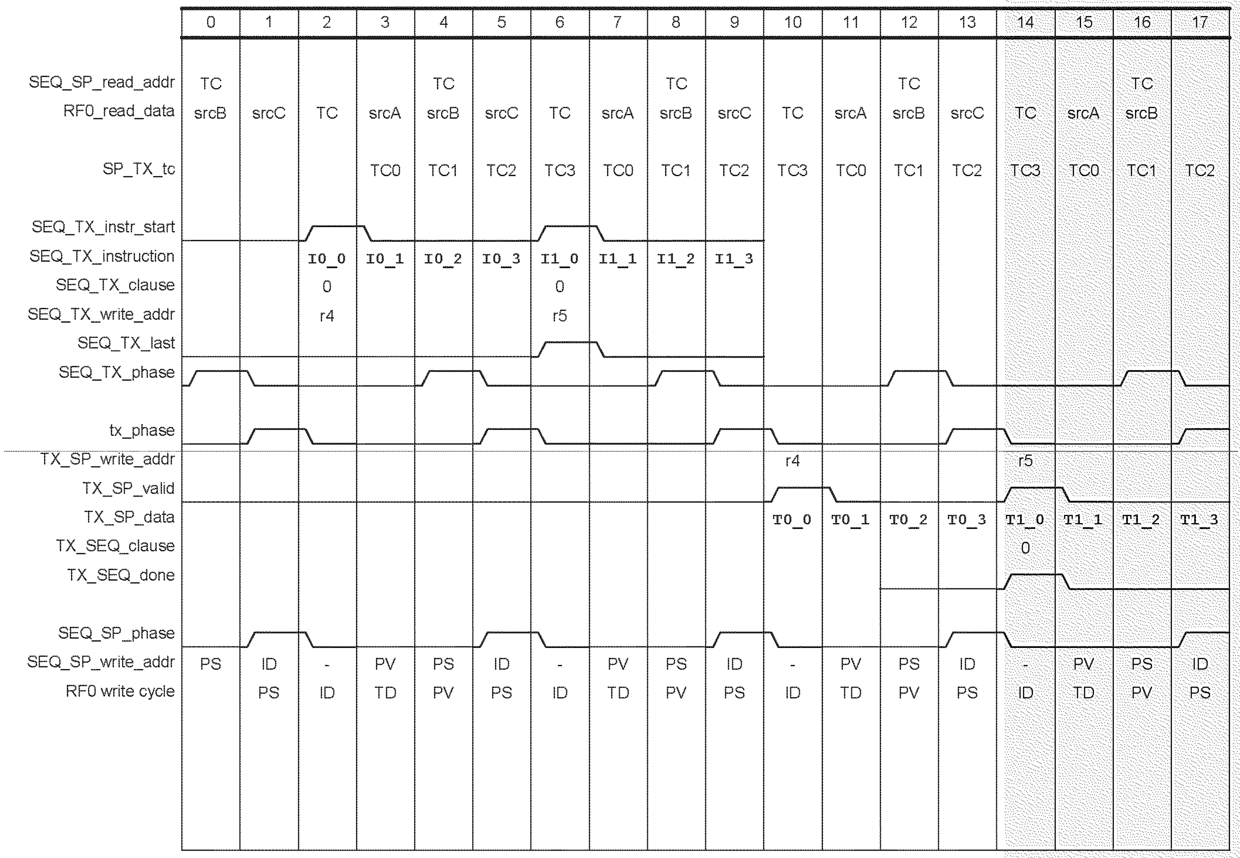


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
PXF_SEQ_rts																		
PXF_SEQ_new_prim																		
PXF_INT_data	Q0	Q0	Q1	Q2	Q3	Q0	Q1	Q2	Q3	Q0'	Q1"	Q2"	Q3"	Q0'	Q1"	Q2"	Q3"	x
SEQ_PXF_rtr																		
SEQ_PXF_vector_pop																		
PMB_INT_data	P0	P0	P1	P1	P1	P1	P0'	P0'	P0'	P0'	P0"	P1'	P1'	P1'	P1"	x	x	x
SEQ_INT_pm_load																		
INT_param_reg	x	x	P0	P0	P0	P0	P1	P1	P1	P1	P0'	P0"	P0"	P0"	P1'	P1"	P1"	P1"
SEQ_INT_px_load																		
INT_quad_reg	x	x	Q0	Q1	Q2	Q3	Q0	Q1	Q2	Q3	Q0'	Q1"	Q2"	Q3"	Q0'	Q1"	Q2"	Q3"
SEQ_SP_phase																		
SEQ_SP_write_addr						ID				ID				ID				ID
RE_SP_valid																		
RE_SP_data						Q0P0	Q1P0	Q2P0	Q3P0	Q0P1	Q1P1	Q2P1	Q3P1	Q0P0'	Q1P0"	Q2P0"	Q3P0"	Q0P1'
RF0 write cycle			ID	TD	PV	PS	ID	TD	PV	PS	ID	TD	PV	PS	ID	TD	PV	PS
mac0_phase																		
mac1_phase																		
mac2_phase																		
mac3_phase																		

Timing Diagram 2: RE Interpolator to Shader Pipe Data Transfer

~~14.3 Sequencer to Texture Pipe~~

Formatted: Bullets and Numbering



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
SEQ_SP_read_addr	TC				TC				TC				TC				TC	
RF0_read_data	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	srcC	TC	srcA	srcB	
SP_TX_tc				TC0	TC1	TC2	TC3	TC0	TC1	TC2	TC3	TC0	TC1	TC2	TC3	TC0	TC1	TC2
SEQ_TX_instr_start																		
SEQ_TX_instruction			I0_0	I0_1	I0_2	I0_3	I1_0	I1_1	I1_2	I1_3								
SEQ_TX_clause			0				0											
SEQ_TX_write_addr			r4				r5											
SEQ_TX_last																		
SEQ_TX_phase																		
tx_phase																		
TX_SP_write_addr											r4				r5			
TX_SP_valid																		
TX_SP_data											T0_0	T0_1	T0_2	T0_3	T1_0	T1_1	T1_2	T1_3
TX_SEQ_clause															0			
TX_SEQ_done																		
SEQ_SP_phase																		
SEQ_SP_write_addr	PS	ID	-	PV	PS	ID	-	PV	PS	ID	-	PV	PS	ID	-	PV	PS	ID
RF0 write cycle		PS	ID	TD	PV	PS	ID	TD	PV	PS	ID	TD	PV	PS	ID	TD	PV	PS

Timing Diagram 3: Sequencer - Texture Unit Interface and Texture Unit - Shader Pipe Data Transfer

~~14.4 Timing diagrams explanations~~

Formatted: Bullets and Numbering

~~The numbering of the four shader pipes, the four shader units, and the four MACs is from left to right and from 0 to 3. So for example the most significant 512 bits of a SP goes to SU0 and the least significant 512 bits go to SU3; within SU0, the most significant 128 bits go to MAC0 and the least significant 128 bits go to MAC3.~~
~~The following assumptions are made:~~

~~1.all block to block signals are register to register~~

Formatted: Bullets and Numbering

~~2.for register file reads, the RF read data is available in the MAC one clock after a RF read address is registered into the MAC (this is the same as saying the read data is valid out of the RF two clocks after the address is asserted on the SEQ to SP interface)~~

~~14.4.1 *Timing Diagram 1: Sequencer to Shader Pipe 0, Shader Unit 0, MAC 0*~~

~~This diagram shows the basics of the Sequencer to Shader Pipe interface. For simplicity only the timing relative to MAC0 is shown. The timing for MAC1 is one clock later than MAC0, MAC2 one clock later than MAC1, etc. This means that most of the signals need to be delayed in the SP by one cycle for MAC1, two cycles for MAC2, and three cycles for MAC3.~~
~~**SEQ_SP_constant0**: Constant 0 (128 bits over 4 cycles). Pipelined in SP for other MACs.~~
~~**SEQ_SP_constant1**: Constant 1 (128 bits over 4 cycles). Pipelined in SP for other MACs.~~
~~**SEQ_SP_read_addr**: Register File Read Address (8 bits). Pipelined in SP for other MACs.~~
~~**SEQ_SP_phase**: This signal syncs the data transfer to the RF from the RE, as well as defining the order of all writes into the RF. It is asserted during the cycle that interpolated data (ID) is valid on the RE_SP_ID bus. Pipelined in SP for other MACs.~~
~~**RE_SP_ID[511:384]**: This is the most significant 128 bits of the RE_SP_data interface (meaning that this MAC0 is in SU0).~~
~~**SEQ_SP_instruction**: 96 bits of instruction are sent over 4 cycles. Pipelined in SP for other MACs.~~
~~**SEQ_SP_instr_start**: control bit that signals the first cycle of the instruction transfer. Pipelined in SP for other MACs.~~
~~**mac0_phase**: registered version of SEQ_SP_phase used in MAC0 (this may not be he actual signal name).~~
~~**mac0_cycle_count**: a counter inside the MAC that keeps track of the RF write cycles; 0 here corresponds to the cycle RE interpolated data is written (this may not be he actual signal name).~~
~~**RF0_read_data**: data that is read out of MAC0's register file (this may not be he actual signal name).~~
~~**mac0_vector_result**: the 32-bit output of the vector ALU (PV is built up over 4 cycles) (this may not be he actual signal name).~~
~~**SEQ_SP_write_addr**: Register File Write Address (8 bits). Note that the SEQ does not send the Texture Data write address over this bus. Pipelined in SP for other MACs.~~
~~**RF0 write cycle**: the cycles allocated to the different write sources (ID = Interpolated Data, TD = Texture Data, PV = Previous Vector, PS = Previous Scalar) (not a signal – just a reference point on the diagram).~~

~~14.4.2 *Timing Diagram 2: RE Interpolator to Shader Pipe Data Transfer*~~

Formatted: Bullets and Numbering

~~This diagram shows how pixel data (barycentric coordinates i, j, and k) is sent from the Pixel FIFO to the interpolator under SEQ control, and how parameter data (for each vertex) is also sent to the interpolator under SEQ control. The output of the Interpolator is then shown being sent over the RE_SP interface.~~
~~**PXF_SEQ_rts**: Indicates that the output of the pixel FIFO is valid.~~
~~**PXF_SEQ_new_prim**: The current output of the Pixel FIFO is from a different primitive that the previous output. Tells the SEQ that new parameter info must be fetched (if its not from a new prim, then new parameter data is not needed).~~
~~**PXF_INT_data**: Data output of the Pixel FIFO – goes to the Interpolator.~~
~~**SEQ_PXF_rtr**: Indicates that the current Pixel FIFO output will be taken by the Interpolator (driven by SEQ). Then next quad of data will be driven the next cycle.~~
~~**SEQ_PXF_vector_pop**: SEQ tells the Pixel FIFO to pop a vector of pixels (otherwise RTRs cause the data to be cycled between the four quads).~~

~~**PMB_INT_data**: Data from the Parameter Buffer to the Interpolator. (Note that the control of the parameter buffer is TBD).~~

~~**SEQ_INT_pm_load**: controls the loading of parameter data into the Interpolator.~~

~~**INT_param_reg**: register in the Interpolator that holds the per-vertex parameter data while the per-pixel parameters are generated for one or more quads (may not be the actual signal name).~~
~~**SEQ_INT_px_load**: controls the loading of pixel data into the Interpolator.~~
~~**INT_quad_reg**: : register in the Interpolator that holds one quad's worth of pixel data(may not be the actual signal name).~~

~~**SEQ_SP_phase**: see above under TD1.~~
~~**SEQ_SP_write_addr**: see above under TD1.~~
~~**RE_SP_valid**: Interpolator Data Valid – indicates that the SP should write the ID on the appropriate cycle.~~
~~**RE_SP_data**: Data from the RE interpolator to the SP.~~
~~**RF0 write cycle**: see above under TD1.~~
~~**mac*_phase**: see above under TD1. These phase signals help to show the timing offset between the MACs. Note also that each Shader Unit has a set of these signals (all with the same timing).~~

~~14.4.3 *Timing Diagram 3: Sequencer - Texture Unit Interface and Texture Unit - Shader Pipe Data Transfer*~~

Formatted: Bullets and Numbering

~~This diagram starts with the texture coordinate read from the register file and its transfer to the TX. The instruction transfer is then shown, followed by the texture data transfer to the shader pipe.~~
~~**SEQ_SP_read_addr**: see above. Here shows the cycle that the texture coordinate read address is asserted.~~
~~**RF0_read_addr**: see above.~~
~~**SP_TX_tc**: Texture coordinate data sent from the shader pipe to the texture unit.~~
~~**SEQ_TX_instr_start**: Asserted on the first cycle of a SEQ to TX instruction transfer.~~
~~**SEQ_TX_instruction**: 96 bits of texture instruction transferred over 4 cycles.~~
~~**SEQ_TX_clause**: the clause number associated with this instruction.~~
~~**SEQ_TX_write_addr**: RF write index used by TX for returned texture data.~~
~~**SEQ_TX_last**: indicates that this is the last texture instruction of a clause.~~
~~**SEQ_TX_phase**: syncs the texture data write. Note that it is asserted early enough to be registered into TX and still allow TX to source the texture data to the SP on the correct cycle.~~

~~**tx_phase**: the phase signal after being registered into TX.~~
~~**TX_SP_write_addr**: RF write index for texture data.~~
~~**TX_SP_valid**: indicates that valid texture data is being driven to the SP.~~
~~**TX_SP_data**: the texture data.~~
~~**TX_SEQ_clause**: the clause number associated with the texture data.~~
~~**TX_SEQ_done**: indicates to the SEQ that the texture data transfer is complete for the clause number that is on the TX_SEQ_clause bus.~~

~~**SEQ_SP_phase**: see above under TD1 - shown here for reference.~~
~~**SEQ_SP_write_addr**: see above under TD1- shown here for reference.~~
~~**RF0 write cycle**: see above under TD1- shown here for reference.~~

Formatted: Bullets and Numbering

~~15.~~20. 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 from the texture store to feed the ALUs. Two solutions exists for this problem:

1) Let the compiler handle the case and put those instructions in a texture clause so we can use the bandwith 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. 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?

Size of the fifo containing the information of a vector of pixels/vertices. And size of the fifos before the reservation stations.

~~Sequencer Instruction memory, and constant memory.~~

~~Arbitration policy for the output file.~~

Loops and branches.

~~The parameter cache may end up in the PA rather than in the RS. Parameter cache management thus may change.~~

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 2015~~17~~ ~~October, 2001~~5 October	GEN-CXXXXX-REVA	1 of 26

Author: Laurent Lefebvre

Issue To: **Copy No:**

R400 Sequencer Specification

SEQ

Version 0.8~~7~~

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

AUTOMATICALLY UPDATED FIELDS:
Document Location: C:\perforce\r400\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.

1. Overview

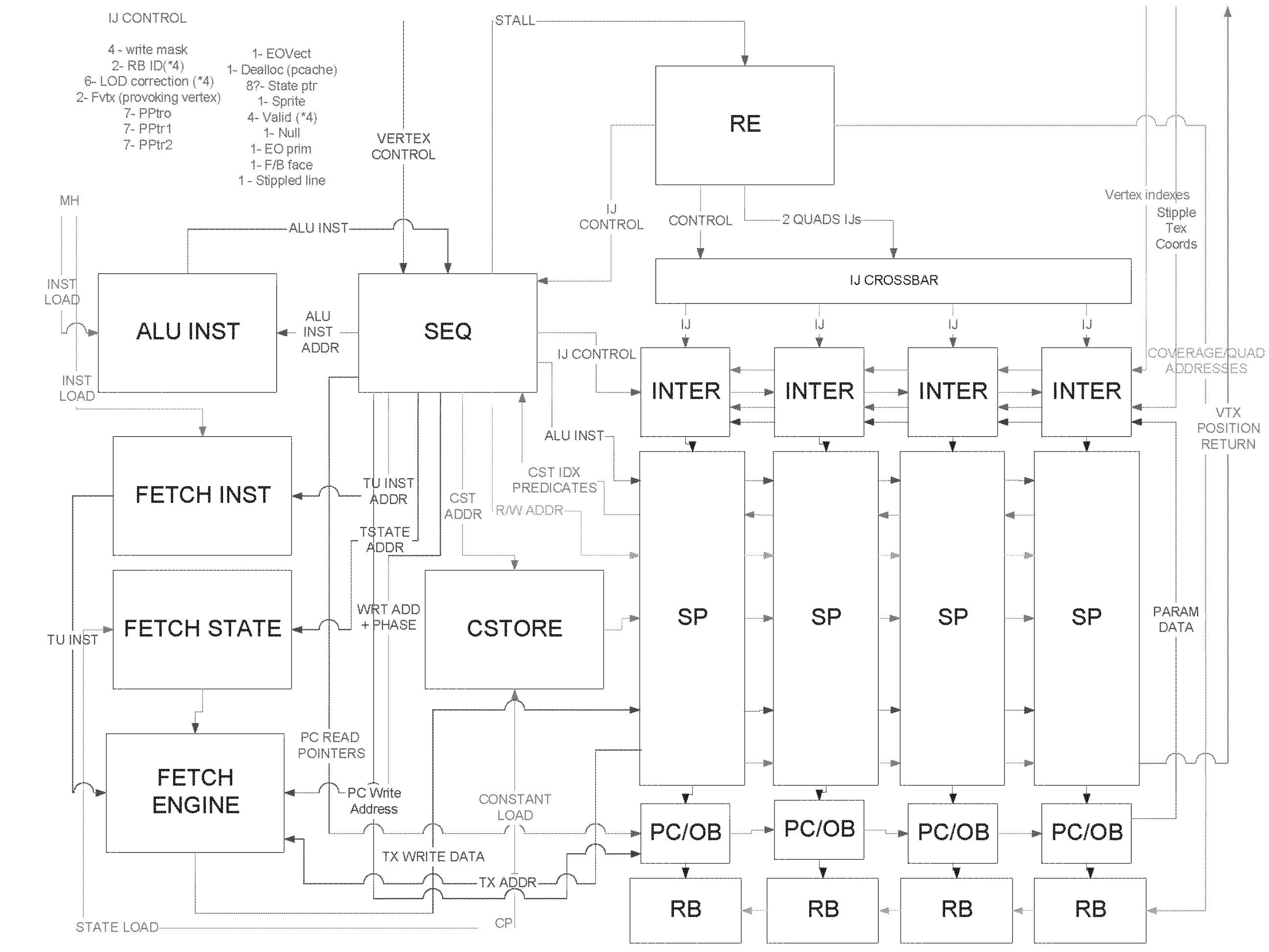
The sequencer first arbitrates between vectors of 64 vertices that arrive directly from primitive assembly and vectors of 16 quads (64 pixels) that are generated in the raster engine.

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.

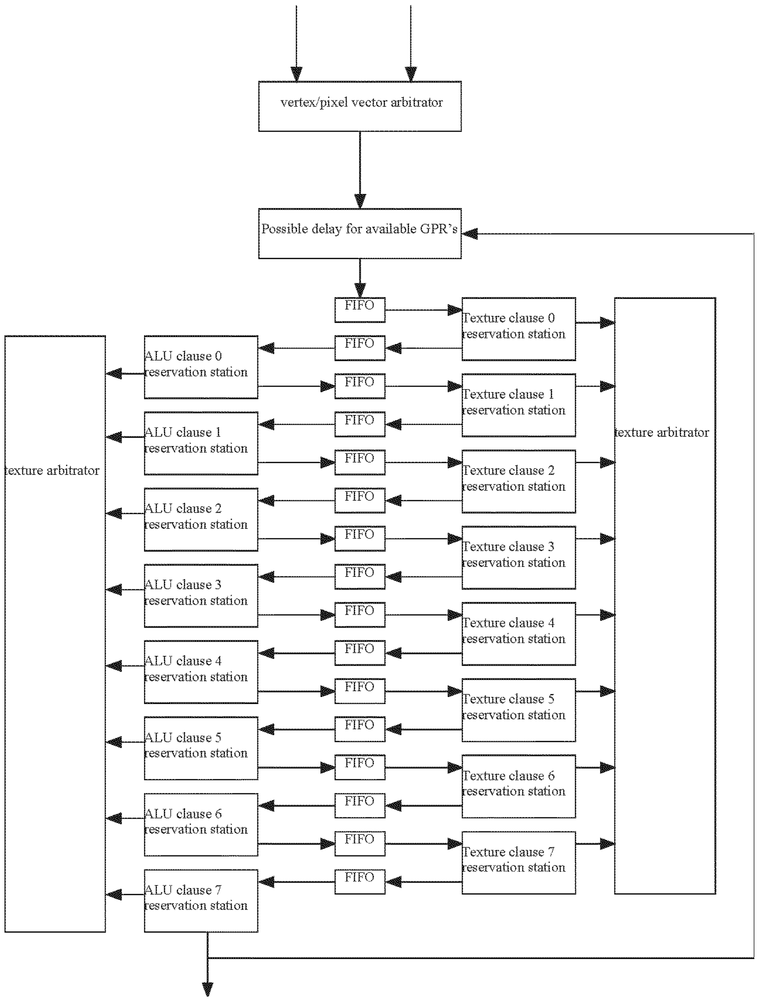
The sequencer is based on the R300 design. It chooses two ALU clauses and a ~~texture~~fetch clause to execute, and executes all of the instructions in a clause before looking for a new clause of the same type. Two ALU clauses are executed interleaved to hide the ALU latency. Each vector will have eight ~~texture~~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 ~~texture~~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 ~~texture~~fetch stations to choose a ~~texture~~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 ~~texture~~fetch fetches initiated by the previous ~~texture~~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 thus there is only one sequencer for the whole chip.

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 2015~~17~~ October, 20015 October	GEN-CXXXXX-REVA	5 of 26



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.

The rasterizer always checks the vertices FIFO first and if allowed by the sequencer sends the data to the shader. If the vertex FIFO is empty then, the rasterizer takes the first entry of the pixel FIFO (a vector of 64 pixels) and sends it to the interpolators. Then the sequencer takes control of the packet. The packet consists of 21 bits of state, 6-7 bits for the base address of the Shader program and some information on the coverage to determine ~~texture~~fetch LOD. All other information (2x2 adresses) is put in a FIFO (one for the pixels and one for the vertices) and retrieved when the packet finishes its last clause.

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 ~~texture~~fetch machine issues a texure request and corresponding register address for the ~~texture~~fetch address (ta). A small command (tcmd) is passed to the ~~texture~~fetch system identifying the current level number (0) as well as the register write address for the ~~texture~~fetch return data. One ~~texture~~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 ~~texture~~fetch unit writes the data to the register file using the write address that was provided by the level 0 ~~texture~~fetch machine and sends the clause number (0) to the level 0 ~~texture~~fetch state machine to signify that the write is done and thus the data is ready. Then, the level 0 ~~texture~~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.

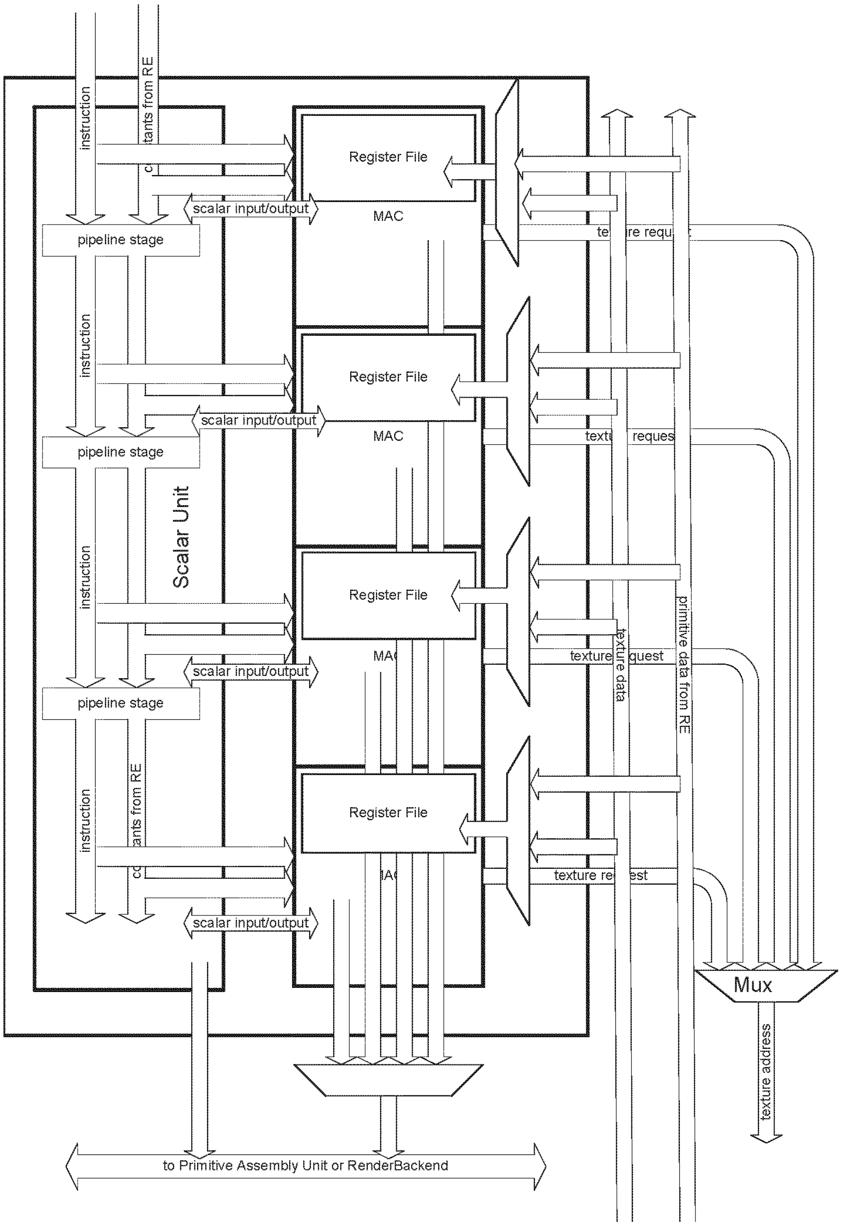
There will always be two active ALU clauses at any given time (and two arbitrers). One arbitrer will arbitrate over the odd clock cycles and the other one will arbitrate over the even clock cycles. The only constraints between the two arbitrers is that they are not allowed to pick the same clause number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

If the packet is a vertex packet, upon reaching ALU clause 4, it can export the position if the position is ready. So the arbitrer must prevent ALU clause 4 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).

All other level process in the same way until the packet finally reaches the last ALU machine (8). On completion of the level 8 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.

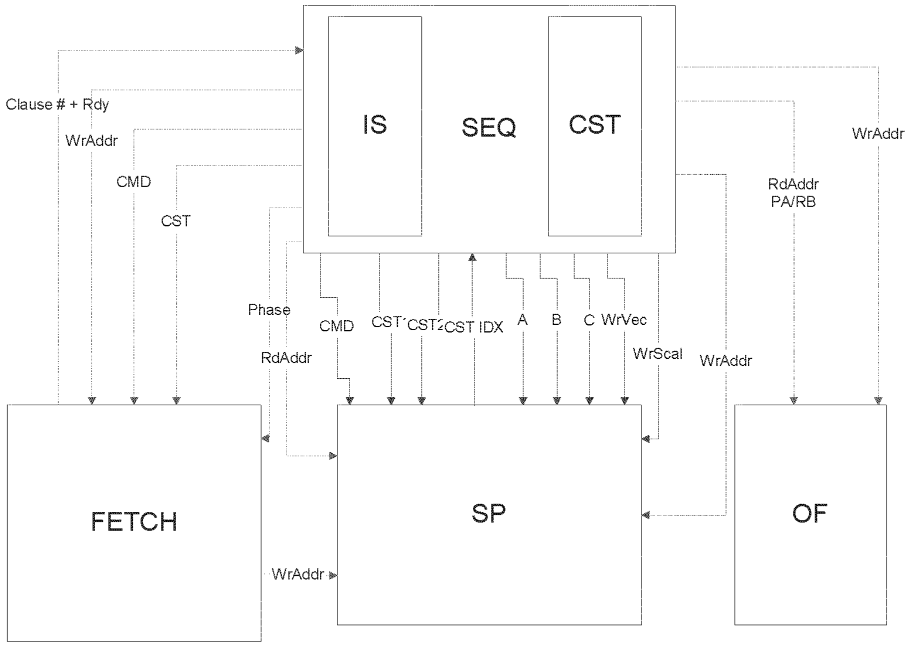
Only two ALU state machine may have access to the register file address bus or the instruction decode bus at one time. Similarly, only one ~~texture~~fetch state machine may have access to the register file address bus at one time. Arbitration is performed by three arbitrer blocks (two for the ALU state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

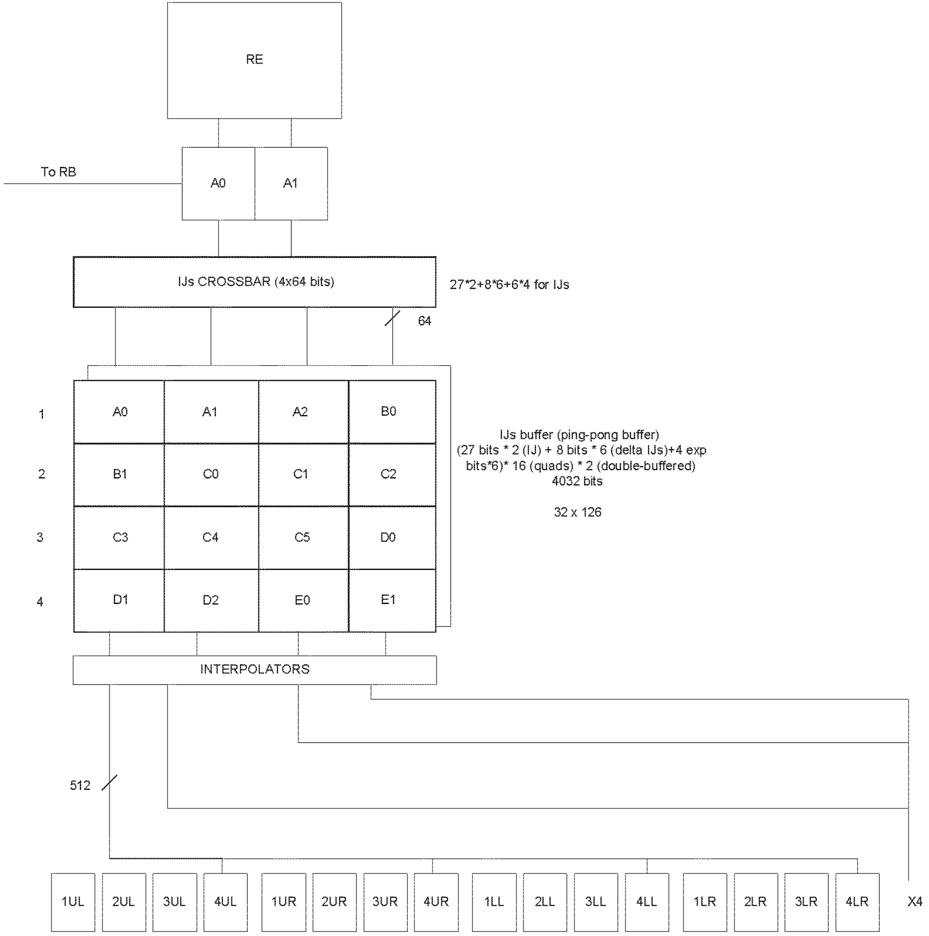
1.3 Control Graph



In green is represented the ~~Texture~~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)
27*2+8*6+6*4 for IJs
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)
(27 bits * 2 (IJ) + 8 bits * 6 (delta IJs)+4 exp bits*6)* 16 (quads) * 2 (double-buffered)
4032 bits
32 x 126
INTERPOLATORS
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
SP0	A0	B1	C3	D1					A0	B1	C3	D1					V 0-3	V 16-19	V 32-35	V 48-51
SP1	A1		C4	D2		C0			A1		C4	D2		C0			V 4-7	V 20-23	V 36-39	V 52-55
SP2	A2		C5			C1		E0	A2		C5			C1		E0	V 8-11	V 24-27	V 40-43	V 56-59
SP3					B0	C2	D0	E1					B0	C2	D0	E1	V 12-15	V 28-31	V 44-47	V 60-63
				P0								P1								

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~~R/1W~~ port memory; we use ~~2~~ 1 clock~~s~~ to load the ALU instruction, ~~and~~ 1~~2~~ clocks to load the ~~Texture~~Fetch instruction, 1 clock to load 2 control flow instructions and 1 clock to write instructions.

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 Store

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 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.

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. The control program has 4(5) instructions:

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~~31~~:0]
Loop_count[7:0][15:0]
Loop_Start[7:0] [15:0]
Loop_End[7:0] [15:0]

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

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 ~~four~~ ten basic instructions:

Execute
Conditional_execute
~~(~~Conditional_-Execut~~e~~e_-Predicates~~)~~
Conditional_execute_or_Jump
Conditional_jump
Call
Return
Loop_start
Loop_end
End_of_clause

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. ~~if the loop end condition is not met.~~

~~if we try and fit the control flow instructions into 32 bit words, the following instructions are possible choices:~~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~~ 4k instructions at the specified address in the instruction memory.

Conditionnal_Execute_or_Jump						
47	46 … 42	41 … 34	33	32 … 21	20 … 12	11 … 0
Addressing	00010	Booleans	Condition	Jump address	Instruction_count	Exec Address

I~~i~~f the specified boolean (~~6~~8 bits can address ~~64~~256 booleans) meets the specified condition then execute the specified instructions (up to ~~64~~ 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.

Conditionnal_Execute						
47	46 … 42	41 … 34	33	32 … 21	20 … 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 512 instructions)

Conditionnal_Execute_Predicates						
47	46 … 42	41 … 38	37	36 … 21	20 … 12	11 … 0
Addressing	00100	Predicate vector	Condition	RESERVED	Instruction_count	Exec Address

Check the OR of all current predicate bits. If OR matches the condition execute the specified number of instructions.

~~Initialize the specified loop~~

~~If the loop condition of the current loop is not met, then branch back to the specified address in the control flow program. Note that jumping back to the loop_start results in an infinite loop, the jump should be to loop_start+1.~~

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.

Conditionnal_Jump					
47	46 … 42	41 … 34	33	32 … 12	11 … 0
Addressing	01010	Boolean address	Condition	RESERVED	Address

If condition met, jumps to the address. FORWARD jump only allowed.

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 is going to break the loop. The sequencer will keep a loop index value of 17 bits. This will be updated everytime 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 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

~~The control program can be up to 256 instructions in size. (There is an offset added to the address from the render state before accessing the control flow program memory to allow for multiple programs resident at the same time)~~

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 addresses from the control program are added to another offset to allow for multiple programs resident at the same time.~~

~~Under this model, all subroutine calls must be inlined into the control program.~~

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 ~~the~~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 ~~(two sets for interleaved operation)~~. This predicate is not maintained across clause boundaries. The # sign is used to specify wich 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 exemple, 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 Register file indexing

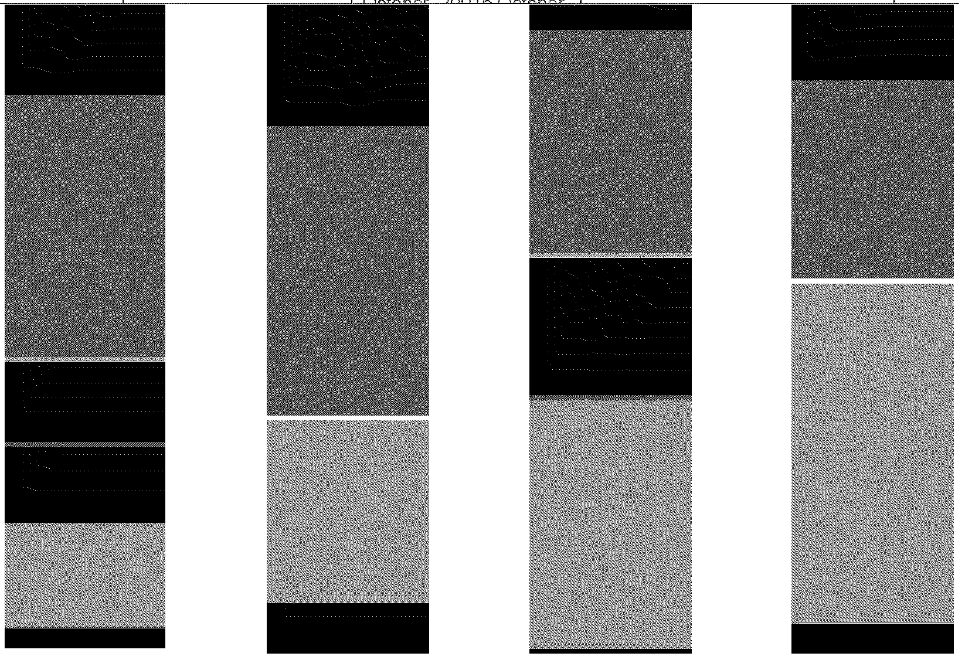
Because we can have loops in ~~texture~~fetch clause, we need to be able to index into the register file in order to retrieve the data created in a ~~texture~~fetch clause loop and use it into an ALU clause. The instruction will include the base address for register indexing and the instruction will 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_counter~~loop_index and this becomes our new address that we give to the shader pipe.

7. 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.

8. ~~Texture~~Fetch Arbitration

The ~~texture~~fetch arbitration logic chooses one of the 8 potentially pending ~~texture~~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 ~~texture~~ fetch per clock (or 4 fetches in one clock every 4 clocks) until all the ~~texture~~ fetch instructions of the clause are sent. This means that there cannot be any dependencies between two ~~texture~~ fetches of the same clause.

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

9. ALU Arbitration

ALU arbitration proceeds in almost the same way than ~~texture~~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.

10. 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?). The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU arbitrer will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

11. Content of the reservation station FIFOs

3 bits of Render State 6-7 bits for the base address of the instruction store, some bits for LOD correction and coverage mask information in order to fetch ~~texture~~fetch for only valid pixels. Every other information (such as the coverage mask, quad address, etc.) is put in a FIFO and is retrieved when the quad exits the shader pipe to enter in the output file buffer. 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.

12. 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).

13. 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 19~~24~~x24 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 ~~full~~ 19~~24~~x24 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 24x24 yielding 8 bits (IJs): 6~~
Subtracts ~~24x24~~ 19x24 (Parameters): 2
Adds: 8

FORMAT OF P0's IJ : Mantissa ~~23~~19 Exp 4 for I + Sign
Mantissa ~~23~~19 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~~23~~*2 + 8*6 + 4*8 + 4*2 = 126 ~~(rounded up on the bus to 128)~~

14. 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.

15. Vertex position exporting

On clause 4 (or 5) the vertex shader can export to the PA both the vertex position and the point sprite. It can also do so at clause 8 if not done at clause 4. ~~The export is done by putting the exported position back into the GPRs. Then using the texture port in an opportunistic manner, 16 positions are put into a FIFO (16x128) in order (left to right). This fifo drains 128 bits per clock to the PA and once empty is filled up again with sprite sizes (if any). The process is~~

~~repeated 4 times. The sequencer must make sure that the program doesn't enter ALU clause 5 (it can enter texture clause 5) because the registers can be reused at this point. The sequencer must also make sure not to dealocate an exporting program before it is done exporting data.~~ 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.

Formatted: Bullets and Numbering

16. 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.

Formatted: Bullets and Numbering

~~16.~~17. Registers

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)
Vshader_fetch[~~7~~11:0][7:0]	eight ~~8~~ 12 bit pointers to the location where each clauses control program is located
Vshader_alu[~~7~~11:0][7:0]	eight ~~8~~ 12 bit pointers to the location where each clauses control program is located
Pshader_fetch[~~7~~11:0][7:0]	eight ~~8~~ 12 bit pointers to the location where each clauses control program is located
Pshader_alu[~~7~~11:0][7:0]	eight ~~8~~ 12 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
~~PCNTLSHADER~~	~~base pointer for the pixel control program~~
~~VCNTLSHADER~~	~~base pointer for the vertex control program~~
VWRAP	wrap point for the vertex shader instruction store
PWRAP	wrap point for the pixel shader instruction store
REG_ALLOC_PIX	number of registers to allocate for pixel shader programs
REG_ALLOC_VERT	number of registers to allocate for vertex shader programs
PARAM_MASK[0…16]	parameter mask to specify ~~wich~~ how parameters maps in the pixel shader
FLAT_GOUR[0…16]	wich parameters are to be gouraud shaded
GEN_TEX[0….16]	for wich parameters do we need to generate tex coords.
CYL_WRAP[0…64]	for wich ~~vertices~~ parameters (and channels (xyzw)) do we do the cyl wrapping.
P_EXPORT	number of exports for pixel shader
V_EXPORT	number of exports for vertex shader (also the number of interpolated parameters for pixel shaders)
V_EXPORT_LOC	Vertex shader exporting to RB or the PCACHE
ARBITRATION_policy	policy of the arbitration between vetexes and pixels

Formatted: Bullets and Numbering

~~17.~~18. Interfaces

~~17.1~~18.1 External Interfaces

~~17.1.1~~18.1.1 *PA/SC to RE : 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

Name	Direction	Bits	Description

IJs	PA→RE	63~~4~~	IJ information sent over 2 clocks
Mask	PA→RE	1	Write Mask

~~17.1.2~~18.1.2 *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.

Name	Direction	Bits	Description
Write Mask	PA→SEQ(RE)	4	Quad Write mask left to right
RB_ID	PA→SEQ(RE)	8	RB id for each quad sent 2 bits per quad
LOD_CORRECT	PA→SEQ(RE)	24	LOD correction per quad (6 bits per quad)
FVTX	PA→SEQ(RE)	2	Provoking vertex for flat shading
PPTR0	PA→SEQ(RE)	11	P Store pointer for vertex 0
PPRT1	PA→SEQ(RE)	11	P Store pointer for vertex 1
PPTR2	PA→SEQ(RE)	11	P Store pointer for vertex 2
E_OFF_VECTOR	PA→SEQ(RE)	1	End of the vector
DEALLOC	PA→SEQ(RE)	1	Deallocation token for the P Store
STATE	PA→SEQ(RE)	21	State/constant pointer (6*3+3)
VALID	PA→SEQ(RE)	16	Valid bits for all pixels
NULL	PA→SEQ(RE)	1	Null Primitive (for PC deallocation purposes)
E_OFF_PRIM	PA→SEQ(RE)	1	End Of the primitive
FBFACE	PA→SEQ(RE)	1	Front face = 1, back face = 0
~~STIPPLE~~_~~LINE~~TYPE	PA→SEQ(RE)	3~~1~~	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
RTRn	SEQ→PA	1	Stalls the PA in n clocks
RTS	PA→SEQ(RE)	1	PA ready to send data
QuadX	PA→SEQ(RE)	~~40~~8	Quad X address ~~10~~ 2 bits per quad
QuadY	PA→SEQ(RE)	~~40~~8	Quad Y address ~~10~~ 2 bits per quad

~~17.1.3~~18.1.3 *~~PA/S~~VGT~~C~~ to RE : Vertex Bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Vertex indexes	VGT~~PA~~→RE	~~32~~128	Pointers of indexes or HOS surface information
EOF_vector	VGT→RE	1	End of the vector
Inputs_vert	VGT→RE	1	0: Normal 128 bits per vert 1: double 256 bits per vert

~~17.1.4~~18.1.4 *VGT~~PA/SC~~ to SEQ : Vertex Control Bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
STATE	VGT~~PA~~→SEQ	21	Render State (6*3+3 for constants)
~~Write Mask~~Vert counter	VGT~~PA~~→SEQ	~~64~~26	Which vertices are valid
Inputs_vert	VGT→SEQ	1	0: Normal 128 bits per vert 1: double 256 bits per vert

This information needs to be sent over 64 clocks.

~~17.1.5~~18.1.5 *CP to SEQ : Constant store load*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Constant Address	CP→SEQ	8	Address of the block of 4 constants

Constant Data	CP→SEQ	512	Data sent over 4 clocks
Remap Address	CP→SEQ	10	Remaping address write address
Remap Data pointer	CP→SEQ	8	Remaping pointer

~~17.1.6~~18.1.6 *CP to SEQ : ~~Texture~~Fetch State store load*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Constant Address	CP→SEQ	8	Address of the block of 4 state constants
Constant Data	CP→SEQ	512	Data sent over 4 clocks
Remap Address	CP→SEQ	10	Remaping address write address
Remap Data pointer	CP→SEQ	8	Remaping pointer

~~17.1.7~~18.1.7 *CP to SEQ : Control State store load*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description

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

~~17.1.8~~18.1.8 *MH to SEQ: Instruction store Load*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Instruction address	MH→SEQ	12	Instruction address
Instruction	MH→SEQ	96	Instruction X times
Control Instruction address	MH→SEQ	9	Pointer to the control instruction store
Control Instruction	MH→SEQ	32	Control Instruction X times

~~17.1.9~~18.1.9 *SP to RB : Pixel read from RBs*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
~~Pixel Data~~Export_data	SP→RB	~~256~~64	~~2 pixels (or ½ quad)~~a pair of 32 bits channel values
ExportID	SP→RB	9	0cvvvvhqq: Vertex data vvvv 0-15 from first or second clause (c=0 or 1), XY or ZW components (h=0 or 1), quad 0-3 in the shader (qq= 0-3) 1cbbkttqq: Pixel data for buffer bb (0-3) from first or second clause (0-1) killed or not (k=1 or 0) quad 0-3 in the shader and data is RG (tt=0), BA (tt=1) or Z (tt=2)
ExportMask	SP→RB	2	Specifies whether to write low, high or both 32 bit words. If export mask is 00 data is invalid
ExportLast	SP→RB	1	Last export instruction of the clause

18.1.10 *SEQ to RB : Control bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Type	SEQ→RB	1	0: Pixel 1: Vertex
Interleaving	SEQ→RB	1	0: first interleaved clause 1: second interleaved clause
Export_size	SEQ→RB	4	0 thru 16 parameters exported for vertexes (vvvv) OR (bbzs) 1-4 color buffers (bb), two component (s=0) or 4 component colors (s=1) with z (z=1) or without z (z=0)
Valid	SEQ→RB	1	Data valid

Only one exporting clause (7) can be selected at any given time.

18.1.11 *RB to SEQ : Output file control*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Buff_Full	RB→SEQ	1	Set if full

Avail_size	RB→SEQ	6	Size available in output buffers (in 32bits increments)

~~17.1.10~~18.1.12 *SP to ~~PA/SC~~RB : Position return bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Position return	SP→~~PA~~RB	128	Position data or sprite size (per clock)
Parameter cache pointer	SP→~~PA~~RB	11	Pointer where the data will be in the parameter cache for each vertex

For point sprites and position exports the size and position are interleaved on a 16 x 16 basis. We export ~~16~~1 position~~s~~ then ~~16~~ 1 point sprite sizes. The storage used is of 64x128 bits for position and 64x32 bits for sprite size, it is taken from the output buffer. Additionnally,if needed the edge flags are packed into the bits of the sprite sizes.~~The registers are taken until the next ALU clause where they are going to be available again. Thus the sequencer has to make sure that we finished exporting data before allowing the program in the next ALU clause.~~

~~17.1.11~~18.1.13 *Shader Engine to ~~Texture~~Fetch Unit Bus (Fast Bus)*

Formatted: Bullets and Numbering

Four quad's worth of addresses is transferred to ~~Texture~~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 ~~Texture~~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_Read_Register_Index	SEQ->SP	7	Index into Register files for reading ~~Texture~~Fetch Address
Tex_RegFile_Read_Data	SP->TEX	2048	16 ~~Texture~~Fetch Addresses read from the Register file
Tex_Write_Register_Index	SEQ->TEX	7	Index into Register file for write of returned ~~Texture~~Fetch Data

~~17.1.12~~18.1.14 *Sequencer to ~~Texture~~Fetch Unit bus (Slow Bus)*

Formatted: Bullets and Numbering

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

Name	Direction	Bits	Description
Tex_Ready	TEX→ SEQ	1	Data ready
Tex_Clause_Num	TEX→ SEQ	3	Clause number
Tex_cst	SEQ→TEX	10	~~Texture~~Fetch state address 10 bits sent over 4 clocks
Tex_Inst	SEQ→TEX	12	~~Texture~~ ~~f~~Fetch instruction address 12 bits sent over 4 clocks
EO_CLAUSE	SEQ→TEX	1	Last instruction of the clause
PHASE	SEQ→TEX	1	Write phase signal

~~18.~~19. Internal interfaces

Formatted: Bullets and Numbering

~~19.~~20. Examples of program executions

~~19.1.1~~20.1.1 *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 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 ~~texture~~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 ~~texture~~fetch clause 0 from the global instruction store
    - TSM0 was first selected by the TSM arbiter before it could start
7. all instructions of ~~texture~~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 ~~Texture~~Fetch Unit to complete; it passes the register file write index for the ~~texture~~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 ~~texture~~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

~~19.1.2~~20.1.2 *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 ~~texture~~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 (such as quad address for example) travels in a separate FIFO
7. TSM0 accepts the control packet and fetches the instructions for ~~texture~~fetch clause 0 from the global instruction store
    - TSM0 was first selected by the TSM arbiter before it could start
8. all instructions of ~~texture~~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 ~~texture~~fetch requests made to the ~~Texture~~Fetch Unit to complete; it passes the register file write index for the ~~texture~~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 ~~texture~~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

~~19.1.3~~20.1.3 *Notes*

Formatted: Bullets and Numbering

14. the state machines and arbitrers 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, parameter buffer allocation, loops and branches and parameter cache de-allocation still needs to be specked out.

Formatted: Bullets and Numbering

~~20.~~21. 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 from the ~~texture~~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 ~~texture~~fetch clause so we can use the bandwith 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. 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?

Size of the fifo containing the information of a vector of pixels/vertices. And size of the fifos before the reservation stations.

Loops and branches.

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 2015~~19 October, 200117~~	GEN-CXXXXX-REVA	1 of 27

Author: Laurent Lefebvre

Issue To:

Copy No:

R400 Sequencer Specification

SEQ

Version 0.9~~8~~

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

AUTOMATICALLY UPDATED FIELDS:
Document Location: C:\perforce\r400\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.

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.

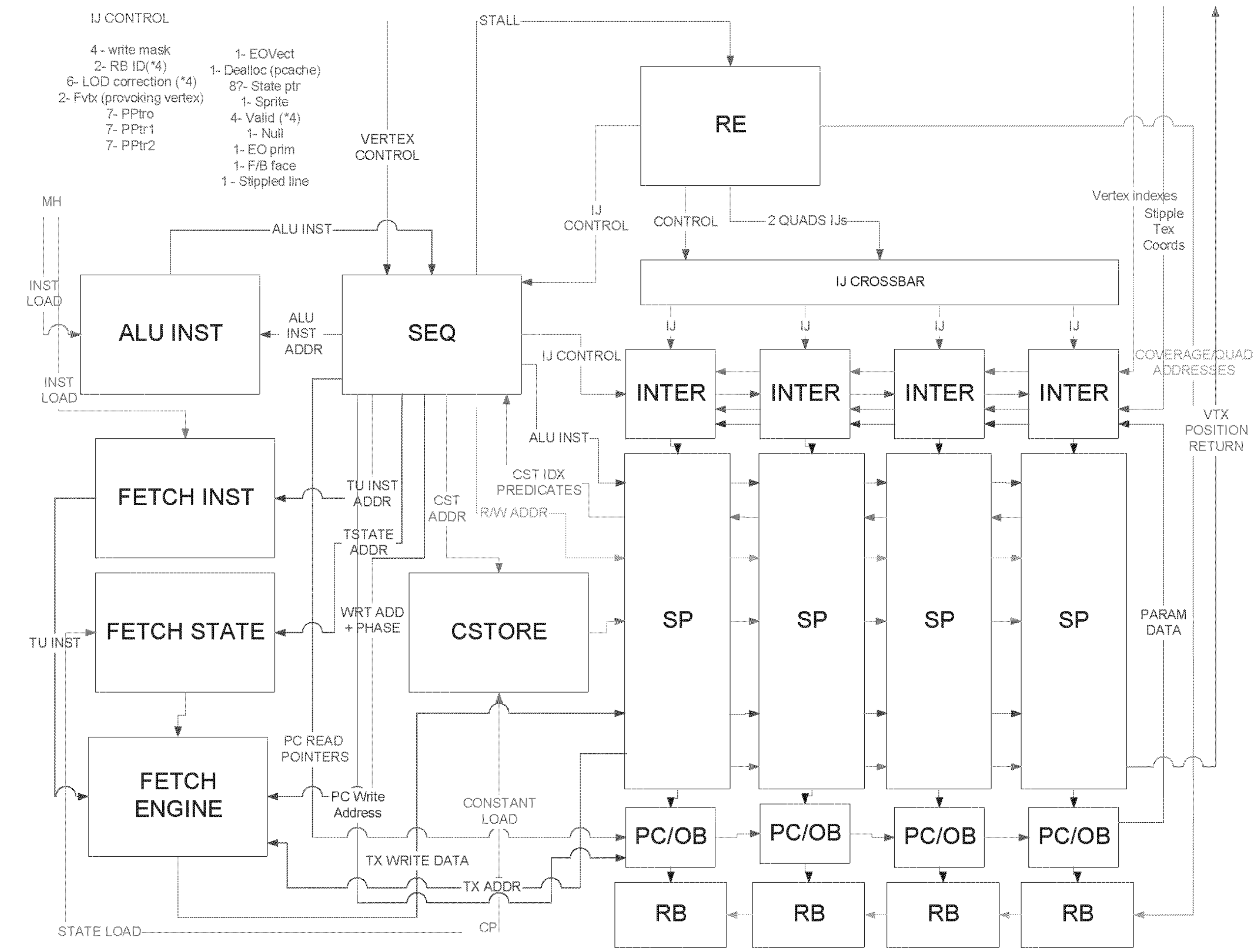
The vertex or pixel program specifies how many GPR's it needs to execute. The sequencer will not start the next vector until the needed space is available.

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.

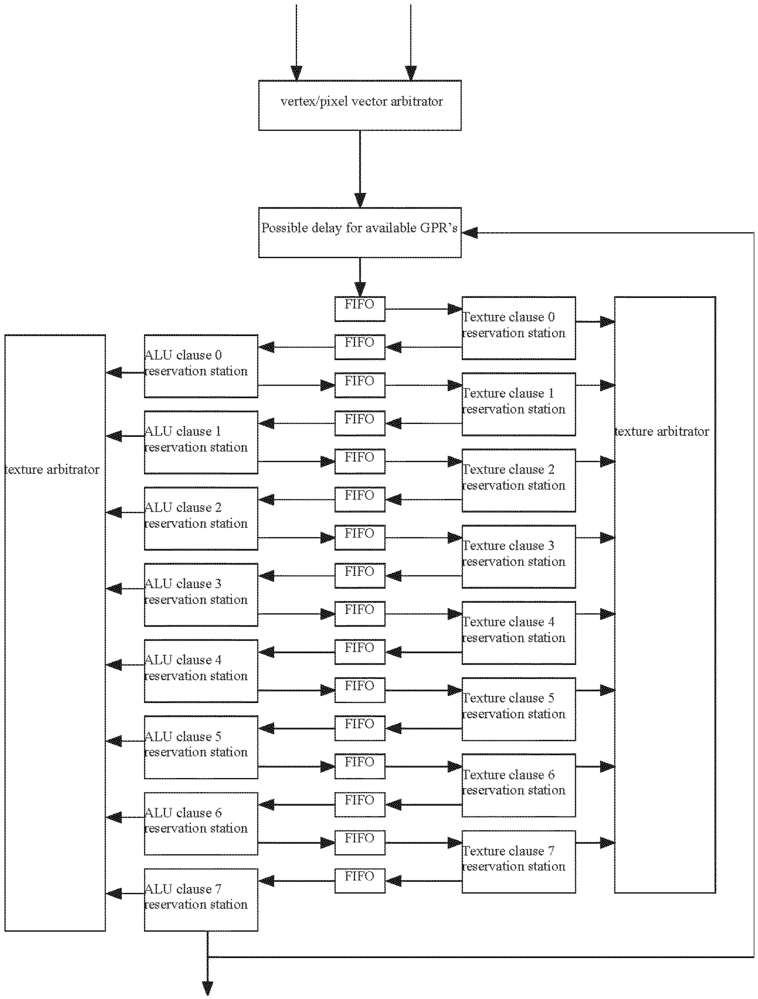
To support the shader pipe the raster engine also contains the shader instruction cache and constant store. There are only one constant store for the whole chip and one instruction store. These will be shared among the four shader pipes. The four shader pipes also execute the same instuction thus there is only one sequencer for the whole chip.

PROTECTIVE ORDER MATERIAL

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 2015~~19~~ ~~October, 200117~~	GEN-CXXXXX-REVA	5 of 27



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.

The rasterizer always checks the vertices FIFO first and if allowed by the sequencer sends the data to the shader. If the vertex FIFO is empty then, the rasterizer takes the first entry of the pixel FIFO (a vector of 64 pixels) and sends it to the interpolators. Then the sequencer takes control of the packet. The packet consists of 21 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. ~~All other information (2x2 adresses) is put in a FIFO (one for the pixels and one for the vertices) and retrieved when the packet finishes its last clause.~~

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 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). One arbitrer 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 is that they are not allowed to pick the same clause number as they other one is currently working on if the packet os of the same type.

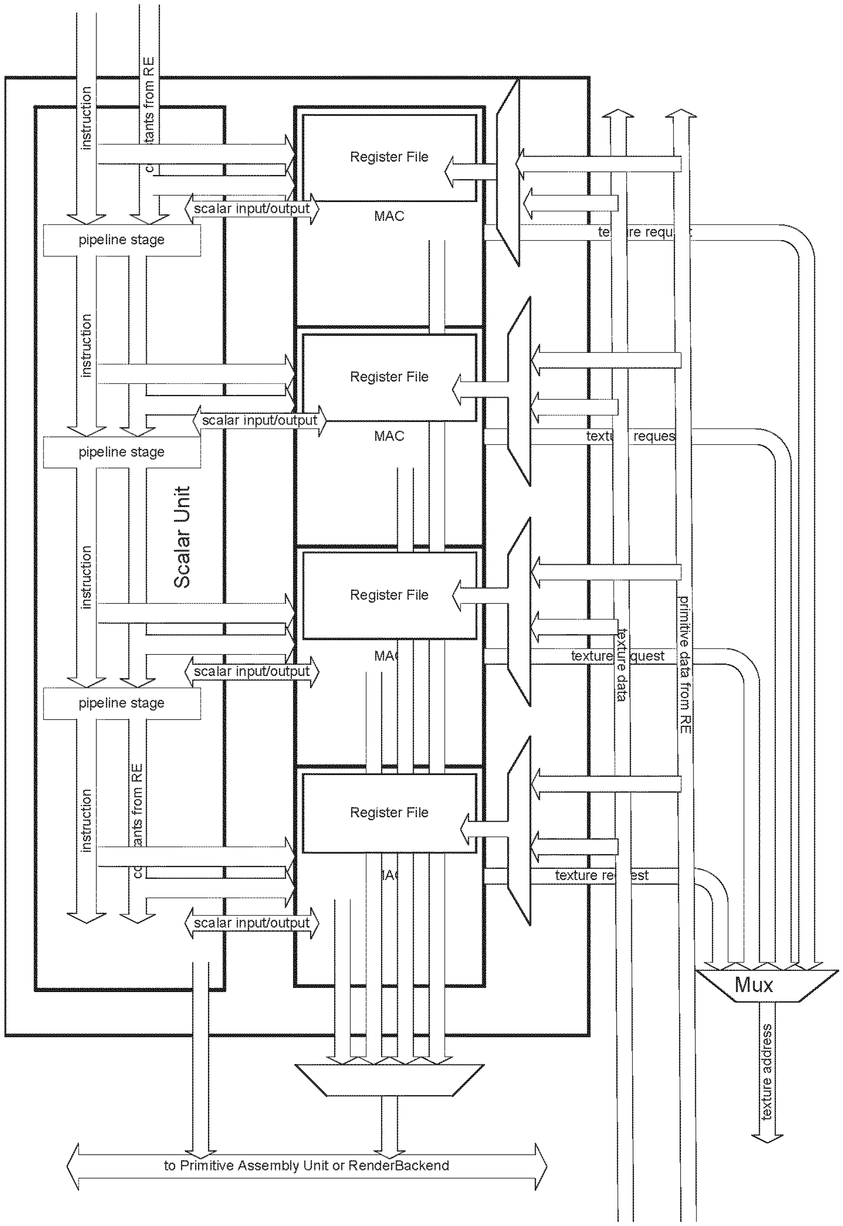
If the packet is a vertex packet, upon reaching ALU clause 3~~4~~, it can export the position if the position is ready. So the arbitrer must prevent ALU clause 3~~4~~ to be selected if the positional buffer is full (or can't be accessed). Along with the positional data, the location where the vertex data is to be put is also sent (parameter data pointers).

A special case is for HOS surfaces wich can export 12 parameters per clause 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.

All other level process in the same way until the packet finally reaches the last ALU machine (8). On completion of the level 8 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.

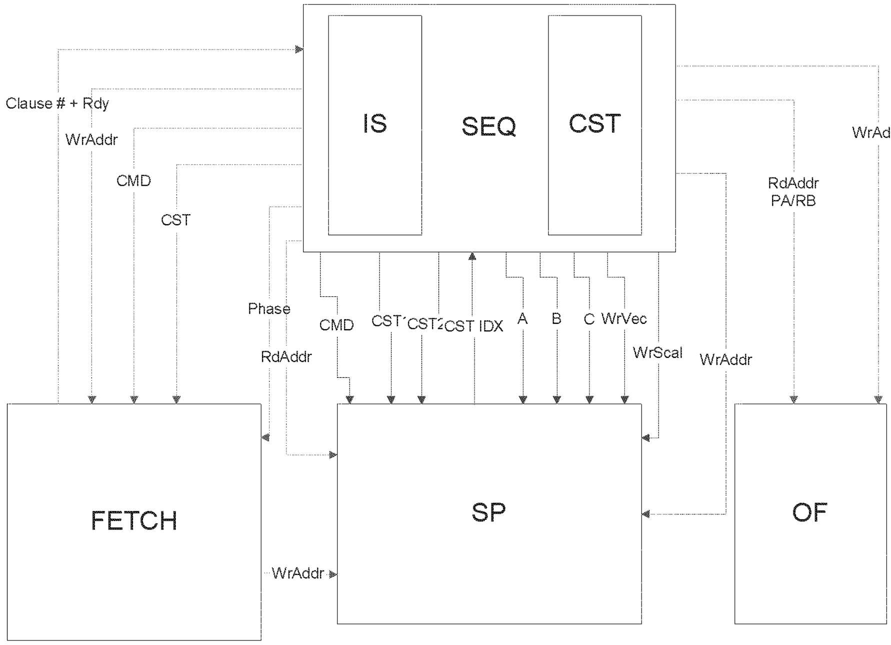
Only two ALU state machine may have access to the register file address bus or the instruction decode bus at one time. Similarly, only one fetch state machine may have access to the register file address bus at one time. Arbitration is performed by three arbitrer blocks (two for the ALU state machines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

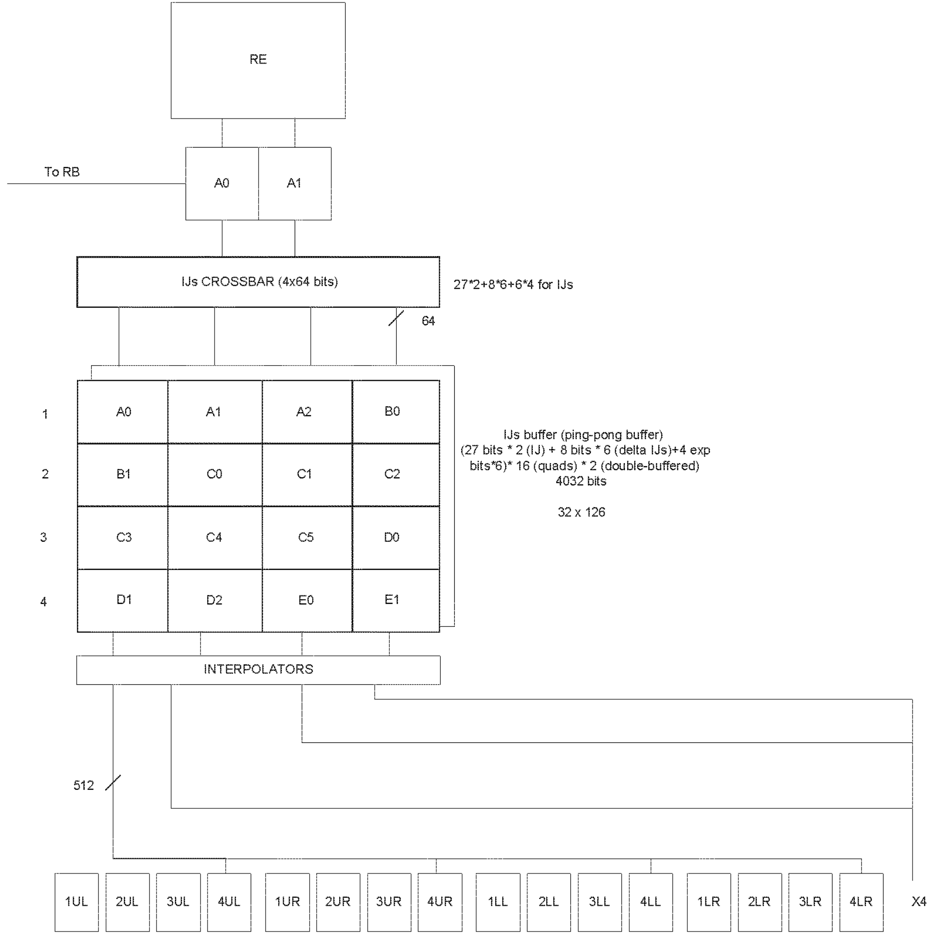
1.3 Control Graph



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)
27*2+8*6+6*4 for IJs
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)
(27 bits * 2 (IJ) + 8 bits * 6 (delta IJs)+4 exp bits*6)* 16 (quads) * 2 (double-buffered)
4032 bits
32 x 126
INTERPOLATORS
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
SP0	A0	B1	C3	D1					A0	B1	C3	D1					V 0-3	V 16- 19	V 32- 35	V 48- 51
SP1	A1		C4	D2		C0			A1		C4	D2		C0			V 4-7	V 20- 23	V 36- 39	V 52- 55
SP2	A2		C5			C1		E0	A2		C5			C1		E0	V 8-11	V 24- 27	V 40- 43	V 56- 59
SP3					B0	C2	D0	E1					B0	C2	D0	E1	V 12- 15	V 28- 31	V 44- 47	V 60- 63
				P0									P1							

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.

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 Store

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 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.

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. ~~The control program has 4(5) instructions:~~

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 (one for constant,registert)
Loop_Step[7:0] [15:0] times 2 (one for constant,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.

We also keep 8 predicate vectors and 8 AND/OR sets of 3 bits. These bits can be 0: all 0s, 1: all ones and 11: mixed.

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_execute_or_Jump
Conditional_jump
Call
Return
Loop_start
Loop_end
End_of_clause

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 4096 we break the program 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.

~~Conditionnal_Execute_or_Jump~~NOP		
47	46 … 42	~~41 … 34~~ ~~33~~ ~~32 … 21~~ ~~20 … 12~~ ~~11 … 0~~41 … 0
Addressing	00010	~~Booleans~~ ~~Condition~~ ~~Jump address~~ ~~Instruction_count~~ ~~Exec Address~~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.

Conditionnal_Execute						
47	46 … 42	41 … 34	33	32 … 24~~1~~	2~~0~~3 … 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~~512~~ instructions)

Conditionnal_Execute_Predicates						
47	46 … 42	41 … 38	37	36 … 24~~1~~	2~~0~~3 … 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.

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 … 12	11 … 0
Addressing	01010	Boolean address	Condition	RESERVED	Address

If condition met, jumps to the address. FORWARD jump only allowed.

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 is going to break the loop and set de debug registers. The sequencer will keep ~~a~~ two loop indexes values~~ of 17 bits~~:

IC index for constant indexing (9 bits)

IR index for register file indexing (7 bits)

~~.~~ This will be updated everytime 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 exeeds 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

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 wich 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 exemple, 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 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.

Formatted: Bullets and Numbering

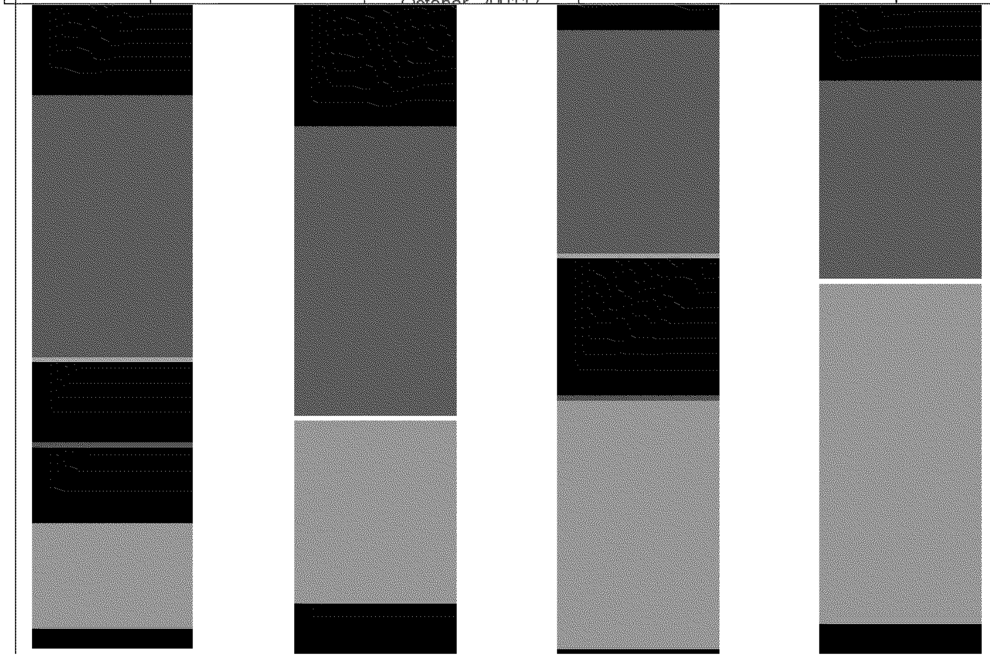
7. HOS surfaces

HOS surfaces are able to export from any clause 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. The buffer they want to export into must be specified in the "exports" field of the state registers.

Formatted: Bullets and Numbering

~~7.~~8. Register file allocation

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

Formatted: Bullets and Numbering

~~8.~~9. 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.

Formatted: Bullets and Numbering

~~9.~~10. 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.

Formatted: Bullets and Numbering

~~10.~~11. 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?). The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU arbitrer will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

Formatted: Bullets and Numbering

~~11.~~12. Content of the reservation station FIFOs

~~3~~ 21 bits of Render State ~~6~~ 77 bits for the base address of the ~~instruction~~ GPRs~~store~~, 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. ~~Every other information (such as the coverage mask, quad address, etc.) is put in a FIFO and is retrieved when the quad exits the shader pipe to enter in the output file buffer.~~ 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.

For texture clauses, 3 bits * 4 are going to be kept. These are the AND/OR of the predicate vectors. 0 for all 0s, 1 for all ones and MIXED.

Formatted: Bullets and Numbering

~~12.~~13. The Output File

The output file is where pixels are put before they go to the RBs. The write BW to this store is 256 bits/clock. Just before this output file are staging registers with write BW 512 bits/clock and read BW 256 bits/clock. 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).

Formatted: Bullets and Numbering

~~13.~~14. IJ Format

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

We have a vector of IJ's (one IJ per pixel at the centroid of the fragment or at the center of the pixel depending on the mode bit). The interpolation is done at a different precision across the 2x2. The upper left pixel's parameters are always interpolated at full 19x24 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 19x24 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 19 Exp 4 for I + Sign
Mantissa 19 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 = 126

Formatted: Bullets and Numbering

~~14.~~15. 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.

Formatted: Bullets and Numbering

~~15.~~16. Vertex position exporting

On clause 4 (or 5) the vertex shader can export to the PA both the vertex position and the point sprite. It can also do so at clause 8 if not done at clause 4. 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.

Formatted: Bullets and Numbering

~~16.~~17. 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.

~~17.~~18. Registers

Formatted: Bullets and Numbering

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)
Vshader_fetch[11:0][7:0]	eight 12 bit pointers to the location where each clauses control program is located
Vshader_alu[11:0][7:0]	eight 12 bit pointers to the location where each clauses control program is located
Pshader_fetch[11:0][7:0]	eight 12 bit pointers to the location where each clauses control program is located
Pshader_alu[11:0][7:0]	eight 12 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
VWRAP	wrap point for the vertex shader instruction store
PWRAP	wrap point for the pixel shader instruction store
REG_ALLOC_PIX	number of registers to allocate for pixel shader programs
REG_ALLOC_VERT	number of registers to allocate for vertex shader programs
PARAM_MASK[0…16]	parameter mask to specify how parameters maps in the pixel shader
FLAT_GOUR[0…16]	wich parameters are to be gouraud shaded
GEN_TEX[0….16]	for wich parameters do we need to generate tex coords.
CYL_WRAP[0…64]	for wich parameters (and channels (xyzw)) do we do the cyl wrapping.
P_EXPORT[8]	number of exports for pixel shader
V_EXPORT[8]	number of exports for vertex shader for each clause. All numbers relate to the output buffer exports but for V_EXPORT[7] than can relate to the PC if Exports[7] is set to 00000.~~(also the number of interpolated parameters for pixel shaders)~~
~~V_EXPORT_LOC~~	~~Vertex shader exporting to RB or the PCACHE~~
ARBITRATION_policy	policy of the arbitration between vetexes and pixels
Exports[8][6]	Wich clause is exporting to the output buffer and what is it exporting.
	000000 : Not exporting (or exporting only to the PC)
	000001 : Exporting position (1)
	000010 : Exporting position (2)
	(1)00100 : Exporting RG
	(1)01000 : Exporting BA
	(1)10000 : Exporting Z
	If MSB set pixel shader exporting linear to memory not to Frame Buffer.
CST_SIZE_P	Size of the constant store for pixels
CST_SIZE_V	Size of the constant store for vertexes

Formatted: Bullets and Numbering

19. DEBUG registers

PROB_ADDR	instruction address where the first problem occurred
PROB_COUNT	number of problems encountered during the execution of the program

Formatted: Bullets and Numbering

~~18.~~20. Interfaces

~~18.1~~20.1 External Interfaces

~~18.1.1~~20.1.1 *PA/SC to RE : 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

Name	Direction	Bits	Description
IJs	PA→RE	63	IJ information sent over 2 clocks
Mask	PA→RE	1	Write Mask

~~18.1.2~~20.1.2 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.

Name	Direction	Bits	Description
Write Mask	PA→SEQ(RE)	4	Quad Write mask left to right
RB_ID	PA→SEQ(RE)	8	RB id for each quad sent 2 bits per quad
LOD_CORRECT	PA→SEQ(RE)	24	LOD correction per quad (6 bits per quad)
FVTX	PA→SEQ(RE)	2	Provoking vertex for flat shading
PPTR0	PA→SEQ(RE)	11	P Store pointer for vertex 0
PPRT1	PA→SEQ(RE)	11	P Store pointer for vertex 1
PPTR2	PA→SEQ(RE)	11	P Store pointer for vertex 2
E_OFF_VECTOR	PA→SEQ(RE)	1	End of the vector
DEALLOC	PA→SEQ(RE)	1	Deallocation token for the P Store
STATE	PA→SEQ(RE)	21	State/constant pointer (6*3+3)
VALID	PA→SEQ(RE)	16	Valid bits for all pixels
NULL	PA→SEQ(RE)	1	Null Primitive (for PC deallocation purposes)
E_OFF_PRIM	PA→SEQ(RE)	1	End Of the primitive
FBFACE	PA→SEQ(RE)	1	Front face = 1, back face = 0
TYPE	PA→SEQ(RE)	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
RTRn	SEQ→PA	1	Stalls the PA in n clocks
RTS	PA→SEQ(RE)	1	PA ready to send data
QuadX	PA→SEQ(RE)	8	Quad X address 2 bits per quad
QuadY	PA→SEQ(RE)	8	Quad Y address 2 bits per quad

~~18.1.3~~20.1.3 VGT to RE : Vertex Bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Vertex indexes	VGT→RE	128	Pointers of indexes or HOS surface information
EOF_vector	VGT→RE	1	End of the vector
Inputs_vert	VGT→RE	1	0: Normal 128 bits per vert 1: double 256 bits per vert
STATE	VGT→SEQ	21	Render State (6*3+3 for constants)

~~18.1.4VGT to SEQ : Vertex Control Bus~~

Formatted: Bullets and Numbering

~~This information needs to be sent over 64 clocks.~~

~~18.1.5~~20.1.4 CP to SEQ : Constant store load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Constant Address	CP→SEQ	8	Address of the block of 4 constants
Constant Data	CP→SEQ	512	Data sent over 4 clocks
Remap Address	CP→SEQ	10	Remaping address write address
Remap Data pointer	CP→SEQ	8	Remaping pointer

~~18.1.6~~20.1.5 CP to SEQ : Fetch State store load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Constant Address	CP→SEQ	8	Address of the block of 4 state constants

Constant Data	CP→SEQ	512	Data sent over 4 clocks
Remap Address	CP→SEQ	10	Remaping address write address
Remap Data pointer	CP→SEQ	8	Remaping pointer

~~18.1.7~~20.1.6 *CP to SEQ : Control State store load*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description

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

~~18.1.8~~20.1.7 *MH to SEQ: Instruction store Load*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Instruction address	MH→SEQ	12	Instruction address
Instruction	MH→SEQ	96	Instruction X times
Control Instruction address	MH→SEQ	9	Pointer to the control instruction store
Control Instruction	MH→SEQ	32	Control Instruction X times

~~18.1.9~~20.1.8 *SP to RB : Pixel read from RBs*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Export_data	SP→RB	64	a pair of 32 bits channel values
ExportID	SP→RB	9	0cvvvvhqq: Vertex data vvvv 0-15 from first or second clause (c=0 or 1), XY or ZW components (h=0 or 1), quad 0-3 in the shader (qq= 0-3) 1cbbkttqq: Pixel data for buffer bb (0-3) from first or second clause (0-1) killed or not (k=1 or 0) quad 0-3 in the shader and data is RG (tt=0), BA (tt=1) or Z (tt=2)
ExportMask	SP→RB	2	Specifies whether to write low, high or both 32 bit words. If export mask is 00 data is invalid
ExportLast	SP→RB	1	Last export instruction of the clause

~~18.1.10~~20.1.9 *SEQ to RB : Control bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Type	SEQ→RB	1	0: Pixel 1: Vertex
Interleaving	SEQ→RB	1	0: first interleaved clause 1: second interleaved clause
Export_size	SEQ→RB	4	0 thru 16 parameters exported for vertexes (vvvv) OR (bbzs) 1-4 color buffers (bb), two component (s=0) or 4 component colors (s=1) with z (z=1) or without z (z=0)
Valid	SEQ→RB	1	Data valid

Only one exporting clause (7) can be selected at any given time.

~~18.1.11~~20.1.10 *RB to SEQ : Output file control*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Buff_Full	RB→SEQ	1	Set if full
Avail_size	RB→SEQ	6	Size available in output buffers (in 32bits increments)

~~18.1.12~~20.1.11 *SP to RB : Position return bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Position return	SP→RB	128	Position data or sprite size (per clock)
Parameter cache pointer	SP→RB	11	Pointer where the data will be in the parameter cache for each vertex

For point sprites and position exports the size and position are interleaved on a 16 x 16 basis. We export 1 position then 1 point sprite sizes. The storage used is of 64x128 bits for position and 64x32 bits for sprite size, it is taken from the output buffer. Additionnally,if needed the edge flags are packed into the bits of the sprite sizes.

~~18.1.13~~20.1.12 *Shader Engine to Fetch Unit Bus (Fast 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
Tex_Read_Register_Index	SEQ->SP	7	Index into Register files for reading Fetch Address
Tex_RegFile_Read_Data	SP->TEX	2048	16 Fetch Addresses read from the Register file
Tex_Write_Register_Index	SEQ->TEX	7	Index into Register file for write of returned Fetch Data

~~18.1.14~~20.1.13 *Sequencer to Fetch Unit bus (Slow Bus)*

Formatted: Bullets and Numbering

Once every four clock, the fetch unit sends to the sequencer on wich 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 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	TEX→ SEQ	1	Data ready
Tex_Clause_Num	TEX→ SEQ	3	Clause number
Tex_cst	SEQ→TEX	10	Fetch state address 10 bits sent over 4 clocks
Tex_Inst	SEQ→TEX	12	Fetch instruction address 12 bits sent over 4 clocks
EO_CLAUSE	SEQ→TEX	1	Last instruction of the clause
PHASE	SEQ→TEX	1	Write phase signal

~~19.~~21. Internal interfaces

Formatted: Bullets and Numbering

21.1.1 *RE to SEQ : Vertex Control Bus*

Name	Direction	Bits	Description
STATE	VGT→SEQ	21	Render State (6*3+3 for constants)
Vert counter	VGT→SEQ	6	Which vertices are valid
Inputs_vert	VGT→SEQ	1	0: Normal 128 bits per vert 1: double 256 bits per vert

This information needs to be sent over 64 clocks.

~~20.~~22. Examples of program executions

Formatted: Bullets and Numbering

~~20.1.1~~22.1.1 *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 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

~~20.1.2~~22.1.2 *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 (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

~~20.1.3~~22.1.3 *Notes*

Formatted: Bullets and Numbering

14. the state machines and arbitrers 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, parameter buffer allocation, loops and branches and parameter cache de-allocation still needs to be specked out.

~~21.~~23. 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 bandwith 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 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. 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?

Size of the fifo containing the information of a vector of pixels/vertices. And size of the fifos before the reservation stations.

Loops and branches.

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 2015~~26 October, 2001~~19	GEN-CXXXXX-REVA	1 of 28

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SEQ

Version ~~0.9~~1.0

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

AUTOMATICALLY UPDATED FIELDS:
Document Location: C:\perforce\r400\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.

ATI 2016
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.

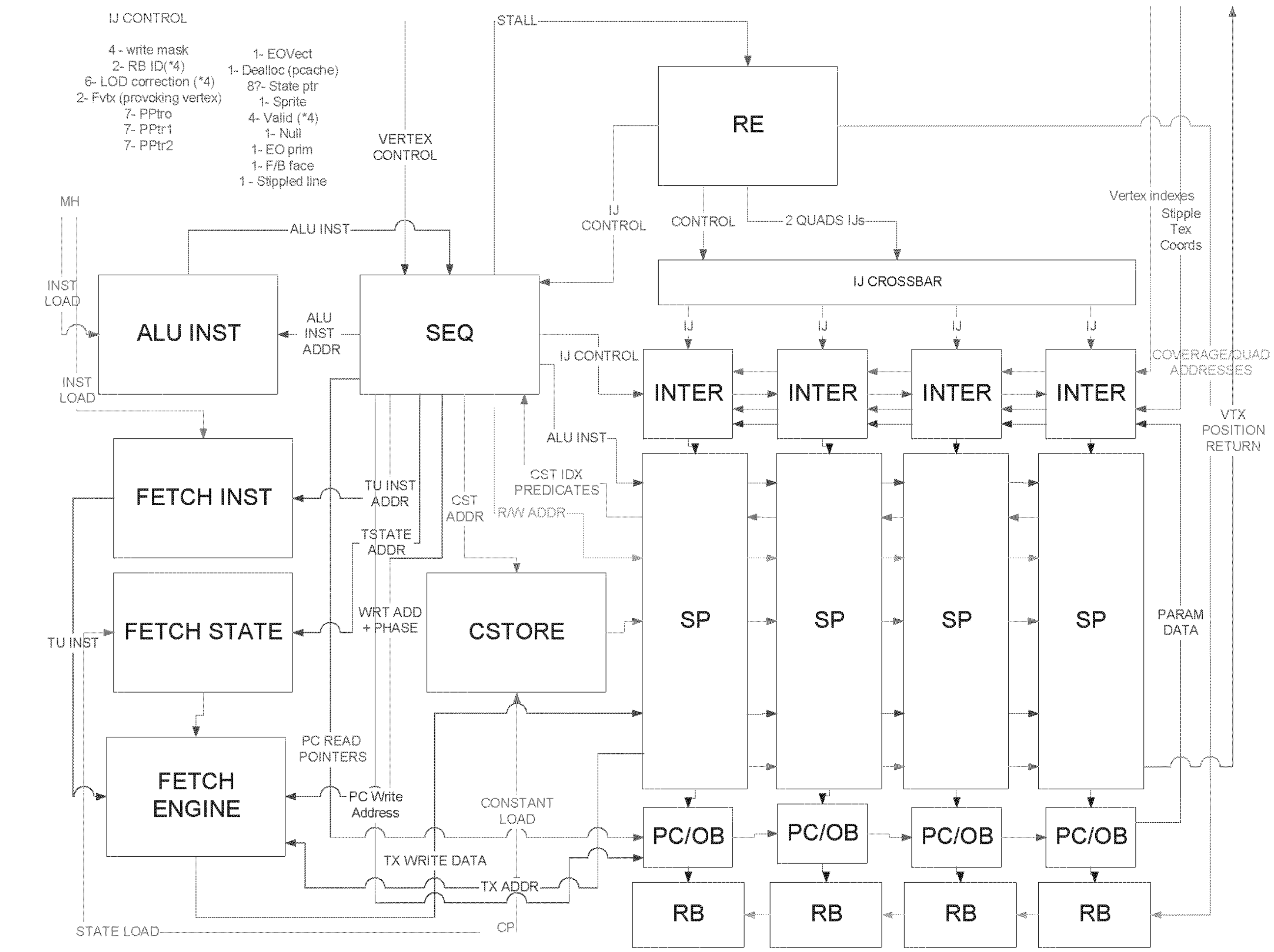
1. Overview

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.

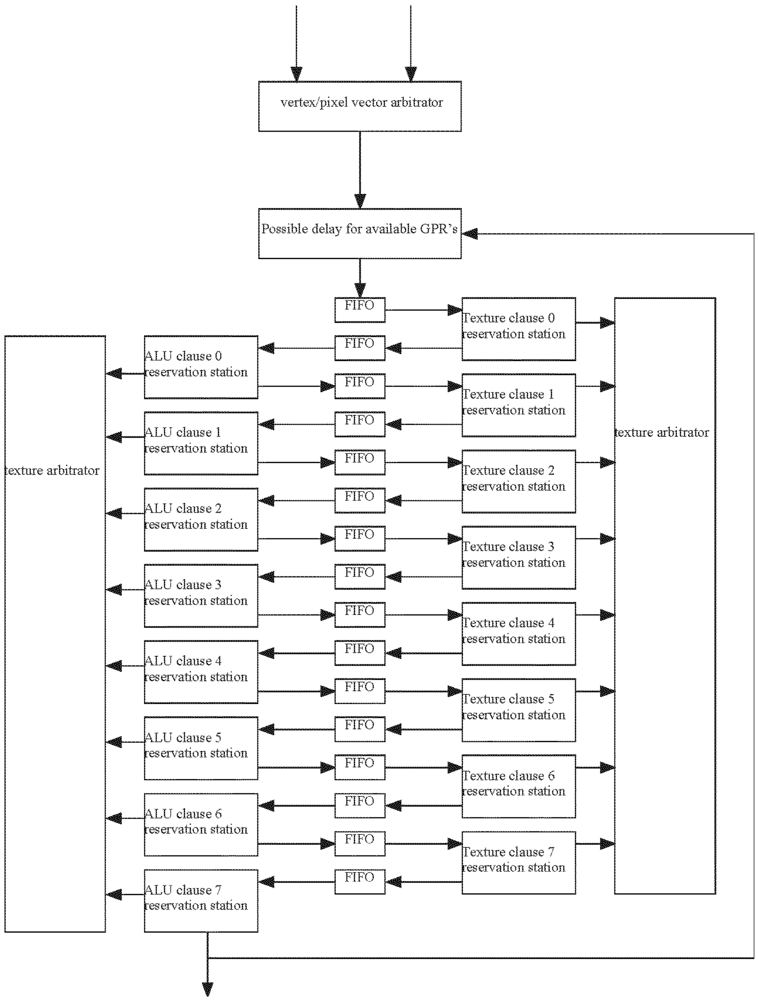
The sequencer is based on the R300 design. It chooses two ALU clauses and a fetch clause to execute, and executes all of the instructions in a clause before looking for a new clause of the same type. Two ALU clauses are executed interleaved to hide the ALU latency. Each vector will have eight fetch and 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 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
MH
ALU INST
IJ CONTROL
CONTROL
2 QUADS IJs
Vertex indexes
Stipple
Tex
Coords
IJ CROSSBAR
INST LOAD
ALU INST
ALU INST ADDR
SEQ
IJ
IJ
IJ
IJ
IJ CONTROL
INTER
INTER
INTER
INTER
COVERAGE/QUAD ADDRESSES
INST LOAD
ALU INST
VTX POSITION RETURN
FETCH INST
TU INST ADDR
CST IDX PREDICATES
CST ADDR
R/W ADDR
TSTATE ADDR
SP
SP
SP
SP
PARAM DATA
TU INST
FETCH STATE
WRT ADD + PHASE
CSTORE
FETCH ENGINE
PC READ POINTERS
PC Write Address
CONSTANT LOAD
PC/OB
PC/OB
PC/OB
PC/OB
TX WRITE DATA
TX ADDR
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.

~~The rasterizer always checks the vertices FIFO first and if allowed by the sequencer sends the data to the shader. If the vertex FIFO is empty then, the rasterizer takes the first entry of the pixel FIFO (a vector of 64 pixels) and sends it to the interpolators. Then the sequencer takes control of the packet~~Depending on the arbitration state, the sequencer will either choose a vertex or a pixel packet. The control packet consists of 21 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 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). One arbitrer 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 is that they are not allowed to pick the same clause number as the~~y~~ other one is currently working on if the packet ~~os~~ 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 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)?}

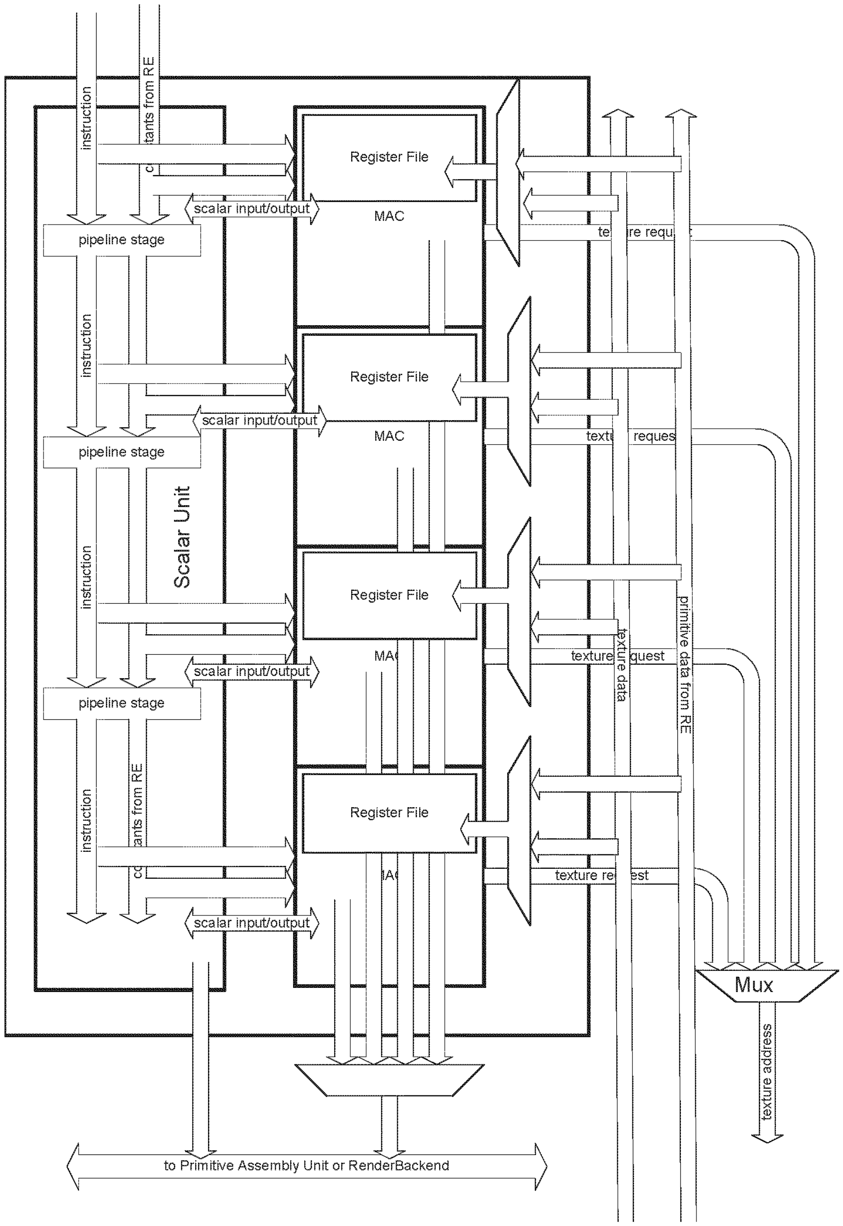
A special case is for HOS surfaces wich can export 12 parameters per last 6 clauses to the output buffer. If the output buffer is full or doesn't have enough space the sequencer will prevent such a vertex group to enter an exporting clause.

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 (~~8~~7). On completion of the level ~~8~~ 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.

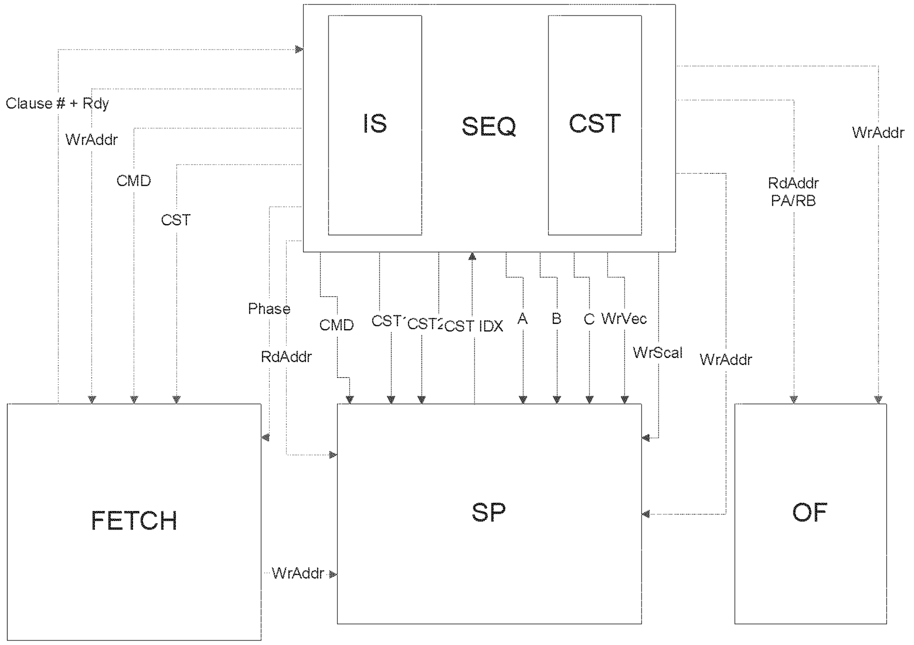
Only two ALU state machine may have access to the register file address bus or the instruction decode bus at one time. Similarly, only one fetch state machine may have access to the register file address bus at one time. Arbitration is performed by three arbitrer blocks (two for the ALU state machines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

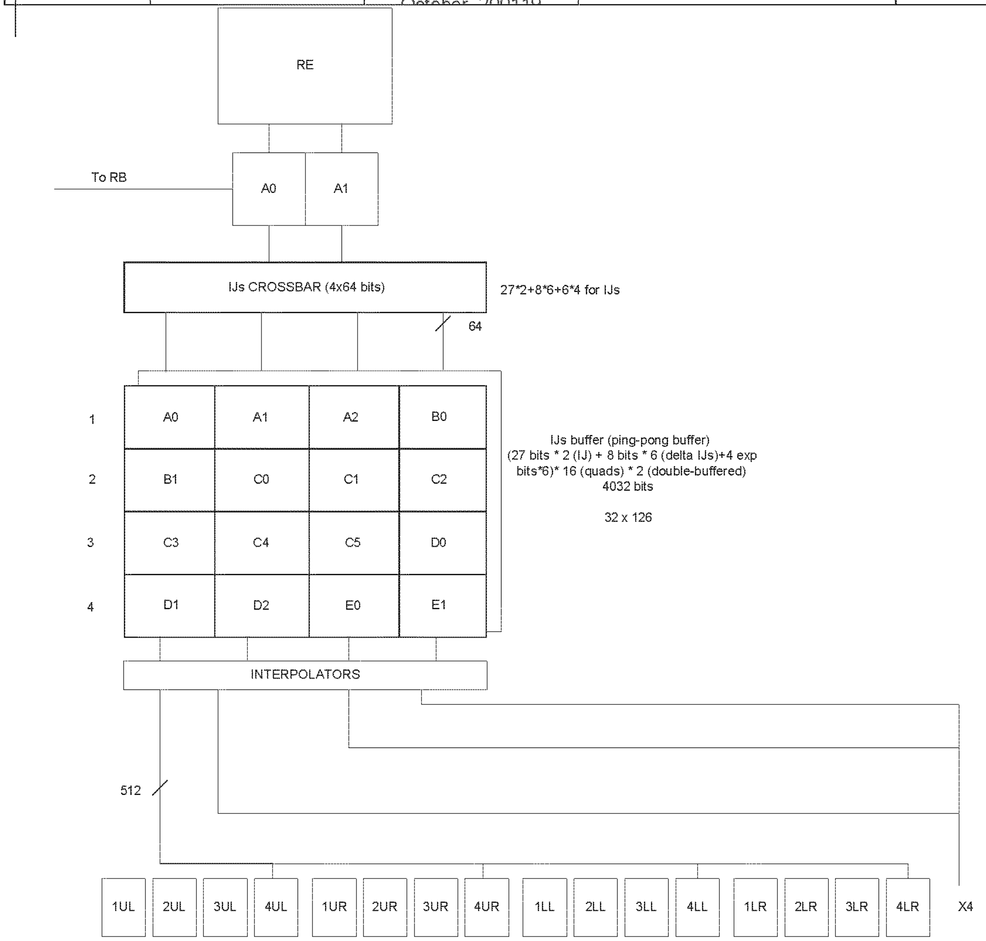
1.3 Control Graph



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.



1	A0	A1	A2	B0
2	B1	C0	C1	C2
3	C3	C4	C5	D0
4	D1	D2	E0	E1

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19
SP0	A0	B1	C3	D1					A0	B1	C3	D1					V 0-3	V 16-19	V 32-35	V 48-51
SP1	A1		C4	D2		C0			A1		C4	D2		C0			V 4-7	V 20-23	V 36-39	V 52-55
SP2	A2		C5			C1		E0	A2		C5			C1		E0	V 8-11	V 24-27	V 40-43	V 56-59
SP3					B0	C2	D0	E1					B0	C2	D0	E1	V 12-15	V 28-31	V 44-47	V 60-63
				P0									P1							

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.

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 Store

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 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.

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 (one for constant,registert)
Loop_Step[7:0] [15:0] times 2 (one for constant,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.

We also keep 8 predicate vectors and 8 AND/OR sets of 3 bits. These bits can be 0: all 0s, 1: all ones and 11: mixed.

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_execute_or_Jump
Conditional_jump
Call
Return
Loop_start
Loop_end
End_of_clause

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 ~~4096~~ 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.

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.

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~~2~~	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 is going to break the loop and set de 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 everytime 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 exeeds 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
```

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 wich 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 exemple, the ~~instruction~~ :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 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.

Formatted: Bullets and Numbering

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

However, if the driver sets the kill_vector_on register to 0 (don't use) then the 64 bit kill mask becomes the 5th predicate vector and is kept across clause boundaries (thus allowing predicated instructions to be used in texture clauses). In this mode, the sequencer is going to send all 1s to the RBs for coverage mask information.

Formatted

Formatted: Bullets and Numbering

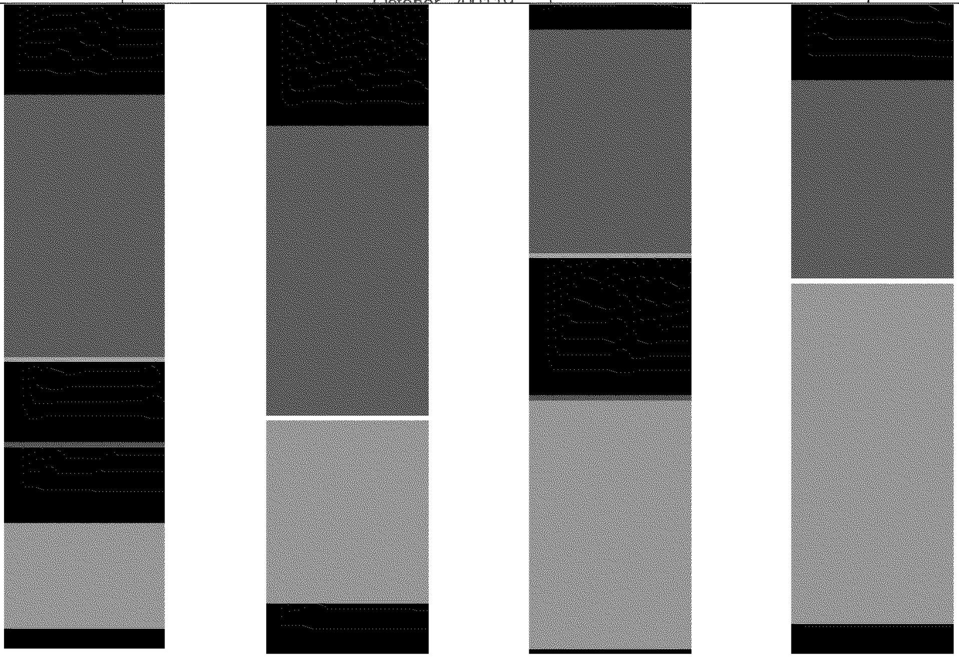
~~7.~~8. HOS surfaces

HOS surfaces are able to export from ~~any~~ 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. The buffer they want to export into must be specified in the "exports" field of the state registers.

Formatted: Bullets and Numbering

~~8.~~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.

Formatted: Bullets and Numbering

~~9.~~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.

Formatted: Bullets and Numbering

~~10.~~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.

Formatted: Bullets and Numbering

~~11.~~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?). The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU arbitrer will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

Formatted: Bullets and Numbering

~~12.~~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.

~~For texture clauses, 3 bits * 4 are going to be kept. These are the AND/OR of the predicate vectors. 0 for all 0s, 1 for all ones and MIXED.~~

Formatted: Bullets and Numbering

~~13.~~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).

Formatted: Bullets and Numbering

~~14.~~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 19x24 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 19x24 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 19 Exp 4 for I + Sign
Mantissa 19 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 = 126

Formatted: Bullets and Numbering

~~15.~~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.

Formatted: Bullets and Numbering

~~16.~~17. Vertex position exporting

On clause 4 (or 5) the vertex shader can export to the PA both the vertex position and the point sprite. It can also do so at clause 8 if not done at clause 4. 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.

Formatted: Bullets and Numbering

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.
~~1)~~9)Parameter exports and memory exports CAN be co-issued.

Formatted: Bullets and Numbering

~~17.~~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 reatime parameter memory as well as the regular parameter store. For higher performance we should be able able to view them as two banks of 16 and do double buffering allowing one to be loaded, while the other is rasterized with. Most overlay shaders will need 2 or 4 scalar coordinates, one option might be to restrict the memory to 16x64 or 32x64 allowing only two interpolated scalars per cycle, the only problem I see with this is, if we view support for 16 vector-4 interpolants important (true only if we map microsoft's high priority stream to the realtime stream), then the PA/sequencer need to support a realtime-specific mode where we need to address 32 vectors of parameters instead of 16.

~~18.~~20. Registers

Formatted: Bullets and Numbering

20.1 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 vetexes 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
VWRAP	wrap point for the vertex shader instruction store
PWRAP	wrap point for the pixel shader instruction store
NO_INTERLEAVE	debug state register. Only allows one program at a time into the GPRs

~~18.1~~20.2 Context

Formatted: Bullets and Numbering

Vshader_fetch[~~11~~7:0][7:0]	eight ~~12~~ 8 bit pointers to the location where each clauses control program is located
Vshader_alu[~~11~~7:0][7:0]	eight ~~12~~ 8 bit pointers to the location where each clauses control program is located
Pshader_fetch[~~11~~7:0][7:0]	eight ~~12~~ 8 bit pointers to the location where each clauses control program is located
Pshader_alu[~~11~~7:0][7:0]	eight ~~12~~ 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)
~~VWRAP~~	~~wrap point for the vertex shader instruction store~~
~~PWRAP~~	~~wrap point for the pixel shader instruction store~~
REG_ALLOC_PIX	number of registers to allocate for pixel shader programs
REG_ALLOC_VERT	number of registers to allocate for vertex shader programs
~~PARAM_MASK[0…16]~~	~~parameter mask to specify how parameters maps in the pixel shader~~
FLAT_GOUR[0…1~~6~~5]	wich parameters are to be gouraud shaded
GEN_TEX~~[0….16]~~	~~for wich parameters do we need to generate tex coords.~~Do we generate texture coordinates for 1st parameter or not
CYL_WRAP[0…~~64~~63]	for wich 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	wich 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)
kill_vector_on	use the mask kill vector to kill pixels and optimize texture pipe fetches OR use it as the fifth predicate vector wich is the only predicate vector kept across clause boundaries.
~~P_EXPORT[8]~~	~~number of exports for pixel shader~~
~~V_EXPORT[8]~~	~~number of exports for vertex shader for each clause. All numbers relate to the output buffer exports but for V_EXPORT[7] than can relate to the PC if Exports[7] is set to 00000.~~
~~ARBITRATION_policy~~	~~policy of the arbitration between vetexes and pixels~~
	~~Exports[8][6] Wich clause is exporting to the output buffer and what is it exporting.~~
	~~000000 : Not exporting (or exporting only to the PC)~~
	~~000001 : Exporting position (1)~~
	~~000010 : Exporting position (2)~~
	~~(1)00100 : Exporting RG~~

Formatted

~~(1)01000 : Exporting BA~~
~~(1)10000 : Exporting Z~~
~~If MSB set pixel shader exporting linear to memory not to Frame Buffer.~~
~~CST_SIZE_P Size of the constant store for pixels~~
~~CST_SIZE_V Size of the constant store for vertexes~~

Formatted: Bullets and Numbering

~~19.~~21. DEBUG registers

PROB_ADDR	instruction address where the first problem occurred
PROB_COUNT	number of problems encountered during the execution of the program

Formatted: Bullets and Numbering

~~20.~~22. Interfaces

~~20.1~~22.1 External Interfaces

~~20.1.1~~22.1.1 *PA/SC to ~~RE~~ SP0 : 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	PA→~~RE~~SP0	63	IJ information sent over 2 clocks
Mask	PA→~~RE~~SP0	1	Write Mask

Formatted: Bullets and Numbering

~~20.1.2~~22.1.2 *PA/SC to SEQ : 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.

Name	Direction	Bits	Description
Write Mask	PA→SEQ(~~RE)~~SP)	4	Quad Write mask left to right
RB_ID	PA→SEQ(SP)~~PA→SEQ(RE)~~	8	RB id for each quad sent 2 bits per quad
LOD_CORRECT	PA→SEQ(SP)~~PA→SEQ(RE)~~	24	LOD correction per quad (6 bits per quad)
FVTX	PA→SEQ(SP)~~PA→SEQ(RE)~~	2	Provoking vertex for flat shading
PPTR0	PA→SEQ(SP)~~PA→SEQ(RE)~~	11	P Store pointer for vertex 0
PPRT1	PA→SEQ(SP)~~PA→SEQ(RE)~~	11	P Store pointer for vertex 1
PPTR2	PA→SEQ(SP)~~PA→SEQ(RE)~~	11	P Store pointer for vertex 2
E_OFF_VECTOR	PA→SEQ(SP)~~PA→SEQ(RE)~~	1	End of the vector
DEALLOC	PA→SEQ(SP)~~PA→SEQ(RE)~~	1	Deallocation token for the P Store
STATE	PA→SEQ(SP)~~PA→SEQ(RE)~~	21	State/constant pointer (6*3+3)
VALID	PA→SEQ(SP)~~PA→SEQ(RE)~~	16	Valid bits for all pixels
NULL	PA→SEQ(SP)~~PA→SEQ(RE)~~	1	Null Primitive (for PC deallocation purposes)
E_OFF_PRIM	PA→SEQ(SP)~~PA→SEQ(RE)~~	1	End Of the primitive
FBFACE	PA→SEQ(SP)~~PA→SEQ(RE~~	1	Front face = 1, back face = 0

	~~)~~		
TYPE	PA→SEQ(SP)~~PA→SEQ(RE)~~	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
RTRn	SEQ→PA	1	Stalls the PA in n clocks
RTS	PA→SEQ(SP)~~PA→SEQ(RE)~~	1	PA ready to send data

~~20.1.3~~22.1.3 VGT to ~~RE~~ SP : Vertex Bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Vertex indexes	VGT→RE	128	Pointers of indexes or HOS surface information
EOF_vector	VGT→RE	1	End of the vector
Inputs_vert	VGT→RE	1	0: Normal 128 bits per vert 1: double 256 bits per vert
STATE	VGT→SEQ	21	Render State (6*3+3 for constants)

~~20.1.4~~22.1.4 CP to SEQ : Constant store load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Constant Address	CP→SEQ	8	Address of the block of 4 constants
Constant Data	CP→SEQ	512	Data sent over 4 clocks
Remap Address	CP→SEQ	10	Remaping address write address
Remap Data pointer	CP→SEQ	8	Remaping pointer

~~20.1.5~~22.1.5 CP to SEQ : Fetch State store load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Constant Address	CP→SEQ	8	Address of the block of 4 state constants
Constant Data	CP→SEQ	512	Data sent over 4 clocks
Remap Address	CP→SEQ	10	Remaping address write address
Remap Data pointer	CP→SEQ	8	Remaping pointer

~~20.1.6~~22.1.6 CP to SEQ : Control State store load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description

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

~~20.1.7~~22.1.7 MH to SEQ: Instruction store Load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Instruction address	MH→SEQ	12	Instruction address
Instruction	MH→SEQ	96	Instruction X times
Control Instruction address	MH→SEQ	9	Pointer to the control instruction store
Control Instruction	MH→SEQ	32	Control Instruction X times

{ISSUE: CP or MH?}

~~20.1.8~~22.1.8 SP to RB : Pixel read from RBs

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Export_data	SP→RB	64*16	32a pairs of 32 bits channel values
~~ExportID~~Shader_Dest	SP→RB	~~9~~4	Specifies one of the of up to 12 export destinations~~0cvvvvhqq: Vertex data vvvv 0-15 from first or second clause (c=0 or 1), XY or ZW components (h=0~~

			~~or 1), quad 0-3 in the shader (qq= 0-3)~~ ~~1cbbkttqq: Pixel data for buffer bb (0-3) from first or second clause (0-1) killed or not (k=1 or 0) quad 0-3 in the shader and data is RG (tt=0), BA (tt=1) or Z (tt=2)~~
Shader_Count~~ExportMask~~	SP→RB	~~2~~3	Each set of four pixels or vectors is exported over eight clocks. This field specifies where the SP is in that sequence.~~Specifies whether to write low, high or both 32 bit words. If export mask is 00 data is invalid~~
Shader_Last~~ExportLast~~	SP→RB	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.~~Last export instruction of the clause~~
Shader_PixelValid	SP→RB	4x4	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
Shader_WordValid	SP→RB	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

~~20.1.9~~22.1.9 SEQ to RB : Control bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Export_Pixel~~Type~~	SEQ→RB	1	1~~0~~: Pixel ~~1~~0: Vertex
Export_SEND~~Interleaving~~	SEQ→RB	1	Raised to indicate that the SQ is starting an export~~0: first interleaved clause~~ ~~1: second interleaved clause~~
Export_Clause~~Export_size~~	SEQ→RB	~~4~~3	Clause number, which is needed for vertex clauses~~0 thru 16 parameters exported for vertexes (vvvv) OR (bbzs) 1-4 color buffers (bb), two component (s=0) or 4 component colors (s=1) with z (z=1) or without z (z=0)~~
Export_State~~Valid~~	SEQ→RB	~~1~~21?	State ID, which is needed for vertex clauses~~Data valid~~

These fields are sent synchronously with SP export data, described in SP→RB interface

{ISSUE: Where are the PC pointers}~~Only one exporting clause (7) can be selected at any given time.~~

~~20.1.10~~22.1.10 RB to SEQ : Output file control

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Export_RTS~~Buff_Full~~	RB→SEQ	1	Raised by RB to indicate that the following two fields reflect the result of the most recent export~~Set if full~~
Export_Position~~Avail_size~~	RB→SEQ	16	Specifies whether there is room for another position.~~Size available in output buffers (in 32bits increments)~~
Export_Buffer	RB→SEQ	7	Specifies the space availble 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

~~20.1.11~~22.1.11 SP to RB : Position return bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Position return	SP→RB	128	Position data or sprite size (per clock)
Parameter cache pointer	SP→RB	11	Pointer where the data will be in the parameter cache for each vertex

For point sprites and position exports the size and position are interleaved on a 16 x 16 basis. We export 1 position then 1 point sprite sizes. The storage used is of 64x128 bits for position and 64x32 bits for sprite size, it is taken from the output buffer. Additionnally,if needed the edge flags are packed into the bits of the sprite sizes.

~~20.1.12~~22.1.12 Shader Engine to Fetch Unit Bus (Fast 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
Tex_RegFile_Read_Data	SP->TEX	2048	16 Fetch Addresses read from the Register file
Tex_RegFile_Write_Data	TEX→SP	2048	16 texture results

Formatted

~~20.1.13~~22.1.13 Sequencer to Fetch Unit bus (Slow Bus)

Formatted: Bullets and Numbering

Once every four clock, the fetch unit sends to the sequencer on wich 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 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	TEX→ SEQ	1	Data ready
Tex_Clause_Num	TEX→ SEQ	3	Clause number
Tex_cst	SEQ→TEX	10	Fetch state address 10 bits sent over 4 clocks
Tex_Inst	SEQ→TEX	12	Fetch instruction address 12 bits sent over 4 clocks
EO_CLAUSE	SEQ→TEX	1	Last instruction of the clause
PHASE	SEQ→TEX	1	Write phase signal
LOD_CORRECT	SEQ→TEX	96	LOD correct 3 bits per comp 2 components per quad * 16 quads
Mask	SEQ→TEX	64	Pixel mask 1 bit per pixel
Tex_Clause_Num	SEQ→TEX	3	Clause number
Tex_Write_Register_Index	SEQ->TEX	7	Index into Register file for write of returned Fetch Data
Tex_Read_Register_Index	SEQ->SP	7	Index into Register files for reading Fetch Address (internal)

~~21.~~23. Internal interfaces

Formatted: Bullets and Numbering

~~21.1.1~~23.1.1 RE to SEQ : Vertex Control Bus

Name	Direction	Bits	Description
STATE	VGT→SEQ	21	Render State (6*3+3 for constants)
Vert counter	VGT→SEQ	6	Which vertices are valid
Inputs_vert	VGT→SEQ	1	0: Normal 128 bits per vert 1: double 256 bits per vert

This information is sent over 4 clocks.~~s information needs to be sent over 64 clocks.~~

Formatted: Bullets and Numbering

~~22.~~24. Examples of program executions

~~22.1.1~~24.1.1 *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 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

~~22.1.2~~24.1.2 *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 (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

~~22.1.3~~24.1.3 *Notes*

Formatted: Bullets and Numbering

14. T~~t~~he state machines and arbitrers will operate ahead of time so that they will be able to immediately start the real threads or stall.
15. T~~t~~he 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~~, parameter buffer allocation, loops and branches and parameter cache de-allocation~~ still needs to be specked out.

~~23.~~25. 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 bandwith 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 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. 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?

~~Size of the fifo containing the information of a vector of pixels/vertices. And size of the fifos before the reservation stations.~~

~~Loops and branches.~~

	ORIGINATE DATE	EDIT DATE	DOCUMENT-REV. NUM.	PAGE
	24 September, 2001	4 September, 20155 ~~November, 200126~~	GEN-CXXXXX-REVA	1 of 32

Author: Laurent Lefebvre

Issue To:	**Copy No:**

R400 Sequencer Specification

SEQ

Version 1.1~~0~~

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

AUTOMATICALLY UPDATED FIELDS:
Document Location: C:\perforce\r400\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.

ATI 2017
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.

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.

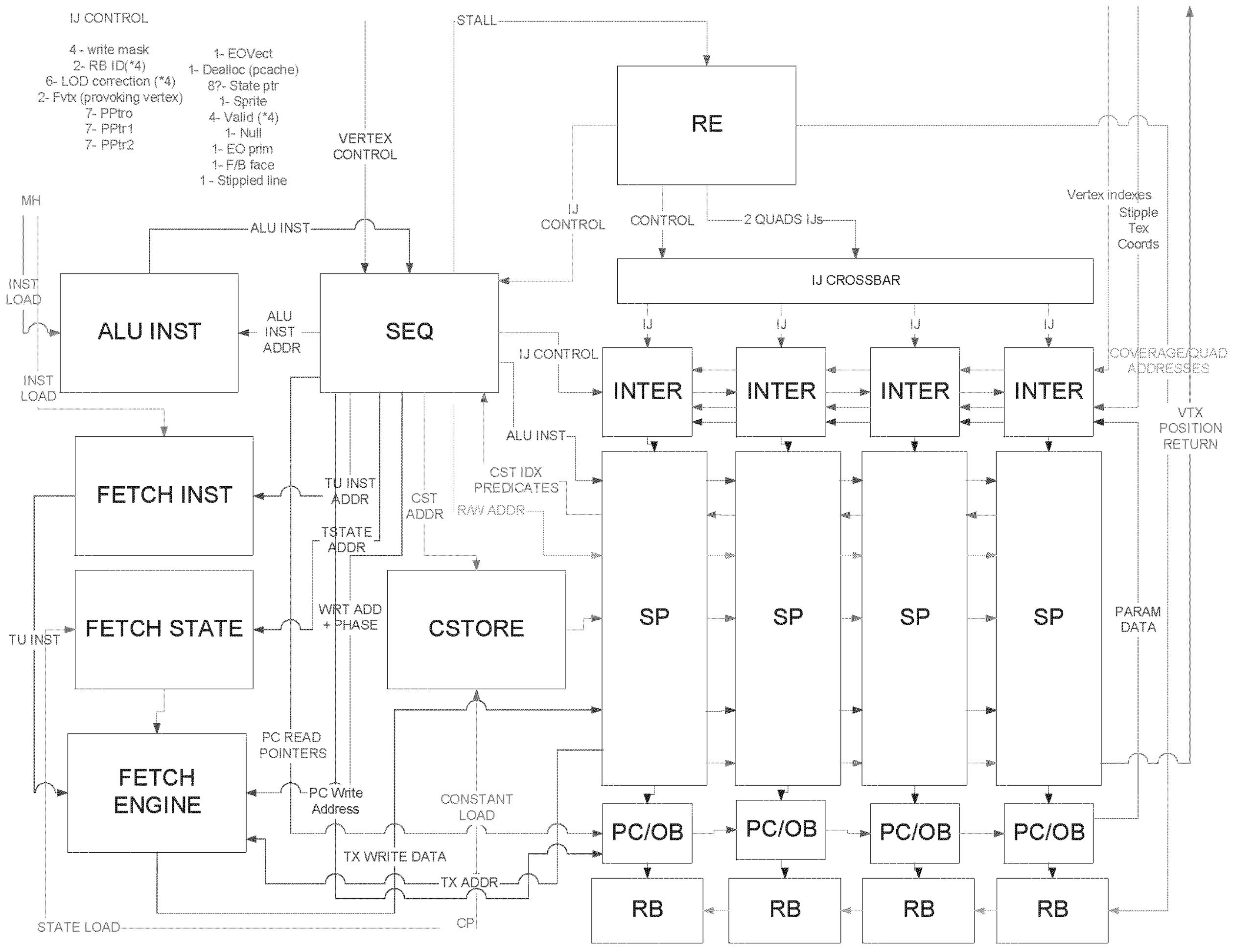
The vertex or pixel program specifies how many GPR's it needs to execute. The sequencer will not start the next vector until the needed space is available.

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.

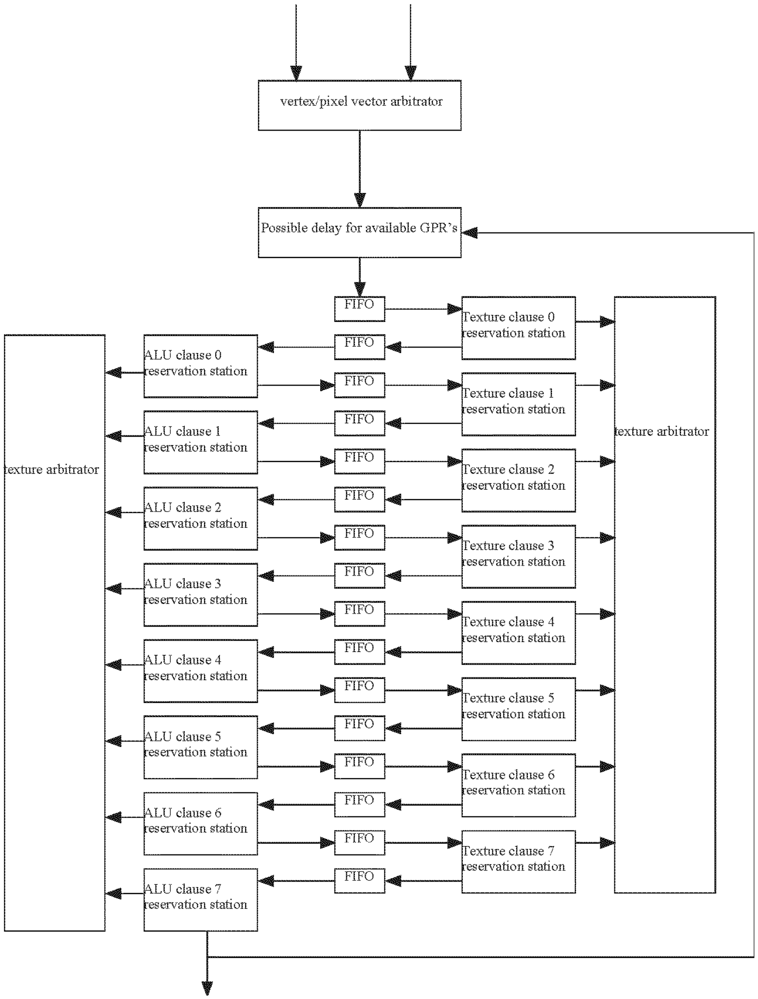
To support the shader pipe the raster engine also contains the shader instruction cache and constant store. There are only one constant store for the whole chip and one instruction store. These will be shared among the four shader pipes. The four shader pipes also execute the same instuction thus there is only one sequencer for the whole chip.

PROTECTIVE ORDER MATERIAL

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
	24 September, 2001	4 September, 20155 ~~November, 200126~~		6 of 32



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 21 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 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). One arbitrer 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 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 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)?}

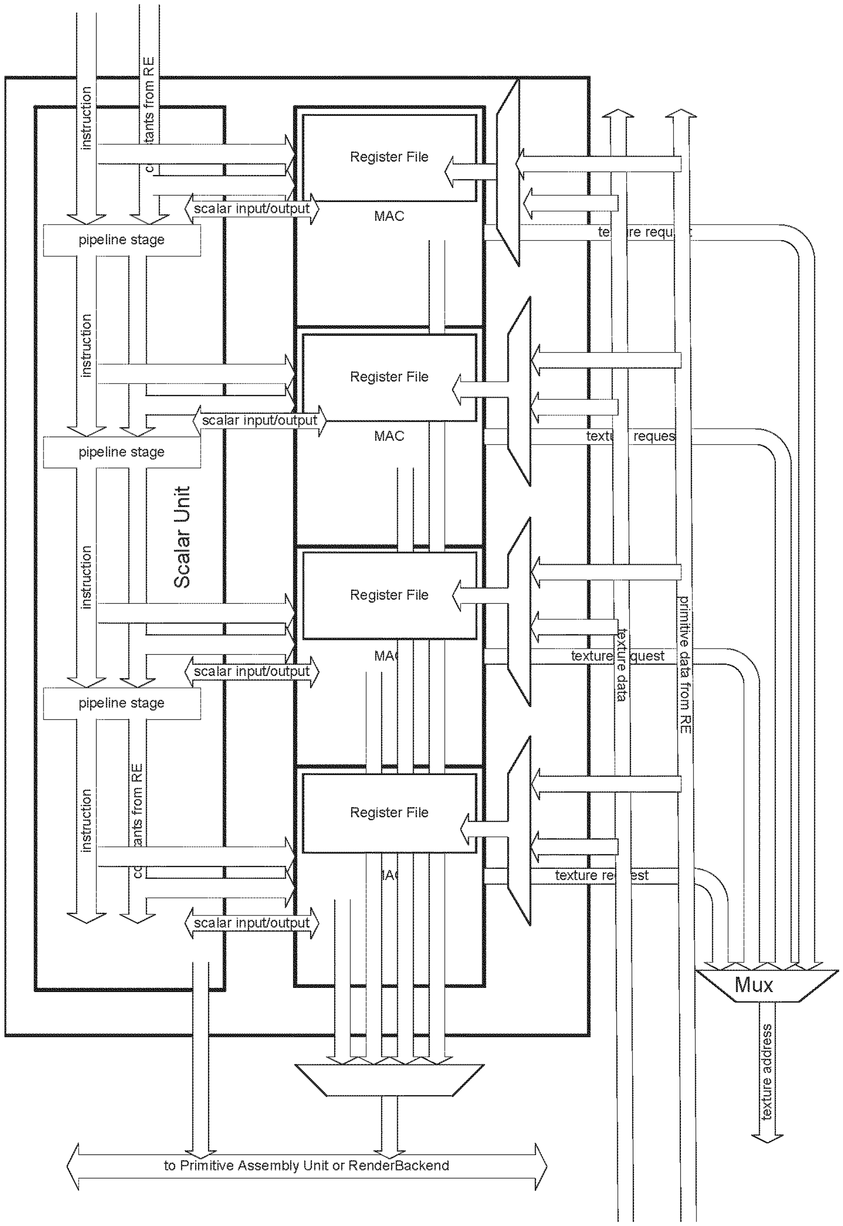
A special case is for HOS surfaces wich can export 12 parameters per last 6 clauses to the output buffer. If the output buffer is full or doesn't have enough space the sequencer will prevent such a vertex group to enter an exporting clause.

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.

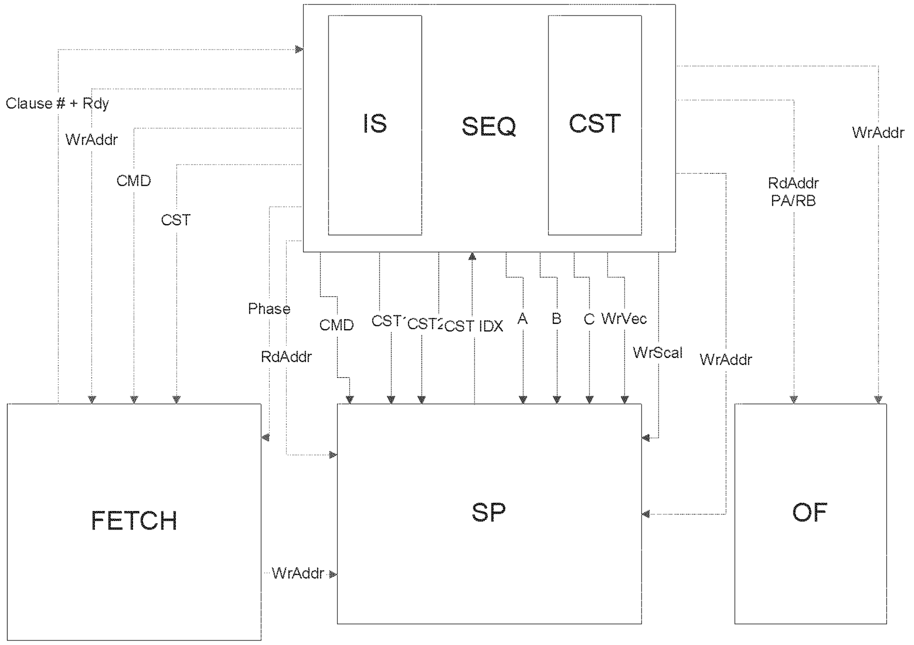
Only two ALU state machine may have access to the register file address bus or the instruction decode bus at one time. Similarly, only one fetch state machine may have access to the register file address bus at one time. Arbitration is performed by three arbitrer blocks (two for the ALU state machines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

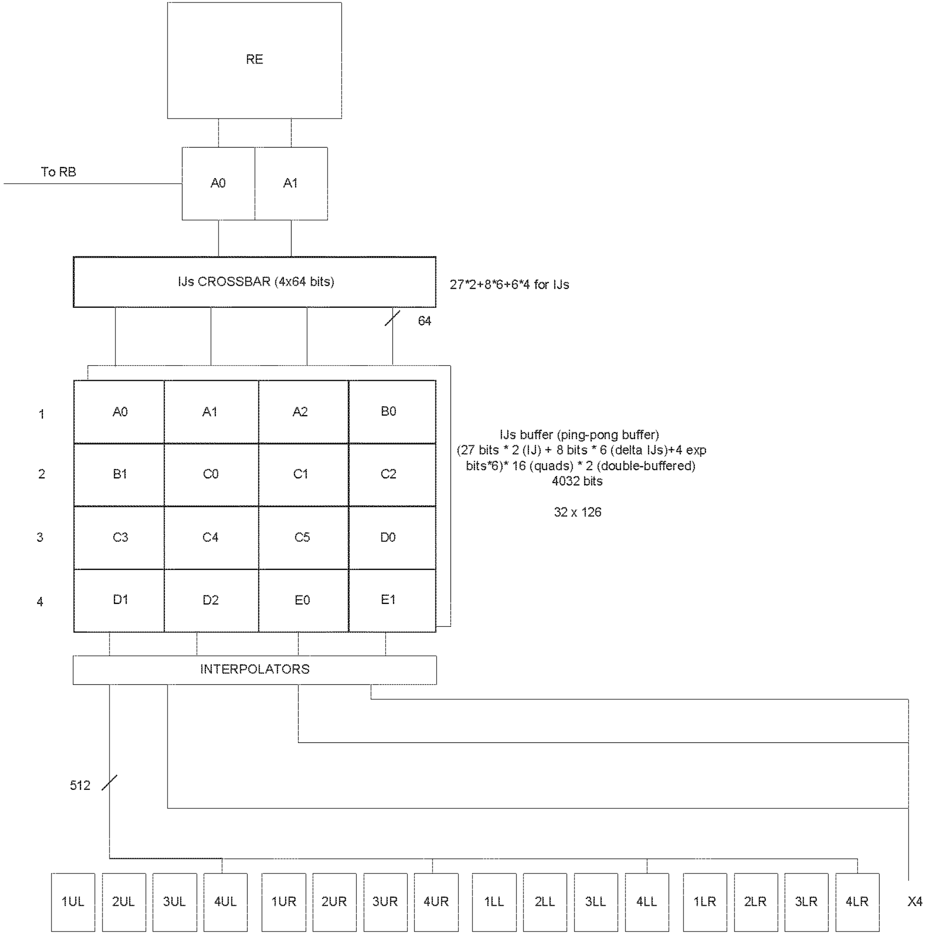
1.3 Control Graph



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)
27*2+8*6+6*4 for IJs
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)
(27 bits * 2 (IJ) + 8 bits * 6 (delta IJs)+4 exp bits*6)* 16 (quads) * 2 (double-buffered)
4032 bits
32 x 126
INTERPOLATORS
512
1UL
2UL
3UL
4UL
1UR
2UR
3UR
4UR
1LL
2LL
3LL
4LL
1LR
2LR
3LR
4LR
X4

PROTECTIVE ORDER MATERIAL

	ORIGINATE DATE	EDIT DATE	R400 Sequencer Specification	PAGE
	24 September, 2001	4 September, 20155 November, 200126		12 of 32

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19
SP0	A0	B1	C3	D1					A0	B1	C3	D1					V 0-3	V 16-19	V 32-35	V 48-51
SP1	A1		C4	D2		C0			A1		C4	D2		C0			V 4-7	V 20-23	V 36-39	V 52-55
SP2	A2		C5			C1		E0	A2		C5			C1		E0	V 8-11	V 24-27	V 40-43	V 56-59
SP3					B0	C2	D0	E1					B0	C2	D0	E1	V 12-15	V 28-31	V 44-47	V 60-63
	P0								P1											

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.

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 Store

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 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.

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 (one for constant,registert)
Loop_Step[7:0] [15:0] times 2 (one for constant,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.

~~We also keep 8 predicate vectors and 8 AND/OR sets of 3 bits. These bits can be 0: all 0s, 1: all ones and 11: mixed.~~

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_execute_or_Jump~~
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.

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.

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 is going to break the loop and set de 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 ~~everytime~~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 ~~exeeds~~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.

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 ~~wich~~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 ~~exemple~~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

Formatted: Bullets and Numbering

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

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

~~6.4~~6.5 Register file indexing

Formatted: Bullets and Numbering

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

6.6 Predicated Instruction support for Texture clauses

Formatted: Bullets and Numbering

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.

6.7 Debugging the Shaders

Formatted: Bullets and Numbering

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

6.7.1 *Method 1: Debugging registers*

Formatted: Bullets and Numbering

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.

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

Formatted: Bullets and Numbering

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
4) Debug Count

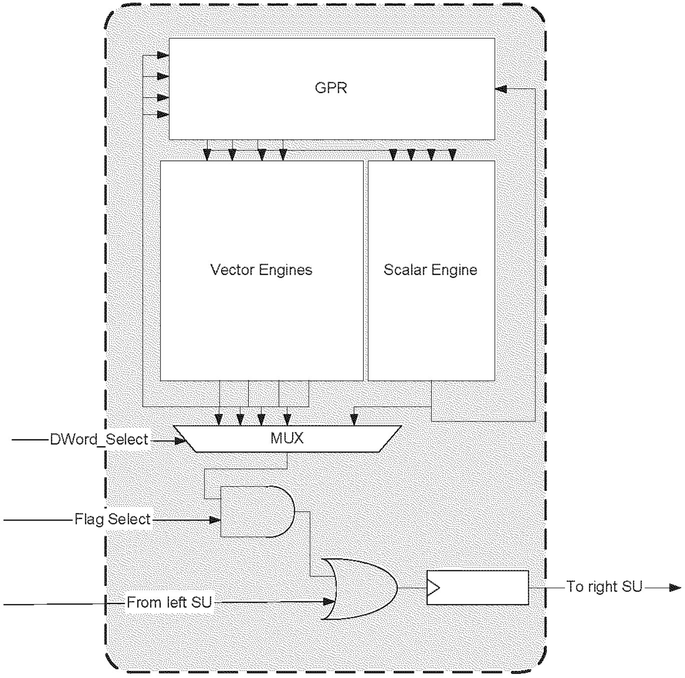
Formatted: Bullets and Numbering

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 modes, normal execution is done until we reach an address specified by the address register or an 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.

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

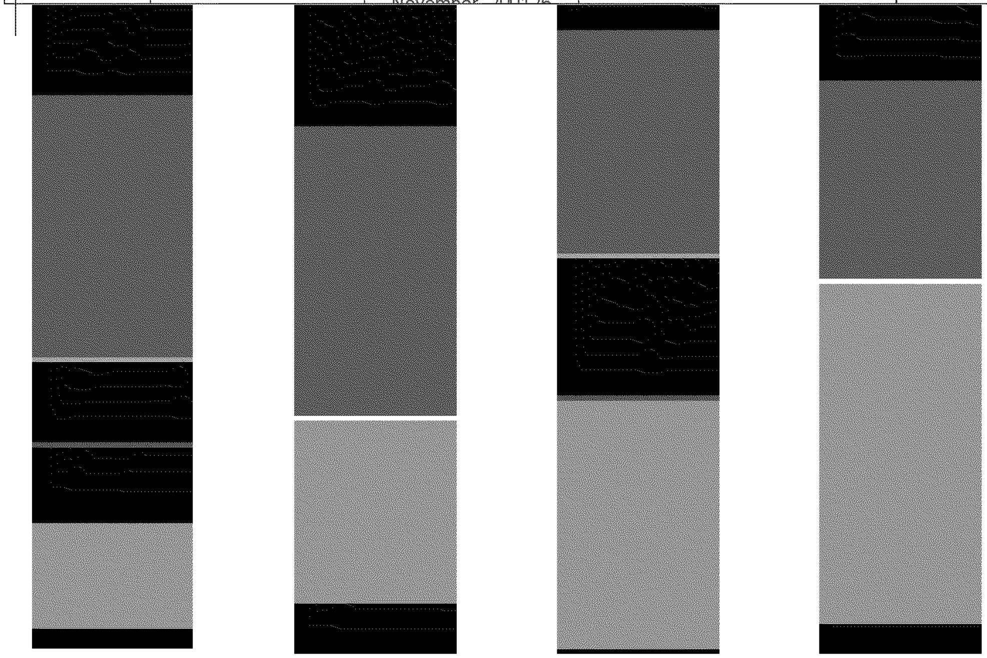
~~However, if the driver sets the kill_vector_on register to 0 (don't use) then the 64 bit kill mask becomes the 5th predicate vector and is kept across clause boundaries (thus allowing predicated instructions to be used in texture clauses). In this mode, the sequencer is going to send all 1s to the RBs for coverage mask information.~~

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. ~~The buffer they want to export into must be specified in the "exports" field of the state registers.~~

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?). The sequencer will set the OUT_FILE_FULL signal n clocks before the output file is actually full and thus the ALU arbitrer will be able read this signal and act accordingly by not preventing exporting clauses to proceed.

13. Content of the reservation station FIFOs

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 ~~19x24~~ 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 ~~19x24~~ 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 ~~19~~ 20 Exp 4 for I + Sign
Mantissa ~~19~~ 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.~~6~~

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) the vertex shader can export to the PA both the vertex position and the point sprite. It can also do so at clause 8 if not done at clause 4. 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.

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 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.

20. Registers

20.1 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 ~~vetexes~~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
~~VWRAP~~PWRAP	~~wrap~~start point for the ~~vertex~~pixel shader instruction store (vertex shader always starts at 0)
~~PWRAP~~SHAREDWRAP	~~wrap~~start point for the ~~pixel shader~~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

20.2 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]	~~wich~~which parameters are to be gouraud shaded
~~GEN_TEX~~	~~Do we generate texture coordinates for 1st parameter or not~~
CYL_WRAP[0…63]	for ~~wich~~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	~~wich~~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)
kill_vector_on	use the mask kill vector to kill pixels and optimize texture pipe fetches OR use it as the fifth predicate vector ~~wich~~which is the only predicate vector kept across clause boundaries.

21. DEBUG registers

21.1 Control

Formatted: Bullets and Numbering

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

21.2 Context

Formatted: Bullets and Numbering

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
Clause_mode[3]   clause mode for debug method 2

22. Interfaces

22.1 External Interfaces

22.1.1 *PA/SC to SP0 : 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	PA→SP0	64~~3~~	IJ information sent over 2 clocks
Mask	PA→SP0	1	Write Mask

22.1.2 *PA/SC to SEQ : 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.

Name	Direction	Bits	Description
Write Mask	PA→SEQ(SP)	4	Quad Write mask left to right
LOD_CORRECT	PA→SEQ(SP)	24	LOD correction per quad (6 bits per quad)
FVTX	PA→SEQ(SP)	2	Provoking vertex for flat shading
PPTR0	PA→SEQ(SP)	11	P Store pointer for vertex 0
PPRT1	PA→SEQ(SP)	11	P Store pointer for vertex 1
PPTR2	PA→SEQ(SP)	11	P Store pointer for vertex 2
E_OFF_VECTOR	PA→SEQ(SP)	1	End of the vector
DEALLOC	PA→SEQ(SP)	1	Deallocation token for the P Store
STATE	PA→SEQ(SP)	21	State/constant pointer (6*3+3)
VALID	PA→SEQ(SP)	16	Valid bits for all pixels
NULL	PA→SEQ(SP)	1	Null Primitive (for PC deallocation purposes)
E_OFF_PRIM	PA→SEQ(SP)	1	End Of the primitive
FBFACE	PA→SEQ(SP)	1	Front face = 1, back face = 0
TYPE	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
RTRn	SEQ→PA	1	Stalls the PA in n clocks
RTS	PA→SEQ(SP)	1	PA ready to send data

22.1.3 *SEQ to SP0 : Interpolator bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
TYPE	SEQ→SP0	3	Type of the primitive

Formatted

			000 : Normal 001 : Stippled line/Poly 011 : Real Time 100 : Line AA 101 : Point AA 110 : Sprite
FVTX	SEQ→SP0	2	Provoking vertex for flat shading
FLAT_GOURAUD	SEQ→SP0	1	Flat or gouraud shading
CYL_WRAP	SEQ→SP0	4	Wich parameter needs to be cylindrical wrapped
Inter_Write_Register_Index	SEQ→SP0	7	Index into Register file for write of Interpolated data
Write_Mask	SEQ→SP0	4	Quad Write mask left to right
IJ_Line number	SEQ→SP0	2	Line in the IJ buffer to use to interpolate
Swap_Buffers	SEQ→SP0	1	Swap the IJ buffers at the end of the interpolation
Param_0	SEQ→SP0	1	We are interpolating parameter 0

Formatted
Formatted
Formatted
Formatted
Formatted

22.1.4 SEQ to SP0 : Parameter Cache bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Ptr1	SEQ→SP0	7	Pointer of PC (7 LSBs of Pointer)
Ptr2	SEQ→SP0	7	Pointer of PC (7 LSBs of Pointer)
Ptr3	SEQ→SP0	7	Pointer of PC (7 LSBs of Pointer)

22.1.5 SEQ to SX0 : Parameter Cache Mux control Bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Mux1	SEQ→SX0	4	Mux control for PC (4 MSbs of Pointer)
Mux2	SEQ→SX0	4	Mux control for PC (4 MSbs of Pointer)
Mux3	SEQ→SX0	4	Mux control for PC (4 MSbs of Pointer)

Formatted
Formatted
Formatted
Formatted
Formatted

22.1.6 SX0 to SP0 : Parameter Cache Return bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Vtx_data_1	SX0→SP0	128	Vertex data to interpolate
Vtx_data_2	SX0→SP0	128	Vertex data to interpolate
Vtx_data_3	SX0→SP0	128	Vertex data to interpolate

~~22.1.3~~22.1.7 VGT to SP0/SEQ : Vertex Bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Vertex indexes	VGT→~~RE~~SP0	128	Pointers of indexes or HOS surface information
EOF_vector	VGT→~~RE~~SP0/SEQ	1	End of the vector
Inputs_vert	VGT→~~RE~~SP0/SEQ	1	0: Normal 128 bits per vert 1: double 256 bits per vert
STATE	VGT→SEQ	21	Render State (6*3+3 for constants)

~~22.1.4~~22.1.8 CP to SEQ : Constant store load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Constant Address	CP→SEQ	8	Address of the block of 4 constants
Constant Data	CP→SEQ	512	Data sent over 4 clocks
Remap Address	CP→SEQ	10	Remaping address write address
Remap Data pointer	CP→SEQ	8	Remaping pointer

~~22.1.5~~22.1.9 CP to SEQ : Fetch State store load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Constant Address	CP→SEQ	8	Address of the block of 4 state constants
Constant Data	CP→SEQ	512	Data sent over 4 clocks
Remap Address	CP→SEQ	10	Remaping address write address

Remap Data pointer	CP→SEQ	8	Remaping pointer

~~22.1.6~~22.1.10 CP to SEQ : Control State store load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Constant Address	CP→SEQ	?	
Constant Data	CP→SEQ	?	

Formatted

Formatted

Formatted

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

~~22.1.7~~22.1.11 MH to SEQ: Instruction store Load

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Instruction address	MH→SEQ	12	Instruction address
Instruction	MH→SEQ	96	Instruction X times
Control Instruction address	MH→SEQ	9	Pointer to the control instruction store
Control Instruction	MH→SEQ	32	Control Instruction X times

{ISSUE: CP or MH?}

~~22.1.8~~22.1.12 SP0 to ~~RB~~ SX0 : Pixel read from RBs

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Export_data	SP0→~~RB~~SX0	64*16	32 pairs of 32 bits channel values
Shader_Dest	SP0→SX0~~SP→RB~~	4	Specifies one of the of up to 12 export destinations
Shader_Count	SP0→SX0~~SP→RB~~	3	Each set of four pixels or vectors is exported over eight clocks. This field specifies where the SP is in that sequence.
Shader_Last	SP0→SX0~~SP→RB~~	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.
Shader_PixelValid	SP0→SX0~~SP→RB~~	4x4	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
Shader_WordValid	SP0→SX0~~SP→RB~~	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

~~22.1.9~~22.1.13 SEQ to ~~RB~~ SX0 : Control bus

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Export_Pixel	SEQ→~~RB~~SX0	1	1: Pixel 0: Vertex
Export_SEND	SEQ→SX0~~SEQ→RB~~	1	Raised to indicate that the SQ is starting an export
Export_Clause	SEQ→SX0~~SEQ→RB~~	3	Clause number, which is needed for vertex clauses
Export_State	SEQ→SX0~~SEQ→RB~~	21?	State ID, which is needed for vertex clauses

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

~~22.1.10~~22.1.14 ~~RB~~ SX0 to SEQ : Output file control

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Export_RTS	~~RB~~SX0→SEQ	1	Raised by ~~RB~~ SX0 to indicate that the following two fields reflect the result of the most recent export
Export_Position	SX0→SEQ~~RB→SEQ~~	1	Specifies whether there is room for another position.
Export_Buffer	SX0→SEQ~~RB→SEQ~~	7	Specifies the space availble 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

~~22.1.11~~22.1.15 *SP0 to ~~RB~~ SX0 : Position return bus*

Formatted: Bullets and Numbering

Name	Direction	Bits	Description
Position return	SP0→~~RB~~SX0	128	Position data or sprite size (per clock)

~~For point sprites and position exports the size and position are interleaved on a 16 x 16 basis. We export 1 position then 1 point sprite sizes. The storage used is of 64x128 bits for position and 64x32 bits for sprite size, it is taken from the output buffer. Additionnally,if needed the edge flags are packed into the bits of the sprite sizes.~~

~~22.1.12~~22.1.16 *Shader Engine to Fetch Unit Bus (Fast 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
Tex_RegFile_Read_Data	SP->TEX	2048	16 Fetch Addresses read from the Register file
Tex_RegFile_Write _Data	TEX→SP	2048	16 texture results

~~22.1.13~~22.1.17 *Sequencer to Fetch Unit bus (Slow Bus)*

Formatted: Bullets and Numbering

Once every four clock, the fetch unit sends to the sequencer on wich 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 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	TEX→ SEQ	1	Data ready
Tex_Clause_Num	TEX→ SEQ	3	Clause number
Tex_cst	SEQ→TEX	10	Fetch state address 10 bits sent over 4 clocks
Tex_Inst	SEQ→TEX	12	Fetch instruction address 12 bits sent over 4 clocks
EO_CLAUSE	SEQ→TEX	1	Last instruction of the clause
PHASE	SEQ→TEX	1	Write phase signal
LOD CORRECT	SEQ→TEX	96	LOD correct 3 bits per comp 2 components per quad * 16 quads
Mask	SEQ→TEX	64	Pixel mask 1 bit per pixel
Tex_Clause_Num	SEQ→TEX	3	Clause number
Tex_Write_Register_Index	SEQ->TEX	7	Index into Register file for write of returned Fetch Data
Tex_Read_Register_Index	SEQ->SP	7	Index into Register files for reading Fetch Address (internal)

Formatted: Bullets and Numbering

~~23. Internal interfaces~~

~~23.1.1 *RE to SEQ : Vertex Control Bus*~~

~~This information is sent over 4 clocks.~~

Formatted: Bullets and Numbering

~~24.~~23. Examples of program executions

~~24.1.1~~23.1.1 *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 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

~~24.1.2~~23.1.2 *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 (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

~~24.1.3~~23.1.3 *Notes*

Formatted: Bullets and Numbering

14. The state machines and arbitrers 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.

~~25.~~24. 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 bandwith 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 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. 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?