UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

LG ELECTRONICS, INC.,
Petitioner,

v.

ATI TECHNOLOGIES ULC,
Patent Owner.

Case IPR2015-00326
Patent 6,897,871 B1

DECLARATION OF ANDREW WOLFE REGARDING U.S. PATENT NO. 6,897,871

Mail Stop "Patent Board"
Patent Trial and Appeal Board
U.S. Patent and Trademark Office
P.O. Box 1450
Alexandria, VA 22313-1450

ATI 2003
LG v. ATI
IPR2015-00326

Table of Contents

I, Andrew Wolfe, declare as follows:

I. INTRODUCTION

1. I have been retained by the patent owner, ATI Technologies ULC ("ATI"), to evaluate several technical issues relating to U.S. Patent No. 6,897,871 ("the '871 patent").

2. *First*, I have been asked to evaluate source code related to the development of the "R400" project at its state of development on August 5, 2002, and to provide my opinion regarding whether the functionality of this source code for the R400 chip and the structure it describes corresponds to each and every element as set forth in claims 1, 2, 3, 5, 6, 8, 9, 10, 11, 13, 15, 17, 18, and 20 of the '871 patent. As set forth below, it is my opinion that this source code includes every limitation of these claims.

3. *Second*, I have been asked to review U.S. Patent Application No. 10/718,318 ("the '318 application"), filed November 20, 2003, to which the '871 patent claims priority, and to provide my opinion regarding whether claims 1, 2, 3, 5, 6, 8, 9, 10, 11, 13, 15, 17, 18, and 20 are supported by the '318 application. As set forth below, it is my opinion that the '318 application provides support for every limitation of these claims.

4. *Third*, I have been asked to review ATI's internal documents relating to the R400 project to provide my opinion regarding whether the inventors of the '871 patent conceived claims 1, 2, 3, 5, 6, 8, 9, 10, 11, 13, 15, 17, 18, and 20. As set forth below, it is my opinion that these internal documents show that the '871 patent inventors conceived of every limitation of these claims.

5. *Fourth*, I have been asked to review Rich and Kurihara and to provide my opinion regarding whether these references render obvious claims 15 and 20. As set forth below, it is my opinion that claims 15 and 20 are patentable over these references.

II. BACKGROUND

6. I have more than 30 years of experience as a computer architect, computer system designer, personal computer graphics designer, educator, and executive in the electronics industry. A curriculum vitae is attached as Exhibit 2003 to this report and is summarized below.

7. In 1985, I earned a B.S.E.E. in Electrical Engineering and Computer Science from The Johns Hopkins University. In 1987, I received an M.S. degree in Electrical and Computer Engineering from Carnegie Mellon University. In 1992, I received a Ph.D. in Computer Engineering from Carnegie Mellon University. My

doctoral dissertation pertained to a new approach for the architecture of a computer processor.

8. In 1983, I began designing touch sensors, microprocessor-based computer systems, and I/O (input/output) cards for personal computers as a senior design engineer for Touch Technology, Inc. During the course of my design projects with Touch Technology, I designed I/O cards for PC-compatible computer systems, including the IBM PC-AT, to interface with interactive touch-based computer terminals that I designed for use in public information systems. I continued designing and developing related technology as a consultant to the Carroll Touch division of AMP, Inc., where in 1986, I designed one of the first custom touch screen integrated circuits.

9. While I studied at Carnegie Mellon University for my master's degree, from 1986 and through 1987, I designed and built a high-performance computer system. From 1986 through early 1988, I also developed the curriculum, and supervised the teaching laboratory, for processor design courses.

10. In the latter part of 1989, I worked as a senior design engineer for ESL-TRW Advanced Technology Division. While at ESL-TRW, I designed and built a bus interface and memory controller for a workstation-based computer system, and also worked on the design of a multiprocessor system.

11. At the end of 1989, I (along with my partners) reacquired the rights to the technology I had developed at Touch Technology and at AMP, and founded The Graphics Technology Company. Over the next seven years, as an officer and a consultant for The Graphics Technology Company, I managed the company's engineering development activities and personally developed dozens of touch screen sensors, controllers, and interactive touch-based computer systems.

12. I have consulted, formally and informally, for a number of fabless semiconductor companies. In particular, I have served on the technical advisory boards for two processor design companies: BOPS, Inc., where I chaired the board, and Siroyan Ltd., where I served in a similar role for three networking chip companies—Intellon, Inc., Comsilica, Inc., and Entridia, Inc.—and one 3D game accelerator company, Ageia, Inc.

13. I have also served as a technology advisor to Motorola and to several venture capital funds in the United States and Europe. Currently, I am a director of Turtle Beach Corporation, providing guidance in its development of premium audio peripheral devices for a variety of commercial electronic products.

14. From 1991 through 1997, I served on the Faculty of Princeton University as an Assistant Professor of Electrical Engineering. At Princeton, I taught undergraduate and graduate-level courses in Computer Architecture,

Advanced Computer Architecture, Display Technology, and Microprocessor Systems, and conducted sponsored research in the area of computer systems and related topics. I was also a principal investigator for Department of Defense ("DOD") research in video technology and a principal investigator for the New Jersey Center for Multimedia Research. From 1999 through 2002, I taught the Computer Architecture course to both undergraduate and graduate students at Stanford University multiple times as a Consulting Professor. At Princeton, I received several teaching awards, both from students and from the School of Engineering. I have also taught advanced microprocessor architecture to industry professionals in IEEE and ACM sponsored seminars. I am currently a lecturer at Santa Clara University teaching graduate courses on Computer Organization and Architecture and undergraduate courses on electronics and embedded computing.

15. From 1997 through 2002, I held a variety of executive positions at a publicly-held fabless semiconductor company originally called S3, Inc. and later called SonicBlue Inc. I held the positions of Chief Technology Officer, Vice President of Systems Integration Products, Senior Vice President of Business Development, and Director of Technology, among others. At the time I joined S3, the company supplied graphics accelerators for more than 50% of the PCs sold in the United States.

16. While at S3/SonicBlue I developed technology for and participated in the development of products for digital music and digital video including HDTVs, DVD players and recorders, DVRs, portable video devices, PDAs, and tablets. I also supervised the video research and development team.

17. I have published more than 50 peer-reviewed papers in computer architecture and computer systems and IC design.

18. I also have chaired IEEE and ACM conferences in microarchitecture and integrated circuit design and served as an associate editor for IEEE and ACM journals.

19. I am a named inventor on at least 43 U.S. patents and 27 foreign patents.

20. In 2002, I was the invited keynote speaker at the ACM/IEEE International Symposium on Microarchitecture and at the International Conference on Multimedia. From 1990 through 2005, I was also an invited speaker on various aspects of technology and the PC industry at numerous industry events including the Intel Developer's Forum, Microsoft Windows Hardware Engineering Conference, Microprocessor Forum, Embedded Systems Conference, Comdex, and Consumer Electronics Show, as well as at the Harvard Business School and the University of Illinois Law School. I have been interviewed on subjects related to

computer graphics and video technology and the electronics industry by publications such as the Wall Street Journal, New York Times, Los Angeles Times, Time, Newsweek, Forbes, and Fortune as well as CNN, NPR, and the BBC. I have also spoken at dozens of universities including MIT, Stanford, University of Texas, Carnegie Mellon, UCLA, University of Michigan, Rice, and Duke.

21. I am being compensated for my time working on this case at my customary rate of $450 per hour for work performed on the case. My compensation is not in any way related to the outcome of the case.

III. EXHIBITS

22. In this Declaration, I cite to the following Exhibits.

Exhibit Number	Reference
1001	United States Patent No. 6,897,871 to Morein *et al.*
1002	Prosecution History of U.S. Patent No. 6,897,871
1003	Declaration of Dr. Nader Bagherzadeh
1004	U.S. Patent 7,015,913 to Lindholm *et al.*
1005	U.S. Patent No. 5,808,690 to Rich
1006	U.S. Patent No. 7,376,811 B2 to Kizhepat
1007	U.S. Patent No. 5,500,939 to Kurihara
1008	Mark Segal and Kurt Akeley, The OpenGL® Graphics System: A Specification (Version 1.4) (Chris Frazier and Jon Leech eds., Silicon Graphics, Inc. 2002)
1009	*Curriculum Vitae* of Dr. Nader Bagherzadeh

2004	*Curriculum Vitae* of Dr. Andrew Wolfe
2010	R400 Sequencer Specification (Version 0.4)
2028	R400 Sequencer Specification (Version 2.0)
2041	R400 Top Level Specification (Version 0.2)
2042	R400 Shader Processor (Version 1.2)
2073	Deposition Transcript of Nader Bagherzadeh, Ph.D., taken Sept. 15, 2015
2074	Deposition Transcript of Nader Bagherzadeh, Ph.D. for IPR2015-00325, taken Aug. 14, 2015
2075	*Uniram Technology, Inc. v. Taiwan Semiconductor Manufacturing Co., Ltd., et al.*, 3:04-cv-01268-VRW, Findings of Facts and Conclusions of Law, Dkt. No. 627, April 14, 2008
2076	United States Patent Application No. 10/718,318 to Morein *et al.*
2077	Graham Singer, History of the Modern Graphics Processor, Part 3, TechSpot (Apr. 10, 2013)
2078	David Luebke & Greg Humphreys, How GPUs Work, IEEE Computer, 96-100 (2007)
2079	Microsoft and ATI Technologies Announce Technology Development Agreement, Microsoft (Aug. 14, 2003)
2080	Anton Shilov, ATI and NVIDIA Proclaim Different Graphics Processors Architecture Goals: ATI Says Unified Rendering Engine – the Way to Go, NVIDIA Disagrees, Xbit (Dec. 23, 2004, 7:55 AM)
2081	Anton Shilov, NVIDIA Chief Architect: Unified Pixel and Vertex Pipelines – The Way to Go. NVIDIA Says It Would Make a Chip with Unified Pipes "When it Makes Sense," Xbit (July 11, 2005, 11:07 PM)
2082	Yoo *et al.*, Mobile 3D Graphics SoC: From Algorithm to Chip (2010)
2083	Luna, Introduction to 3D Game Programming with DirectX 9.0, Figures 4.2, 5.7, pp. 94-97, 107-109 (2003)
2084	Ahmed *et al.*, OpenGL - Lighting, Material, Shading and Texture Mapping (August 28, 2009)

2085	MICROSOFT COMPUTER DICTIONARY (5th Ed. 2002)
2086	Foley *et al.*, Fundamentals of Interactive Computer Graphics (1984)
2087	S3 Graphics, DirectX 10 Architecture for Chrome 400 Series Discrete Graphics Processors, A S3 Graphics White Paper (July 21, 2007)
2088	COLLIN, DICTIONARY OF COMPUTING (4th ed., 2002)
2089	Woo, J.H. et al., A 195/152-mW mobile multimedia SoC with fully programmable 3D graphics and MPEG4/H.264/JPEG. IEEE J. Solid-St. Circ., 43 (9), 2047–2056 (2008)
2090	Technical Brief, NVIDIA GeForce® GTX 200 GPU Architectural Overview (May, 2008)
2091	Intel® Processor Graphics DirectX Developer's Guide (2008-2010)
2092	The Rise of Mobile Gaming on Android: Qualcomm® Snapdragon™ Technology Leadership (2014)
2093	RTL Code File: sq.v
2094	RTL Code File: sq_ais_output.v
2095	RTL Code File: sq_alu_instr_queue.v
2096	RTL Code File: sq_alu_instr_seq.v
2097	RTL Code File: sq_thread_arb.v
2098	RTL Code File: sq_input_arb.v
2099	RTL Code File: sq_instruction_store.v
2100	RTL Code File: sq_defs.v
2101	RTL Code File: sq_thread_buff.v
2102	RTL Code File: sq_target_fetch.v
2103	RTL Code File: sq_export_alloc.v
2104	RTL Code File: vector.v
2105	RTL Code File: macc_gpr.v
2106	RTL Code File: export_control.v
2107	RTL Code File: macc.v
2108	RTL Code File: macc32.mc
2109	RTL Code File: sx.v

2110	RTL Code File: parameter_caches.v
2111	RTL Code File: param_cache_ctl.v
2112	RTL Code File: sp.v
2113	RTL Code File: export_buffers.v
2114	RTL Code File: pa.v
2115	RTL Code File: pa_ag.v
2116	RTL Code File: pa_sxifccg.v
2117	RTL Code File: pa_ccg_sxifsm.v
2118	RTL Code File: sc.v
2119	Takahashi, The XBOX 360 Uncloaked (2006)
2120	Microsoft Corporation Annual Report (2006)

23. Exhibits 2077-2092 and 2119 are true and accurate copies of what they purport to be.

24. This declaration represents only the opinions I have formed to date. I may consider additional documents as they become available or other documents that are necessary to form my opinions. I reserve the right to revise, supplement, or amend my opinions based on new information and on my continuing analysis.

IV. OVERVIEW OF THE LAW USED FOR THIS DECLARATION

25. When considering the '871 patent and stating my opinions, I am relying on legal principles that have been explained to me by counsel.

A. Burden of Proof

26. I understand that for a claim to be found patentable, the claims must be, among other requirements, novel and nonobvious from what was known at the time of the invention.

27. I understand that the information that is used to evaluate whether a claim is novel and nonobvious is referred to as prior art.

28. I understand that in this proceeding, LG has the burden of proving that each element of the challenged claims is rendered obvious by the alleged prior art references.

B. Level of skill in the art

29. I have been asked to consider the level of ordinary skill in the art that someone would have had from August 2001 to November 2003. With over 30 years of experience as a computer architect, computer system designer, personal computer graphics designer, educator, and executive in the electronics industry, I am well informed of the level of ordinary skill in the art. I understand that determining the level ordinary skill in the art takes into consideration:

- Levels of education and experience of persons working in the field;
- Types of problems encountered in the field; and

- Sophistication of the technology.

30. Based on the technologies disclosed in the '871 patent and the considerations listed above, a person having ordinary skill in the art ("POSA") would have at least a bachelor's degree in electrical or computer engineering or computer science plus five years of experience in the computer graphics hardware industry, or a master's degree in electrical or computer engineering or computer science plus two years of experience in that industry, or an equivalent combination of education and experience.

31. Throughout my declaration, even if I discuss my analysis in the present tense, I am always making my determinations based on what a POSA would have known at the time of the invention. Additionally, throughout my declaration, even if I discuss something stating "I," I am referring to a POSA's understanding.

C. Reduction to Practice

32. I understand there are two types of reduction to practice—actual reduction to practice and constructive reduction to practice. My understanding of each, I describe below.

1. Actual Reduction to Practice

33. I understand that actual reduction to practice requires proof of either (i) an embodiment of a claimed invention or (ii) performance of a process that includes all limitations of the claimed invention.

34. Here, I have examined the R400 RTL code for an early version of the R400 written in Verilog. Verilog RTL code is a structural and functional embodiment of a design that in the development of 3D graphics chips is generally used to model, define, and instantiate a hardware design. Below, I identify the specific files, objects, input/output interfaces, and functions that describe each element of claims 1, 2, 3, 5, 6, 8, 9, 10, 11, 13, 15, 17, 18, and 20 of the '871 patent.

2. Constructive Reduction to Practice

35. I understand that constructive reduction to practice occurs when the patent application discussing the subject matter of the claims is filed. In this case, the constructive reduction to practice occurred on November 20, 2003, with the filing of the '318 Application. Below, I include a claim chart where I identify support for each element of claims 1, 2, 3, 5, 6, 8, 9, 10, 11, 13, 15, 17, 18, and 20 of the '318 Application.

D. Novelty

36. I understand that a claim is unpatentable for being anticipated (sometimes called lack of novelty) if a prior art reference disclosed, at the time of the invention, each claim element as arranged in the claim. I also understand that if a prior art reference fails to expressly disclose one or more claim elements, the claim may be anticipated if the missing element(s) are inherently disclosed. I understand that to establish inherency, the evidence must make clear that the missing claim element is necessarily present in the prior art reference. I understand that anticipation requires a high threshold because each and every claim element must be unambiguously taught by a single reference, either explicitly or inherently.

E. Obviousness

37. I understand that a patent claim is invalid if the claims would have been obvious to a POSA at the time of the invention. I understand that the obviousness inquiry should not be done in hindsight, but from the perspective of a POSA as of the time of invention of the patent claim.

38. I understand that to obtain a patent, the claims must have, as of the time of the invention, been nonobvious in view of the prior art.

39. I understand that a claim is obvious when the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious to a POSA at the time the invention.

40. I understand that to prove that prior art reference or a combination of prior art references renders a patent obvious, it is necessary to: (1) identify the particular references that, singly or in combination, make the patent obvious; (2) specifically identify which elements of the patent claim appear in each of the asserted references; and (3) explain how a POSA could have combined the prior art references to create the claimed invention.

41. I understand that to support a conclusion that a prior art reference or a combination of prior art references renders a patent obvious, there must be some documentary evidence. Mere statements about what is basic knowledge or common sense, *i.e.*, common knowledge as a replacement for documentary evidence, is insufficient to support a conclusion of obviousness.

42. I understand that certain objective indicia can be important evidence regarding whether a patent is obvious. Such indicia include: industry acceptance, commercial success of products covered by the patent claims; long-felt need for the invention; failed attempts by others to make the invention; copying of the invention by others in the field; unexpected results achieved by the invention as

compared to the closest prior art; praise of the invention by the infringer or others in the field; taking of licenses under the patent by others; expressions of surprise or skepticism by experts and those skilled in the art at the making of the invention; and the patentee proceeded contrary to the accepted wisdom of the prior art.

F. Obviousness to combine

43. I understand that obviousness can be established by combining multiple prior art references to meet each and every claim element, but I also understand that a proposed combination of references can be susceptible to hindsight bias.

44. I understand that references are more likely to be combinable if the nature of the problem to be solved is the same.

45. I understand that if the combination of references results in the references being unsatisfactory for their intended purposes or the combination changes the references' principle of operation, a POSA would not have a motivation to combine the references.

46. I understand that teaching away, *e.g.*, discouragement, is strong evidence that the references are not combinable. I also understand that a disclosure of more than one alternative does not necessarily constitute a teaching away. I understand that the combination does not need to result in the most desirable

embodiment, but if the proposed combination does not have a reasonable expectation of success at the time of the invention, a POSA would not have a teaching, suggestion, or motivation to combine the references.

G. Claim construction

47. I understand that in this *Inter Partes* Review proceeding the claims must be given their broadest reasonable interpretation consistent with the specification. In this declaration, I have used this broadest-reasonable-interpretation standard when interpreting the claim terms.

48. I understand that the Board construed the term "means for performing vertex operations and pixel operations and performing one of the vertex operations or pixel operations based on the selected one of the plurality of inputs" to include *a register, an instruction sequencer capable of providing instructions for performing vertex operations and pixel operations, and a processor capable of floating point, arithmetic, and logical operations on a selected input.* For the purposes of this proceeding, I apply that construction to my analysis below.

V. INSTITUTED GROUNDS

49. I understand that LG proposed nine grounds for *inter partes* review based on two primary references: Lindholm and Rich. I understand that the Board denied LG's Grounds 5, 7, and 8 in their entirety, and denied Ground 6 with

respect to claims 6 and 17. I further understand that the Board instituted *inter partes* review of claims 1-3, 5, 6, 8-11, 13, 15, 17, 18, and 20 of the '871 patent in the manner shown in the table below.

Grounds	**Claims**	**Type**	**Primary Reference**	**Secondary References**
Ground 1	1, 2, 5, 8, 10, 11, 13, and 15	Anticipation § 102	Lindholm	N/A
Ground 2	3 and 6	Obviousness § 103	Lindholm	OpenGL
Ground 3	9, 17, and 18	Obviousness § 103	Lindholm	Kizhepat
Ground 4	20	Obviousness § 103	Lindholm	Kurihara
Ground 6	15	Obviousness § 103	Rich	N/A
Ground 9	20	Obviousness § 103	Rich	Kurihara

VI. TECHNOLOGY

A. Terminologies

50. This section provides exemplary descriptions for the following terms as they are used with respect to the technology of the '871 patent. I use these descriptions when providing a general overview of computer graphics technology.

- Pixel: Short for picture (pix) element. One spot in a rectilinear grid of thousands of such spots that a device individually "paints" to form an image produced on a computer screen or on paper. A pixel is the smallest element that displays or prints. A set of pixels can be manipulated to create letters, numbers, or graphics. Ex. 2085, p. 406.

- Vertex: A point in space defined by the three coordinates x, y, and z. Ex. 2088, p. 374.
- Render: To produce a graphic image from data on an output device such as a video display or a printer. Ex. 2085, p. 449.
- Primitive: In computer graphics, a shape, such as a line, circle, curve, or polygon, that a graphics program can draw, store, and manipulate as a discrete entity. *Id.* at 419.
- Polygon: Any two-dimensional closed shape composed of three or more line segments, such as a hexagon, an octagon, or a triangle. *Id.* at 411.

B. General overview

51. In computer graphics, complex three-dimensional shapes are typically represented by a wireframe collection of simple polygons, called **primitives**, as illustrated in Figure 1 (reproduced below). Transforming these wireframe models into rich, colorful images primarily involves two types of graphics-processing calculations: (i) vertex calculations and (ii) pixel calculations. Ex. 1001, 1:11-60.

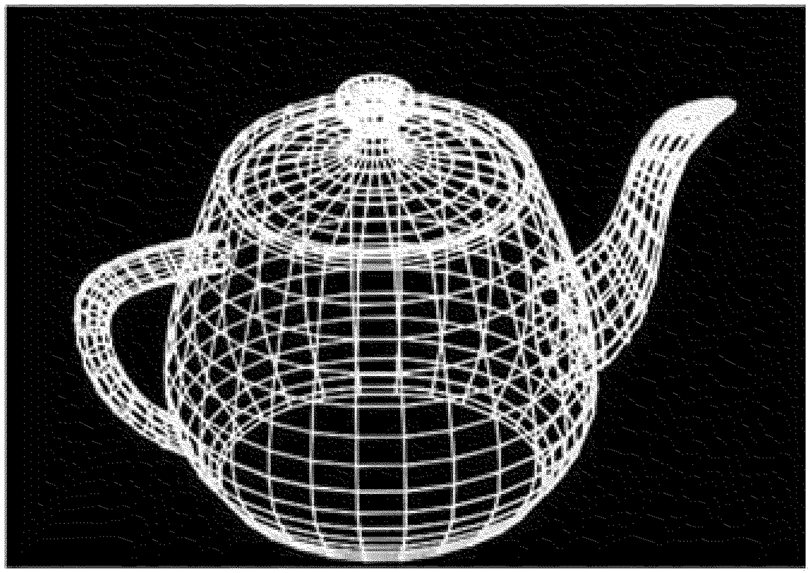


Figure 1 – Polygon wireframe of a teapot

52. **Vertex calculations** are applied to the primitives of a wireframe model to orient (*i.e.*, rotate, translate, or scale) the primitives in a desired way. Ex. 1001, 1:37-49. Each primitive can be represented by a set of numbers, called vertices (V_x, V_y, V_z). *Id.* at 1:37-42. To make the wireframe model appear to rotate, a transformation matrix is applied to the vertices of each primitive to provide a new set of reoriented vertices ($V_{x'}$, $V_{y'}$, $V_{z'}$). *Id.* at 1:42-48. In addition to rotations, transformation matrices may be applied to the vertices to make the wireframe model appear to move, grow, or shrink. These transformations are collectively referred to as vertex calculations. After the desired transformations are applied to the vertices, the reoriented vertices are then translated into pixels to generate a rendered object that can be displayed as a two-dimensional image. In some systems

vertex calculations are also used to determine the appearance characteristics of a polygon at the vertices.

53. **Pixel calculations** are different. Pixel calculations are performed on each pixel of the rendered object to determine each pixel's color and appearance attributes. *Id.* at 1:50-54. These pixel calculations may also be applied to texture data to generate the pixel color or other appearance attributes of interest. *Id.* at 3:42-46.

C. Conventional graphics systems used separate shaders for vertex calculations and pixel calculations

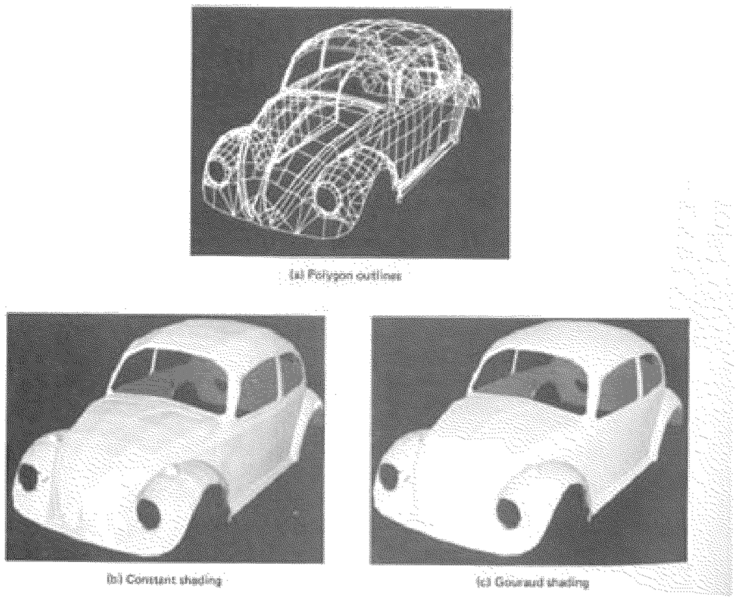
54. Around the time the technology described in the '871 patent was invented, conventional graphics-system architectures included a vertex shader to perform the vertex calculations and a separate pixel shader to perform the pixel calculations. *Id.* at 1:60-65. In these conventional architectures, vertex calculations and pixel calculations were performed sequentially. *Id.* at 2:1-6. In the first stage (vertex shading), vertex calculations built a three-dimensional scene out of polygons (*i.e.*, primitives). In the second stage (pixel shading), the primitives were translated to pixels and filled in with color. In the third stage, the shaded pixels were stored in memory called a "frame buffer" for display on a screen.

D. Drawbacks of graphics systems using separate vertex and pixel shaders

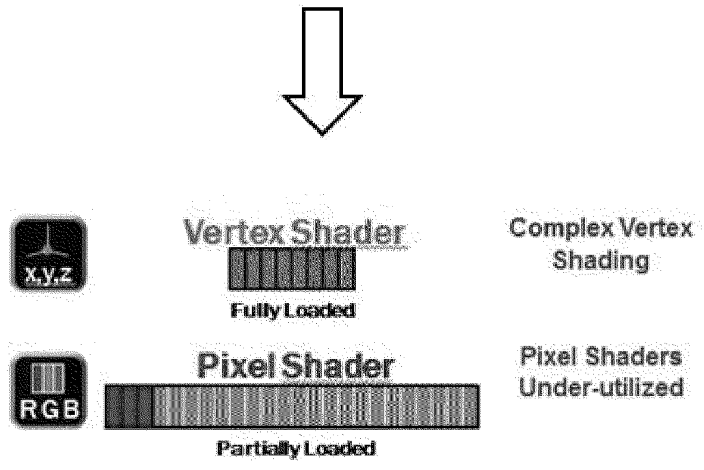
55. Graphics-system architectures using separate vertex and pixel shaders generally do not utilize shader resources efficiently. Using separate types of shaders is inefficient because graphics-processing tasks are generally not perfectly balanced between vertex and pixel calculations. As the examples in the following paragraphs show, a task involving complex-geometry processing (*e.g.*, a complex 3D model but a simple shading scheme) usually requires many more vertex calculations than pixel calculations. A task of complex-pixel processing (*e.g.*, a simple 3D model but a complex pixel shading scheme) usually requires many more pixel calculations than vertex calculations.

56. If a graphics-processing task requires many more vertex calculations than pixel calculations, the pixel shader is (relatively) idle, resulting in wasted pixel resources. *Id.* at 1:60-65, 1:67-2:6. The figures below show a scenario where complex-geometry processing would keep the pixel shader underutilized. In this scenario, a large number of polygons form an image of a car. Ex. 2086, p. 582, Fig. 16.8(a). Each polygon (*i.e.*, a primitive) is represented by vertices, and processing these vertices keeps the vertex shader fully loaded. When constant shading (also called "flat shading") renders the scene (*id.*), there are relatively few pixel calculations because "[c]onstant shading calculates a single intensity value for

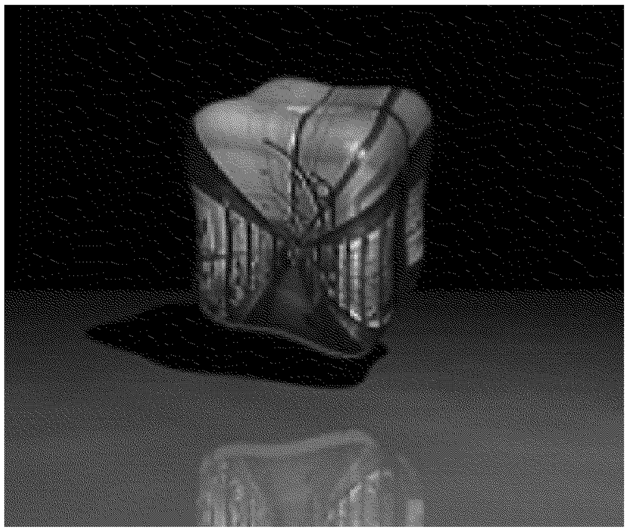
shading an entire polygon." *Id.* at 580. Thus, the pixel shader is only partially utilized. *See* Ex. 2087, p. 13; Figure 5. Unfortunately, the pixel shader cannot use its extra resources to help the vertex shader because the pixel shader is of a separate type and cannot perform vertex calculations.



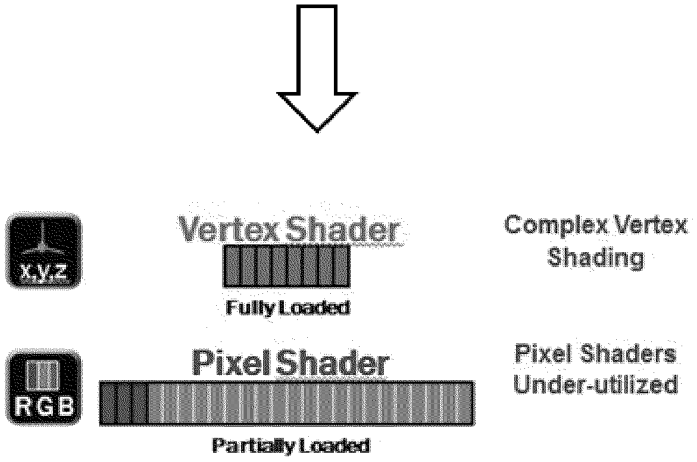
(Ex. 2086, p. 582, Fig. 16.8)



57. Conversely, if a graphics-processing task requires many pixel calculations but has simple geometry, the vertex shader is (relatively) idle, resulting in wasted vertex resources. Ex. 1001, 1:60-65, 1:67-2:6. The figures below show a scenario where complex pixel processing would keep the vertex shader underutilized. In this scenario, a simple 3D model is rendered with shadows, reflections, and texture mapping. Ex. 2087, p. 13, Figure 5. Consequently, the pixel shader becomes the bottleneck of the system because rendering a simple 3D model with shadows, reflections, and texture mapping requires heavy pixel calculations. While the pixel shader is fully loaded, the vertex shader is only partially loaded because a simple 3D model does not have as many vertices to process. *See id.* Unfortunately, the vertex shader cannot use its extra resources to help offload the pixel shader's load because a conventional vertex shader is of a separate type and cannot perform pixel calculations.



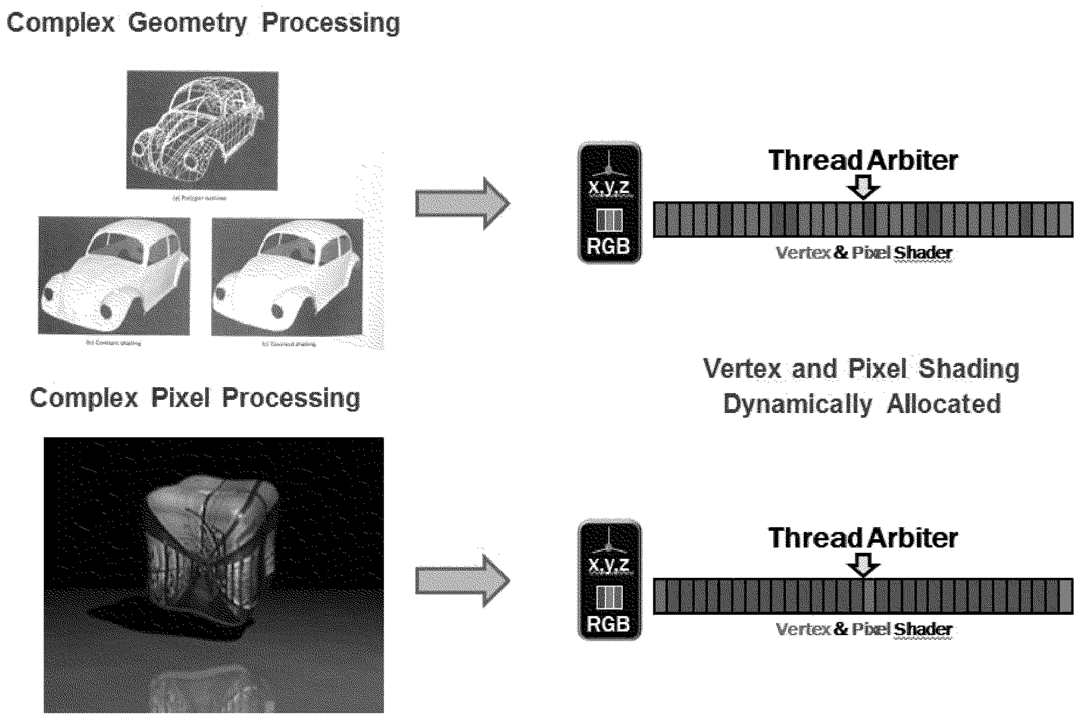
(Ex. 2087, p. 13, Figure 5)



58. Either way, graphics systems using separate vertex and pixel shaders are almost always unable to efficiently use all available resources. Ex. 1001, 1:60-65, 1:67-2:6.

59. As discovered by the inventors of the '871 patent, a single unified shader can perform both vertex and pixel calculations, such that a graphics processing system using one or more unified shaders has more flexibility. Such a system has more flexibility because the system can allocate shader resources more efficiently and balance resource utilization between vertex and pixel calculations.

60. A graphics-processing system with a unified shader can manage shader resources and balance resource utilization by using an arbiter to select between vertex processing and pixel processing. As the figures below show, when a graphics task requires more vertex calculations (*e.g.*, complex-geometry processing), the arbiter can select more vertex command threads for the unified shader to perform vertex calculations (represented in green). When a graphics task requires more pixel calculations (*e.g.*, complex pixel processing), the arbiter can select more pixel command threads for the unified shader to perform pixel calculations (represented in blue). *See* Ex. 2087, pp. 14-15; Figure 7. Either way, the arbiter and the unified shader increase the graphics processing system's performance by reducing underutilization of the shader resources. *See* Ex. 2082, pp. 113-15.



VII. U.S. PATENT NO. 6,897,871

61. The '871 patent is directed to a graphics-processing system having a single, unified shader. Ex. 1001, Abstract. The graphics-processing system includes an arbiter circuit operative to select one of a plurality of inputs in response to a control signal. *Id.* The graphics-processing system also includes a shader. *Id.* The shader is coupled to the arbiter and is operative to process the selected one of the plurality of inputs. *Id.* The shader includes means for performing vertex and pixel operations, such that the shader performs one of the vertex operations or pixel operations based on the selected one of the plurality of inputs. *Id.*

62. The shader in the graphics-processing system also includes a register block, a sequencer, and a processor unit. *Id.* The register block is used to maintain the plurality of selected inputs. *Id.* The sequencer maintains instructions that perform vertex manipulation and pixel manipulation operations. *Id.* And a processor is capable of executing both floating point arithmetic and logical operations on the selected inputs in response to the instructions maintained in the sequencer. *Id.*

63. The unified shader graphics processing system is an improvement over conventional systems - which include a vertex shader and a pixel shader as separate components. *Id.* at 1:55-57. Both of these shaders are required to perform a position and texture transformation and generate an object. *Id.* at 1:57-62. Because both vertex and pixel shaders are required, the graphics processors are large in size and with most real estate being taken up the vertex and pixel shaders. *Id.* at 1:62-65. In addition to the real-estate penalty associated with conventional graphics processors, there is also a corresponding performance penalty associated therewith. *Id.* at 1:66-2:1.

VIII. BACKGROUND ON CHIP DESIGN AND ATI'S CHIP DESIGN

64. In my experience, modern graphics chip production is a two-step process. First, the integrated-circuit designers design a chip almost entirely on a

computer using computer-aided–design ("CAD") tools. The integrated-circuit designers depend on software-based design, simulation, verification, and layout tools. These tools ensure that production integrated circuits function and work as intended. This process can take several months or years. These CAD tools are used to create a chip specification, generally at multiple levels of abstraction that serve as both a detailed specification of the chip and as a model of its structure and function. This has been the predominant design methodology for graphics chips since at least 1990.

65. The CAD tools are used to model and validate the chip design. While the design representation at this stage may resemble software, its primary purpose is to be an accurate representation of a hardware chip design. In the case of hardware description languages like Very High Speed Integrated Circuit Hardware Description Language ("VHDL") or Verilog, the design language is generally the most accurate formal specification of the structure and function of the chip that the design engineer will prepare. It is used to directly create the manufacturing tooling. Only after the integrated-circuit designers are confident that the design will function properly, and the chip design passes commercial specifications, the layout file created by the CAD tools from the design language is sent to a chip-manufacturing facility for fabrication. Since layout files were historically provided on a magnetic tape, this is often called a "tape-out." At this point the design

process has been completed and the manufacturing step is intended to simply reproduce an exact copy of what is described in the layout file. The layout file represents the manufacturing tooling for the chip-manufacturing facility. The chip-manufacturing facility uses this tooling to fabricate a physical integrated circuit, commonly referred to as a "chip."

66. In my experience, although both circuit design and circuit fabrication are both necessary components of chip production, in reality they are separate and distinct activities. Typically, chip design and chip fabrication are performed by different entities, particularly with respect to graphics chips. Ordinarily, circuit designers do not fabricate chips, and chip fabricators do not design circuits.

67. It is my understanding that, the patent owner here, ATI, is a chip-design company. This means that ATI designs integrated circuits, such as chips. ATI does not fabricate chips. Instead, ATI uses software-based CAD tools to design and reduce to practice the chip components claimed in the '871 patent. Only after the components claimed in the '871 patent (along with other chip components) worked for their intended purpose, would ATI generate the tooling and send it for fabrication. Because the '871 patent pertains to the chip-circuit design, the actual reduction to practice of the claims of the '871 patent would have occurred when the RTL code performed all limitations of the claims.

IX. THE CODE FOR ATI'S R400 CHIP

68. I have been asked to review the source code for ATI's R400 chip. This code includes files generated before or on August 5, 2002 by ATI. The source code includes two corresponding design databases that comprise the source code: R400 RTL code and Emulator Code.

69. The R400 Emulator Code is written in a well-known C++ programming language. The R400 Emulator Code includes source code that, when executed, emulates the behavior of the graphics-processing system using software that executes on a computer. C++ is commonly used to specify the function of a software system, but chip designers often also use it to specify and emulate structural aspects of hardware systems, such as, chips, and also to model, validate, and test the functionality and certain structural features of a hardware design.

70. In my experience having both RTL code and C++ code implementation is common in the chip design industry. The C++ code is faster to write and easier to debug by the chip designers. It runs faster, so larger examples of user input can be tested. The chip designers often first write and test the chip design in C++ or another software language. The test results from the chip code in C++ are saved. Next the RTL code (in this case the R400 code) is written in Verilog or another hardware-description language and is compared against the test

results generated using the C++ code. By comparing two different descriptions of the hardware implementation, it is more likely that errors can be found and removed.

71. The R400 RTL code is implemented in a hardware-description language (HDL), called Verilog. Verilog is a hardware-description language used to design and specify hardware systems. That is, Verilog describes behavior of a hardware circuit in terms of inputs, outputs, state machines, logic equations, and modules. When a module is declared in Verilog, the declaration is definitional. This serves as a specification of function and structure. Copies of that module can then be instantiated by specifying the inputs and outputs that carry information to and from a particular copy of the module. This instructs the CAD tools to create a copy of the specified circuits in each final product. It is possible to have multiple copies of a module, with the inputs and outputs of each copy separately specified in the design. The logic equations for the module, which describe how the module operates based on different inputs, are also specified. This logic can be combinational, representing a set of basic logic gates, or sequential, which can include a state machine that controls the operation over time. There are many different ways to write these logic equations, but each is converted to a set of basic logic gates by the CAD tools. From the files produced by the R400 RTL code, a chip manufacturer is able to manufacture a hardware circuit that includes structure

and behavior described in the R400 RTL code. This is a standard practice in any modern graphics integrated circuit design.

72. Here, in the R400 program, Verilog was used to validate the integrated-circuit version of the graphics-processing system recited in claims 1, 2, 3, 5, 6, 8, 9, 10, 11, 13, 15, 17, 18, and 20. At least one version of the R400 RTL code which discloses all elements of these includes the files generated before or on August 5, 2002. These files are attached as Exhibits 2093-2096, 2098-2101, 2104-2118.

73. I have compared each element of claims 1, 2, 3, 5, 6, 8, 9, 10, 11, 13, 15, 17, 18, and 20 to the R400 RTL code and the corresponding files, functions, and interfaces using the broadest reasonable construction standard for all terms that the Board did not construe. For the term "means for performing vertex operations and pixel operations and performing one of the vertex operations or pixel operations based on the selected one of the plurality of inputs," I applied the Board's construction—namely, *a register, an instruction sequencer capable of providing instructions for performing vertex operations and pixel operations, and a processor capable of floating point, arithmetic, and logical operations on a selected input*. I point to thefiles, pages and line numbers in the RTL that disclose

each element recited in these claims. In my opinion, the R400 RTL code discloses all elements of claims 1, 2, 3, 5, 6, 8, 9, 10, 11, 13, 15, 17, 18, and 20.

74. The R400 RTL code includes the `sq.v`, `sp.v`, `sx.v`, `pa.v`, and `sc.v` files and their corresponding sub-files and referenced modules that specify and generate a hardware circuit which is a graphics-processing system as recited in claims 1, 2, 3, 5, 6, 8, 9, 10, 11, 13, 15, 17, 18, and 20. In particular, the `sq.v` file specifies and generates a sequencer which includes parts of an arbiter circuit, and arbiter, and an instruction store. The `sp.v` specifies and generates a shader, a register, the selection multiplexer, a computation element and a processor unit. The `sx.v` specifies and generates a shader export block which includes a parameter cache. The `pa.v` specifies and generates a primitive assembly block which includes a position cache. The `sc.v` file specifies and generates a raster engine (also referred to as a rasterizer or a scan converter).

75. I cite to the R400 RTL source code using the following format: (`sq.v`, 1:1-10). This example citation points to exhibit `sq.v`, at page 1, lines 1-10.

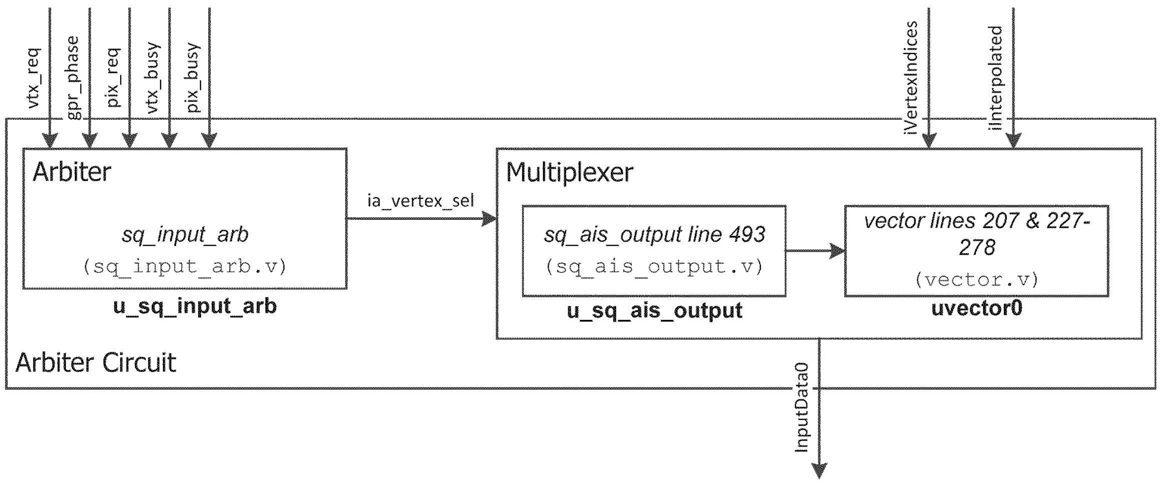
A. *Claim 1*

1. *The Preamble*

76. The preamble of claim 1 recites "*A graphics processor, comprising:*" The files, `sq.v` and `sp.v` (and their referenced modules) define the hardware block component of the graphics-processing system.

2. *The Arbiter Circuit*

77. The first element of claim 1 recites "*an arbiter circuit for selecting one of a plurality of inputs in response to a control signal.*" I have generated a visual representation of the components, as I understand them, based on the R400 RTL code, that describe an arbiter circuit in a figure below.



78. The `sq.v` and `sp.v` file instantiate blocks of an arbiter circuit. The arbiter circuit includes an arbiter instantiated as `u_sq_input_arb`. (`sq.v`,

28:11-29:7.) The *u_sq_input_arb* is specified as an *sq_input_arb* module in *sq_input_arb.v*.

79. The arbiter *u_sq_input_arb* is coupled to a multiplexer, implemented in *u_sq_ais_output* and at least one of the four vector units, *uvector0, uvector1, uvector2,* and *uvector3*. The *u_sq_ais_output* module is instantiated in *sq.v*. (*sq.v*, 77:20-81:4) and the four vector units are instantiated in *sp.v* (*sp.v*, 15:6-18:16).

80. The arbiter in the *sq_input_arb* module receives five input signals *vtx_req, gpr_phase, pix_req, vtx_busy*, and *pix_busy* signals which are replicated below.

```
module sq_input_arb
(
   vtx_req,      // request from VISM
   vtx_busy,     // busy from VISM - tells arb to keep gpr write
mux set to verts
   pix_req,      // request from PISM
   pix_busy,     // busy from PISM - tells arb to keep gpr write
mux set to pixels
   gpr_phase,    //
…
   input [0:0] vtx_req;
   input [0:0] vtx_busy;
   input [0:0] pix_req;
   input [0:0] pix_busy;

   input [1:0] gpr_phase;
…
)
```

(*sq_input_arb.v*, 2:4-11, 3:8-13.)

81. The `vtx_req` control signal is a request from a vertex input state machine to process a vertex inputs. The `pix_req` control signal is a request from a pixel input state machine to process a pixel input. The arbiter generates a control signal based on these five input signals, as replicated below:

```
[/ next state logic
   always @(
           vtx_req or pix_req or vtx_busy or pix_busy or
gpr_phase or
         current_state
           )
      begin
       // default assignments
       next_state = IDLE;
       next_vtx_gnt = LO;
       next_pix_gnt = LO;
       next_vtx_sel = LO;

       case (current_state)
         IDLE:
           begin
            // - assert grants based on gpr phase
            //   - gnt is reg'd out, so need to look for phase
before the one that lines up
              // - the phase for pix gnt is calculated based on
interp latency
            if ( vtx_req & (gpr_phase == `SQ_ID_PHASE) )
              begin
               next_vtx_gnt = HI;
               next_vtx_sel = HI;
                    next_state = V_XFER;
                 end
            else if ( pix_req & (gpr_phase == `SQ_PV_PHASE) )
              begin
               next_pix_gnt = HI;
                    next_state = P_XFER;
                 end

         end

         V_XFER:
           begin
            // - hold vtx_sel high while VISM is busy
            if ( vtx_busy )
              begin
              next_vtx_sel = HI;
              next_state = V_XFER;
             end
           else
             begin
              next_state = IDLE;
             end
        end

        P_XFER:
           begin
           // - first check if there's another pix req (and no
vtx req)
           //   - if so, grant it and stay here
           // - otherwise continue to hold vtx_sel low while PISM
is busy

           if ( pix_req & ~vtx_req & (gpr_phase == `SQ_PV_PHASE)
)
             begin
              next_pix_gnt = HI;
                  next_state = P_XFER;
               end
           else if ( pix_busy )
             begin
              next_state = P_XFER;
             end
           else
             begin
              next_state = IDLE;
             end
        end

      endcase // case(current_state)

     end // always @ (*)
```

(*Id.* at 6:16-9:10.)

82. The R400 RTL code above shows that the `sq_input_arb` arbiter circuit uses `vtx_req` to select a vertex input, and `pix_req` to select a pixel input, with the vertex input having a priority over the pixel input if both the vertex input state machine and the pixel input state machine simultaneously request that their respective inputs are selected.

83. If the `sq_input_arb` arbiter circuit selects a vertex input, the R400 RTL code above sets `next_vtx_gnt` and `next_vtx_sel` to "HI". If the `sq_input_arb` arbiter circuit selects a vertex input, the R400 RTL code above sets `next_vtx_gnt` and `next_vtx_sel` to "HI". If the `sq_input_arb` arbiter circuit selects a pixel input, the R400 RTL code above sets `next_pix_gnt` to "HI". The arbiter then indicates to the selection multiplexer and shader that the vertex input or the pixel input has been selected using the R400 RTL code below:

```
        output [0:0] vtx_gnt;
        output [0:0] pix_gnt;
        output [0:0] vtx_sel;
...
         begin
              current_state <= next_state;
              vtx_gnt <= next_vtx_gnt;
              vtx_sel <= next_vtx_sel;
              pix_gnt <= next_pix_gnt;
         end
```

(*Id.* at 3:15-17, 6:7-12.)

84. As shown in the R400 RTL code above, the `sq_input_arb` arbiter sets the output signals `vtx_gnt` and `pix_gnt` to a "HI" or "LO" value from `next_vtx_gnt` and `next_pix_gnt` respectively, and sets `vtx_sel` to identify whether the vertex input or the pixel input was selected. The `sq_input_arb` arbiter outputs the `vtx_sel` signal as `ia_vertex_sel` at

`sq.v`, 29:1. The signal `ia_vertex_sel` corresponds to the claimed control signal.

85. The arbiter passes the `vtx_sel` signal to the `sq_ais_output` module called `u_sq_ais_output` as `ia_vertex_sel` signal. As I discussed above `u_sq_ais_output` is instantiated in the `sq` module (`sq.v`, 77:20-81:4) and is defined in the `sq_ais_output` module in `sq_ais_output.v`. The `u_sq_ais_output` receives the `ia_vertex_sel` at 79:8 in `sq.v` and 3:17 of `sq_ais_output` module.

86. The `sq_ais_output` module uses the control signal `ia_vertex_sel` to generate a control signal `SQ_SP_gpr_input_sel` (shown as line 493 in my figure above), which becomes an `SQ_SP_gpr_input_mux` signal in `sq.v`. (*See* `sq_ais_output.v`, 21:7; `sq.v`, 80:6.)

87. The *sp* module receives the `SQ_SP_gpr_input_mux` signal as `SQ_SP_gpr_input_mux` in `sp.v` at 2:7 and 9:11. The `sp` module also instantiates four instances of *vector* units: `uvector0, uvector1,` `uvector2`, and `uvector3`. (*See* `sp.v`, 15:6-16:18.) Each of the `vector0-3` units is defined in the `vector` module in `vector.v`. Each of the four *vector*

units receives the *SQ_SP_gpr_input_mux* signal referred to as *q_sq_gpr_input_mux*, as shown at 16:4 of *uvector0*, at 16:18 for *uvector1*, at 17:19 for *uvector2*, and 18:11 for *uvector3* in *sp.v*. The *q_sq_gpr_input_mux* is the control signal provided by the arbiter.

88. Each of the vector units *uvector0-3* receives a plurality of inputs. These inputs include the interpolated data (for pixel operations) and vertex indices (for vertex operations). The interpolated data is generated using the *uinterpolator instance* of the *interpolator* module described in *interpolator.v*. The R400 RTL code which instantiates the *uinterpolator* instance is replicated below.

```
wire [127:0]   Interpolated0,
Interpolated1,Interpolated2,Interpolated3;
```

(*sp.v*, 14:1.)

```
//-----------------------------------------------------------------
----------------------------//
   //Interpolation Units---------------------------------------------
-------------------------------//
   //--------------------------------------------------------------
-------------------------------//
   interpolator uinterpolator(.oInterpolated0(Interpolated0),
.oInterpolated1(Interpolated1), .oInterpolated2(Interpolated2),
.oInterpolated3(Interpolated3), .sx_sp_vtx_data0(q_sx_vtx_data0),

.sx_sp_vtx_delta10(q_sx_vtx_data1),.sx_sp_vtx_delta20(q_sx_vtx_da
ta2),
.sq_sp_interp_ijline(q_sq_interp_ijline),.sq_sp_interp_valid(q_sq
_interp_valid),  .sq_sp_interp_buff_swap(q_sq_interp_buff_swap),
.sc_sp_data(q_sc_data),.sc_sp_valid(q_sc_valid),.sq_sp_interp_mod
e(q_sq_interp_mode), .sc_sp_type(q_sc_type),
.sc_sp_quad_last(q_sc_last_quad),
.sclk(sclk),.srst(srst));
```

(*Id.* at 14:5-19.)

89. The input data to `uinterpolator` comes from the `q_sc_data` signal which is sent by a raster engine. The `sp` module receives `q_sc_data` using the SC_SP interface as `SC_SP_data`. (*Id.* at 2:7, 10:40.)

90. The output from `uinterpolator`, including `Interpolated0`, `Interpolated1`, `Interpolated2`, and `Interpolated3` is passed to each of the vector units `uvector0-3`. For example, `uvector0` receives `Interpolated0`, `uvector1` receives `Interpolated1`, `uvector2` receives `Interpolated2`, and `uvector3` receives `Interpolated3`. (*Id.* at 16:6, 17:1, 17:20, and 18:12.)

91. Each of the vector units `uvector0-3` also receive vertex indices as one of a plurality of inputs. The vertex indices are passed to the vector units `uvector0-3`. For example, the vertex indices are generated using `usp_vsr_ctl` defined in the `sp_vsr_ctl` module in `sp_vsr_ctl.v`. The R400 RTL code which outputs the vertex indices is replicated below.

```
//------------------------------------------------------------------
-----------------------------//
   //Vertex Indices Staging registers and Control
   //--------------------------------------------------------------
-------------------------------//
   sp_vsr_ctl usp_vsr_ctl(.ovtx_index0(VertexIndex0),
.ovtx_index1(VertexIndex1).ovtx_index2(VertexIndex2),
.ovtx_index3(VertexIndex3), .isq_vsr_data(q_sq_vsr_data),
.isq_vsr_double(q_sq_vsr_double), .isq_vsr_valid(q_sq_vsr_valid),
.isq_vsr_read(q_sq_vsr_read), .sclk(sclk),.srst(srst));
```

(*Id.* at 14:22-15:4)

92. The input data to `usp_vsr_ctl` comes from `q_sq_vsr_data`. The `sp` module receives `q_sc_data` using the `SQ_SP` interface as `SQ_SP_vsr_data`. (*Id.* at 2:9, 11:7, 11:17.)

93. The outputs from `usp_vsr_ctl`, including `VertexIndex0`, `VertexIndex1`, `VertexIndex2`, `VertexIndex3` are passed to each of the respective vector units `uvector0-3`. For example, `uvector0` receives `VertexIndex0`, `uvector1` receives `VertexIndex1`, `uvector2` receives `VertexIndex2`, and `uvector3` receives `VertexIndex3`. (*Id.* at 16:6, 17:2, 17:21, and 18:13.)

94. The arbiter circuitry in `vector` units `uvector0-3` selects one of a plurality of inputs from the vertex indicies (which are the vertex data) and the interpolated pixel inputs (which are the pixel data). For example, the vector module uses the `sq_sp_gpr_input_mux` parameter provided by the arbiter to select the vertex data input (`iVertexIndices`) or the pixel data input (`iInterpolated`), using the R400 RTL code replicated below:

```
//-----------------------------------------------------------
---------------------------------------------------
```

```
   //Muxing logic to select from data comming from the
Interpolators(in reality more than just interpolated
data....there can be
   //also faceness and XY data), AutoCount data and Vertex
Indices comming from the staging registers.
   //Each MACC unit has its own mux logic since the controls are
phased out by one cycle from one MACC to the other.
   //-----------------------------------------------------------
------------------------------------------------------------
   //muxing logic for the inputs of the first MACC
   always @(/*AUTOSENSE*/iAutoCount or iInterpolated or
iVertexIndices
         or sq_sp_gpr_input_mux)
     begin
      case(sq_sp_gpr_input_mux)
        2'b00: InputData0 = iAutoCount ;
        2'b01: InputData0 = iInterpolated ;
        2'b10: InputData0 = iVertexIndices ;
        default: InputData0 = iInterpolated;
      endcase // case(sq_sp_gpr_input_mux)
     end

   //muxing logic for the inputs of the second MACC
   always @(/*AUTOSENSE*/iAutoCount or iInterpolated or
iVertexIndices
         or q0_gpr_input_mux)
     begin
      case(q0_gpr_input_mux)
        2'b00: InputData1 = iAutoCount ;
        2'b01: InputData1 = iInterpolated ;
        2'b10: InputData1 = iVertexIndices ;
        default: InputData1 = iInterpolated;
      endcase // case(q0_gpr_input_mux)
     end

   //muxing logic for the inputs of the third MACC
   always @(/*AUTOSENSE*/iAutoCount or iInterpolated or
iVertexIndices
         or q1_gpr_input_mux)
     begin
      case(q1_gpr_input_mux)
        2'b00: InputData2 = iAutoCount ;
        2'b01: InputData2 = iInterpolated ;
        2'b10: InputData2 = iVertexIndices ;
        default: InputData2 = iInterpolated;
      endcase // case(q1_gpr_input_mux)
     end

   //muxing logic for the inputs of the fourth MACC
   always @(/*AUTOSENSE*/iAutoCount or iInterpolated or
iVertexIndices
         or q2_gpr_input_mux)
```

```
begin
 case(q2_gpr_input_mux)
   2'b00: InputData3 = iAutoCount ;
   2'b01: InputData3 = iInterpolated ;
   2'b10: InputData3 = iVertexIndices ;
   default: InputData3 = iInterpolated;
 endcase // case(q2_gpr_input_mux)
end
```

(`vector.v`, 10:2-12:6 (emphasis added).)

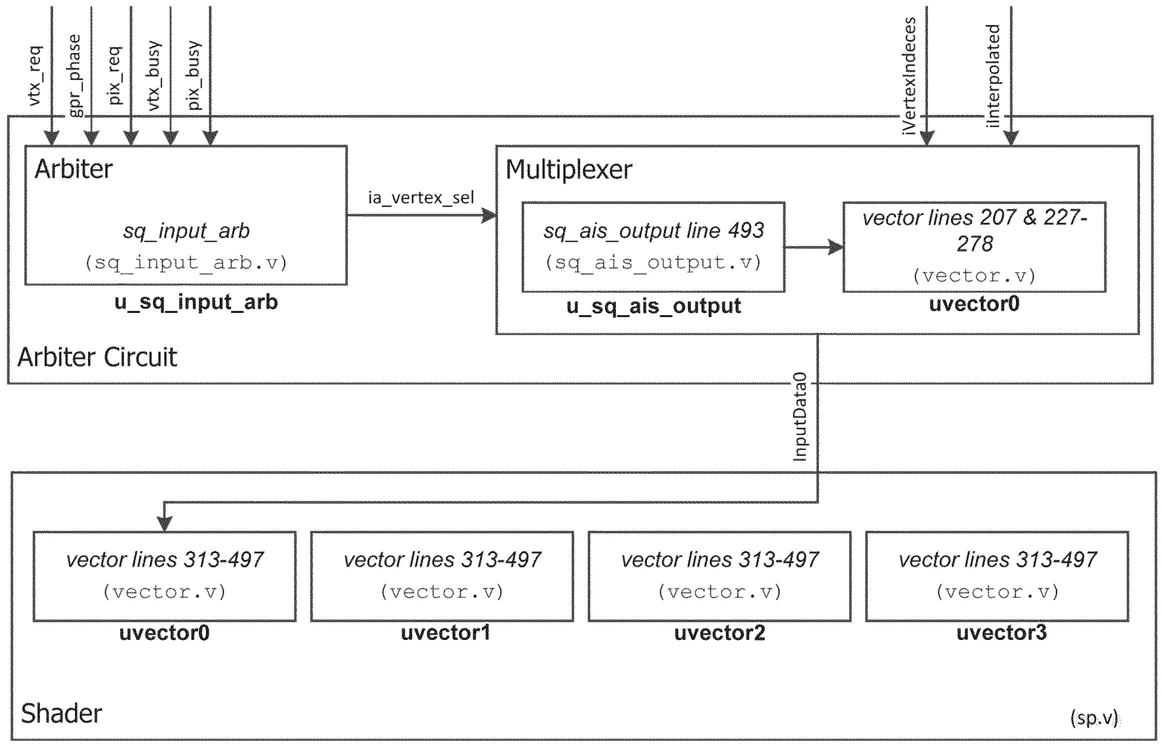
95. This is also shown in my figure above, where lines 207 & 227-228 map to the `sq_sp_gpr_input_mux` parameter and `vector.v`, 10:2-12:6.

96. The selected input is stored in `InputData0`, `InputData1`, `InputData2`, and `InputData3`.

97. As explained above, R400 RTL code specifies an arbiter circuit for selecting one of a plurality of inputs in response to a control signal.

3. The shader coupled to the arbiter circuit

98. The second element of claim 1 recites "*a shader, coupled to the arbiter circuit*." I have generated a visual representation of the components, as I understand them, based on the R400 RTL code, that describe how an arbiter circuit is coupled to the shader, in a figure below:



99. The shader includes the scalar and vector processing pipes and registers in the *sp* module. (*sp.v*.) For example, *sp.v* instantiates four vector units (described in Section IX.A.2). The *sq* module and the *sp* module connect the arbiter circuit to the shader components, such as, the vector units.

100. The arbiter circuit is coupled to the shader via a number of selected data lines from the selection multiplexer. These signals include *InputData0*, *InputData1*, *InputData2*, and *InputData3*. These signals are then provided to the *macc_gpr* modules within the vector units to couple the selection multiplexer in the arbiter circuit with the input of the shader. This is shown using

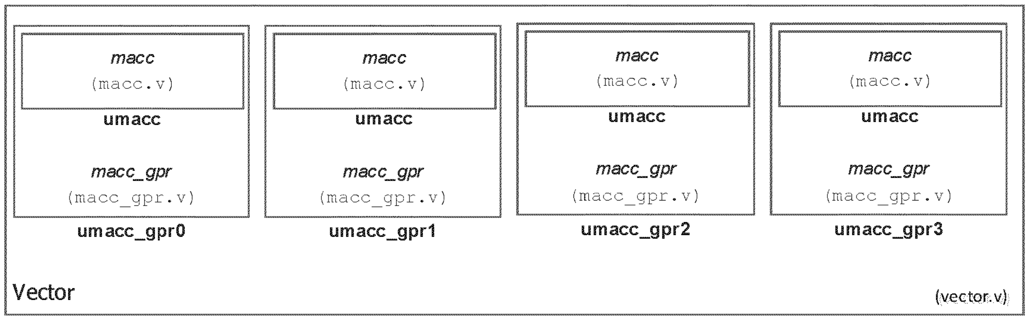
the R400 RTL code below, and also in my figure above as lines 313-497 which map to `vector.v` at 13:9-22:8.

```
//-------------------------------------------------------------------
-----------------------------------------------------------------
   //Instantiation of all four MACC units that create a Vector Unit
   //----------------------------------------------------------------
-------------------------------------------------------------------
   macc_gprumacc_gpr0(.oVectorOutput(VectorResult0)
,.oScalarInput(ScalarInput0), .oScalarOpcode(ScalarOpcode0),
.oRegData(RegData0), .oexport_dst(sq_sp_exp_dst),
.sq_sp_instruct(sq_sp_instruct), .sq_sp_instruct_start
(sq_sp_instruct_start), .sq_sp_gpr_rd_addr(sq_sp_gpr_rd_addr),
.sq_sp_gpr_wr_addr(sq_sp_wr_addr),.sq_sp_wr_ena(sq_sp_wr_ena),.sq_sp_m
em_rd_ena(sq_sp_mem_rd_ena),.sq_sp_mem_wr_ena(sq_sp_mem_wr_ena),.sq_sp
_gpr_cmask(sq_sp_channel_mask), .sq_sp_gpr_phase_mux
(sq_sp_gpr_phase_mux), .iInterpolated(InputData0),
.sq_sp_constant(sq_sp_constant),.iScalarData(ScalarData),.tp_sp_data(t
p_sp_data), .tp_sp_gpr_dst(tp_sp_gpr_dst), .tp_sp_gpr_cmask
(tp_sp_gpr_cmask), .tp_sp_data_valid(tp_sp_data_valid),.sclk(sclk),
.srst(srst));

   macc_gpr umacc_gpr1(.oVectorOutput(VectorResult1) ,.oScalarInput
(ScalarInput1),.oScalarOpcode(ScalarOpcode1),.oRegData(RegData1),.sq_s
p_instruct(q0_instruct),.sq_sp_instruct_start(q0_instruct_start),.sq_s
p_gpr_rd_addr(q0_gpr_rd_addr),.sq_sp_gpr_wr_addr(q0_gpr_wr_addr),.sq_s
p_wr_ena(q0_gpr_we),.sq_sp_mem_rd_ena(q0_gpr_mre),.sq_sp_mem_wr_ena(q0
_gpr_mwe),.sq_sp_gpr_cmask(q0_gpr_cmask),.sq_sp_gpr_phase_mux(q0_gpr_p
hase_mux),.iInterpolated(InputData1),.sq_sp_constant(sq_sp_constant),.
iScalarData(q0_ScalarData),.tp_sp_data(tp_sp_data),
.tp_sp_gpr_dst(q0_tp_gpr_dst), .tp_sp_gpr_cmask(q0_tp_gpr_cmask),
.tp_sp_data_valid(q0_tp_data_valid),.sclk(sclk), .srst(srst));

   macc_gpr umacc_gpr2(.oVectorOutput(VectorResult2), .oScalarInput
(ScalarInput2),.oScalarOpcode(ScalarOpcode2),.oRegData(RegData2),.sq_s
p_instruct(q1_instruct),.sq_sp_instruct_start(q1_instruct_start),.sq_s
p_gpr_rd_addr(q1_gpr_rd_addr),.sq_sp_gpr_wr_addr(q1_gpr_wr_addr),.sq_s
p_wr_ena(q1_gpr_we),.sq_sp_mem_rd_ena(q1_gpr_mre),.sq_sp_mem_wr_ena(q1
_gpr_mwe),.sq_sp_gpr_cmask(q1_gpr_cmask),.sq_sp_gpr_phase_mux(q1_gpr_p
hase_mux),.iInterpolated(InputData2),.sq_sp_constant(sq_sp_constant),.
iScalarData(q1_ScalarData),.tp_sp_data(tp_sp_data),.tp_sp_gpr_dst(q1_t
p_gpr_dst), .tp_sp_gpr_cmask(q1_tp_gpr_cmask), .tp_sp_data_valid
(q1_tp_data_valid),.sclk(sclk), .srst(srst));

   macc_gpr umacc_gpr3(.oVectorOutput(VectorResult3), .oScalarInput
(ScalarInput3),.oScalarOpcode(ScalarOpcode3),.oRegData(RegData3),.sq_s
p_instruct(q2_instruct),.sq_sp_instruct_start(q2_instruct_start),.sq_s
p_gpr_rd_addr(q2_gpr_rd_addr),.sq_sp_gpr_wr_addr(q2_gpr_wr_addr),
.sq_sp_wr_ena(q2_gpr_we),.sq_sp_mem_rd_ena(q2_gpr_mre),.sq_sp_mem_wr_e
na(q2_gpr_mwe),.sq_sp_gpr_cmask(q2_gpr_cmask),.sq_sp_gpr_phase_mux(q2_
gpr_phase_mux),.iInterpolated(InputData3),.sq_sp_constant(sq_sp_consta
nt), .iScalarData(q2_ScalarData), .tp_sp_data(tp_sp_data),
.sclk(sclk),.tp_sp_gpr_dst(q2_tp_gpr_dst), .tp_sp_gpr_cmask
(q2_tp_gpr_cmask), .tp_sp_data_valid(q2_tp_data_valid),.srst(srst));
```

(`vector.v`, 14:1-16:7; (emphasis added).)

a. The shader is operative to process the selected one of the plurality of inputs

101. The shader is "*operative to process the selected one of the plurality of inputs*." Based on my understanding of R400 RTL code, I have generated a figure below which represents my understanding of the components and describes the code with the reference to the figure.



102. As shown in Section IX.A.2, each of the four `vector` units, `uvector0`, `uvector1`, `uvector2`, and `uvector3` process an input selected from the interpolated pixel data or the vertex indices which has been provided on the signals `InputData0`, `InputData1`, `InputData2`, and `InputData3`.

The *InputData0*, *InputData1*, *InputData2*, and *InputData3* signals are each a selected one of the plurality of inputs.

103. To process the *InputData0*, *InputData1*, *InputData2*, and *InputData3* signals, each vector unit instantiates four MACC modules: *umacc_gpr0*, *umacc_gpr1*, *umacc_gpr2*, and *umacc_gpr3*. The MACC modules (*umacc_gpr0*, *umacc_gpr1*, *umacc_gpr2*, and *umacc_gpr3*) receive the corresponding selected input data (*InputData0*, *InputData1*, *InputData2*, and *InputData3*) and corresponding instructions (*sq_sp_instruct*, *q0_instruct*, *q1_instruct*, and *q2_instruct*) using the R400 RTL code below:

```
//----------------------------------------------------------------------
-------------------------------------------------------------
   //Instantiation of all four MACC units that create a Vector Unit
   //-------------------------------------------------------------------
----------------------------------------------------------------
   macc_gprumacc_gpr0(.oVectorOutput(VectorResult0)
,.oScalarInput(ScalarInput0), .oScalarOpcode(ScalarOpcode0),
.oRegData(RegData0), .oexport_dst(sq_sp_exp_dst),
.sq_sp_instruct(sq_sp_instruct), .sq_sp_instruct_start
(sq_sp_instruct_start), .sq_sp_gpr_rd_addr(sq_sp_gpr_rd_addr),
.sq_sp_gpr_wr_addr(sq_sp_wr_addr),.sq_sp_wr_ena(sq_sp_wr_ena),.sq_sp_m
em_rd_ena(sq_sp_mem_rd_ena),.sq_sp_mem_wr_ena(sq_sp_mem_wr_ena),.sq_sp
_gpr_cmask(sq_sp_channel_mask), .sq_sp_gpr_phase_mux
(sq_sp_gpr_phase_mux), .iInterpolated(InputData0),
.sq_sp_constant(sq_sp_constant),.iScalarData(ScalarData),.tp_sp_data(t
p_sp_data), .tp_sp_gpr_dst(tp_sp_gpr_dst), .tp_sp_gpr_cmask
(tp_sp_gpr_cmask), .tp_sp_data_valid(tp_sp_data_valid),.sclk(sclk),
.srst(srst));

   macc_gpr umacc_gpr1(.oVectorOutput(VectorResult1) ,.oScalarInput
(ScalarInput1),.oScalarOpcode(ScalarOpcode1),.oRegData(RegData1),.sq_s
p_instruct(q0_instruct),.sq_sp_instruct_start(q0_instruct_start),.sq_s
p_gpr_rd_addr(q0_gpr_rd_addr),.sq_sp_gpr_wr_addr(q0_gpr_wr_addr),.sq_s
p_wr_ena(q0_gpr_we),.sq_sp_mem_rd_ena(q0_gpr_mre),.sq_sp_mem_wr_ena(q0
_gpr_mwe),.sq_sp_gpr_cmask(q0_gpr_cmask),.sq_sp_gpr_phase_mux(q0_gpr_p
hase_mux),.iInterpolated(InputData1),.sq_sp_constant(sq_sp_constant),.
iScalarData(q0_ScalarData),.tp_sp_data(tp_sp_data),
.tp_sp_gpr_dst(q0_tp_gpr_dst), .tp_sp_gpr_cmask(q0_tp_gpr_cmask),
.tp_sp_data_valid(q0_tp_data_valid),.sclk(sclk), .srst(srst));

   macc_gpr umacc_gpr2(.oVectorOutput(VectorResult2), .oScalarInput
(ScalarInput2),.oScalarOpcode(ScalarOpcode2),.oRegData(RegData2),.sq_s
p_instruct(q1_instruct),.sq_sp_instruct_start(q1_instruct_start),.sq_s
p_gpr_rd_addr(q1_gpr_rd_addr),.sq_sp_gpr_wr_addr(q1_gpr_wr_addr),.sq_s
p_wr_ena(q1_gpr_we),.sq_sp_mem_rd_ena(q1_gpr_mre),.sq_sp_mem_wr_ena(q1
_gpr_mwe),.sq_sp_gpr_cmask(q1_gpr_cmask),.sq_sp_gpr_phase_mux(q1_gpr_p
hase_mux),.iInterpolated(InputData2),.sq_sp_constant(sq_sp_constant),.
iScalarData(q1_ScalarData),.tp_sp_data(tp_sp_data),.tp_sp_gpr_dst(q1_t
p_gpr_dst), .tp_sp_gpr_cmask(q1_tp_gpr_cmask), .tp_sp_data_valid
(q1_tp_data_valid),.sclk(sclk), .srst(srst));

   macc_gpr umacc_gpr3(.oVectorOutput(VectorResult3), .oScalarInput
(ScalarInput3),.oScalarOpcode(ScalarOpcode3),.oRegData(RegData3),.sq_s
p_instruct(q2_instruct),.sq_sp_instruct_start(q2_instruct_start),.sq_s
p_gpr_rd_addr(q2_gpr_rd_addr),.sq_sp_gpr_wr_addr(q2_gpr_wr_addr),
.sq_sp_wr_ena(q2_gpr_we),.sq_sp_mem_rd_ena(q2_gpr_mre),.sq_sp_mem_wr_e
na(q2_gpr_mwe),.sq_sp_gpr_cmask(q2_gpr_cmask),.sq_sp_gpr_phase_mux(q2_
gpr_phase_mux),.iInterpolated(InputData3),.sq_sp_constant(sq_sp_consta
nt), .iScalarData(q2_ScalarData), .tp_sp_data(tp_sp_data),
.sclk(sclk),.tp_sp_gpr_dst(q2_tp_gpr_dst), .tp_sp_gpr_cmask
(q2_tp_gpr_cmask), .tp_sp_data_valid(q2_tp_data_valid),.srst(srst));
```

(`vector.v`, 14:1-16:7; (emphasis added))

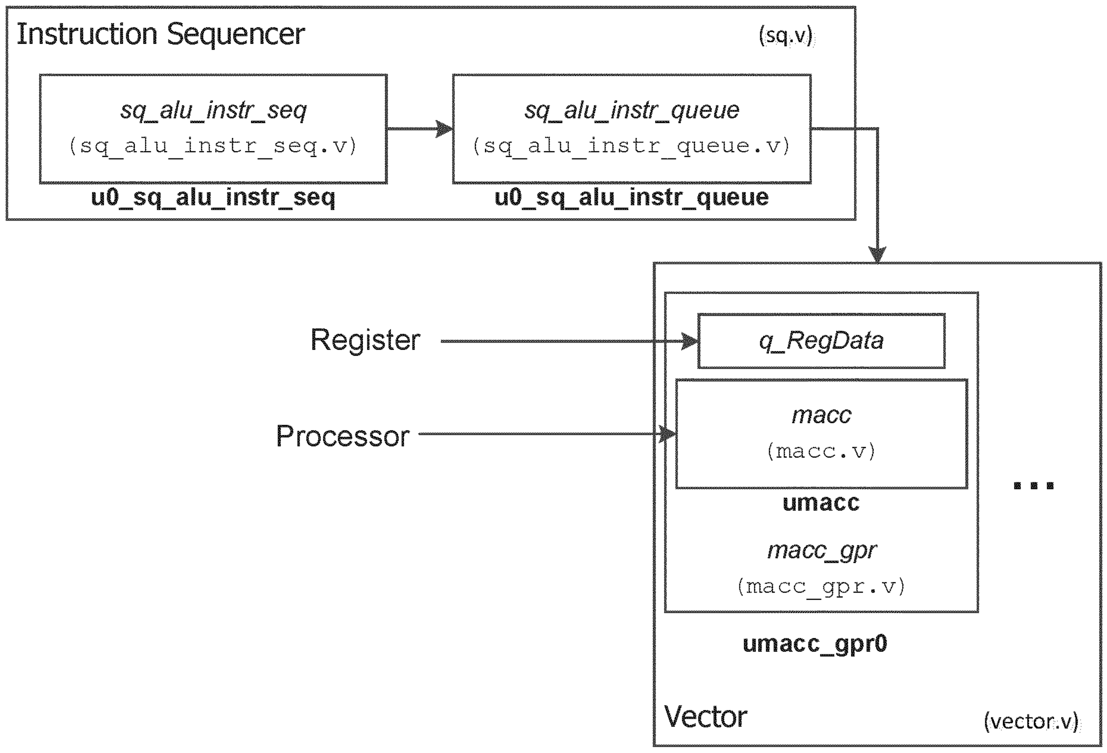
104. The `macc_gpr` module inputs the selected data as the `iInterpolated` signal, as shown in `macc_gpr.v` at 1:20 and 2:11. The *macc_gpr* module is operative to process the selected one of the plurality of inputs (the `iInterpolated` signal). (*See* `macc_gpr.v`.)

b. The shader including means for performing vertex operations and pixel operations

105. The shader also includes "means for performing vertex operations and pixel operations."

106. It is my understanding that the Board construed the above term to include a register, an instruction sequencer capable of providing instructions for performing vertex operations and pixel operations, and a processor capable of floating point arithmetic and logical operations on a selected input. Here, I opine on whether the R400 RTL code includes the functionality and the corresponding structure as construed by the Board.

107. Based on my understanding of the R400 RTL code, I have generated a figure below which represents my understanding of the components, and describe the code with reference to the figure.



108. The *macc_gpr* module includes a register that stores data. For example, the *macc_gpr* module includes register file output signal *RegData* that provides data retrieved from a register file memory called *ugpr_mem*" of module type "*rfsd2_128x128cm2sw1".* (*macc_gpr.v*, 8:1-12:20.) The values from *RegData* are then stored in the register called *q_RegData*. (*macc_gpr.v*, 3:15 and 15:13.) As I describe below, *q_RegData* stores the selected input. As such, the shader includes a register.

109. The shader also includes an instruction sequencer capable of providing instructions. I discuss the instruction sequencer in greater detail below in Section IX.K.3., but briefly; the *sq* module includes an instruction sequencer. The instruction sequencer passes instructions to the shader included in the *sp* module using the *SQ_SP* interface. The interface includes the *SQ_SP_instruct* signal which provides the instruction. (*sq.v*, 2:17, 12:9, and 80:11.)

110. The *sp* module receives the instruction using the *SQ_SP* interface, and converts the instruction into *q_sq_instruct*, as shown using the R400 RTL code below:

```
module sp(/*AUTOARG*/
…
   SQ_SP_instruct_start, SQ_SP_instruct, SQ_SP_stall,
…

input [20:0]   SQ_SP_instruct;

…

ati_dff_in #(21) sq_instruct(sclk,SQ_SP_instruct,q_sq_instruct);
```

(`sp.v`, 1:18, 2:3, 6:13, 7:2.)

111. The *sp* module passes the `q_sq_instruct` instruction to the vector units `uvector0`, `uvector1`, `uvector2`, and `uvector3`. (*Id.* at 14:18-18:2.) Each of the vector units `uvector0`, `uvector1`, `uvector2`, and `uvector3` receives the instruction as `q_sq_instruct` and converts the instruction to `sq_sp_instruct`. (*Id.* at 15:20, 16:9, 17:10, 18:13; and `vector.v`, 1:19, 2:7.)

112. The `vector` unit passes the `q_sq_instruct` instruction to the `macc_gpr0`, `macc_gpr1`, `macc_gpr2`, or `macc_gpr3` modules, as `sq_sp_instruct`, `q0_instruct`, `q1_instruct`, and `q2_instruct`. (`vector.v`, 8:12-14, 14:10, 14:26, 15:11, 15:27.) The four instances of the `mac_gpr` module (the `macc_gpr0`, `macc_gpr1`, `macc_gpr2`, or `macc_gpr3`) receive the instruction as `sq_sp_instruct` and pass the instruction to a *MACC* module called `umacc`, which is replicated using the R400 RTL code below:

```
    macc umacc(.oResult(VectorResult), .oScalarOpcode(oScalarOpcode)
    ,.oScalarInput(oScalarInput),.oExportDst(oexport_dst),
.iRegData(q_RegData),.iConstantData(sq_sp_constant),.iScalarData(
iScalarData), .iInstruction(sq_sp_instruct),
.iInstStart(sq_sp_instruct_start), .sclk(sclk), .srst(srst));
```

(*macc_gpr.v*, 3:17-21.)

113. The *MACC* module receives instructions from the sequencer using the *sq_sp_instruct* signal and converts it to *iInstruction*. The *MACC* module also receives *q_RegData*, and manipulates *q_RegData* using *iInstruction*. For example, the *MACC* module includes a *mad* unit called macc32 that performs the required calculations, replicated below:

```
//Floating point Multiply and Accumulate
   macc32 mad(OperandAMod, OperandBMod, OperandCMod,
VectorOpcode,MaccResult,sclk);
```

(*macc.v*, 24:25-25:2.)

114. The *macc32* module receives *OperandAMod*, *OperandBMod*, *OperandCMod* as operands which include the data maintained in the register block (*oRegData*), and the *VectorOpcode* which includes instructions. The *macc32* module is then operative to use *OperandAMod*, *OperandBMod*, *OperandCMod* and *VectorOpcode* to perform 1) floating point operations which are arithmetic operations, and 2) logical comparisons which are logical operations. *See e.g. macc32.mc.*

115. The `vector` modules, which include the `macc_gpr` modules, and the *MACC* modules within the `sp` modules are the structures that perform the vertex operations and the pixel operations.

116. Within the instruction sequencer, `SQ_SP_instruct` is generated in the `sq_ais_output` module, as shown using the R400 RTL code below:

```
module sq_ais_output
(
…
SQ_SP_instr,
…
output [20:0]  SQ_SP_instr;
…
reg [20:0]  SQ_SP_instr;
…
// --------------------------------
   // -- SP instruction, write_mask --
   // ------------------------------
   // - valid with instruction start

   always @(posedge clk)
    begin
      case (gpr_phase)
       `SQ_SRCB_PHASE: begin
         case (alu_phase)
       LO: begin
            SQ_SP_instr <= {3'b000, aiq0_instr[06:00],
aiq0_instr[55:48], aiq0_instr[58], aiq0_instr[101:99]};
            u0_SQ_SP_write_mask <= aiq0_valid_bits  [3:0];
u1_SQ_SP_write_mask <= aiq0_valid_bits  [7:4];
            u2_SQ_SP_write_mask <= aiq0_valid_bits [11:8];
u3_SQ_SP_write_mask <= aiq0_valid_bits [15:12];
          end
       HI: begin
            SQ_SP_instr <= {aiq1_instr[07:00], aiq1_instr[55:48],
aiq1_instr[58], aiq1_instr[101:99]};
            u0_SQ_SP_write_mask <= aiq1_valid_bits  [3:0];
u1_SQ_SP_write_mask <= aiq1_valid_bits  [7:4];
            u2_SQ_SP_write_mask <= aiq1_valid_bits [11:8];
u3_SQ_SP_write_mask <= aiq1_valid_bits [15:12];
          end
         endcase
```

```
        end
        `SQ_SRCC_PHASE: begin
          case (alu_phase)
         LO: begin
              SQ_SP_instr <= {aiq0_instr[15:08], aiq0_instr[47:40],
aiq0_instr[57], aiq0_instr[98:96]};
              u0_SQ_SP_write_mask <= aiq0_valid_bits [19:16];
u1_SQ_SP_write_mask <= aiq0_valid_bits [23:20];
              u2_SQ_SP_write_mask <= aiq0_valid_bits [27:24];
u3_SQ_SP_write_mask <= aiq0_valid_bits [31:28];
           end
         HI: begin
              SQ_SP_instr <= {aiq1_instr[15:08], aiq1_instr[47:40],
aiq1_instr[57], aiq1_instr[98:96]};
              u0_SQ_SP_write_mask <= aiq1_valid_bits [19:16];
u1_SQ_SP_write_mask <= aiq1_valid_bits [23:20];
              u2_SQ_SP_write_mask <= aiq1_valid_bits [27:24];
u3_SQ_SP_write_mask <= aiq1_valid_bits [31:28];
           end
          endcase
        end
        `SQ_FA_PHASE: begin
          case (alu_phase)
         LO: begin
              SQ_SP_instr <= {aiq0_instr[23:16], aiq0_instr[39:32],
aiq0_instr[56], aiq0_instr[95:93]};
              u0_SQ_SP_write_mask <= aiq0_valid_bits [35:32];
u1_SQ_SP_write_mask <= aiq0_valid_bits [39:36];
              u2_SQ_SP_write_mask <= aiq0_valid_bits [43:40];
u3_SQ_SP_write_mask <= aiq0_valid_bits [47:44];
           end
         HI: begin
              SQ_SP_instr <= {aiq1_instr[23:16], aiq1_instr[39:32],
aiq1_instr[56], aiq1_instr[95:93]};
              u0_SQ_SP_write_mask <= aiq1_valid_bits [35:32];
u1_SQ_SP_write_mask <= aiq1_valid_bits [39:36];
              u2_SQ_SP_write_mask <= aiq1_valid_bits [43:40];
u3_SQ_SP_write_mask <= aiq1_valid_bits [47:44];
           end
          endcase
        end
        `SQ_SRCA_PHASE: begin
          case (alu_phase)
         LO: begin
              SQ_SP_instr <= {aiq0_instr[23:16], aiq0_instr[25:24],
aiq0_instr[31:26], aiq0_instr[92:88]};
              u0_SQ_SP_write_mask <= aiq0_valid_bits [51:48];
u1_SQ_SP_write_mask <= aiq0_valid_bits [55:52];
              u2_SQ_SP_write_mask <= aiq0_valid_bits [59:56];
u3_SQ_SP_write_mask <= aiq0_valid_bits [63:60];
           end
         HI: begin
```

```
            SQ_SP_instr <= {aiq1_instr[23:16], aiq1_instr[25:24],
aiq1_instr[31:26], aiq1_instr[92:88]};
            u0_SQ_SP_write_mask <= aiq1_valid_bits [51:48];
u1_SQ_SP_write_mask <= aiq1_valid_bits [55:52];
            u2_SQ_SP_write_mask <= aiq1_valid_bits [59:56];
u3_SQ_SP_write_mask <= aiq1_valid_bits [63:60];
          end
         endcase
       end
      endcase
   end
```

(*sq_ais_output.v*, 2:7-8, 15, 7:1, 9:1, 16:9-19:13.)

117. These instructions come from the *sq_alu_instr_queue* module which is instantiated in *sq.v* as *u0_sq_alu_instr_queue*. (*sq.v*, 68:6-69:24.) The instructions pass through the instruction sequencer which is instantiated as *u0_sq_alu_instr_seq* (*sq.v*, 70:2-71:21) and is defined in sq_alu_instr_seq.v.

118. Further, signal *aif_thread_type_q*, replicated below, shows that the instructions can be for vertex or pixel operations.

```
aif_thread_type_q, // vector type (0: pixel, 1: vertex)
```

(*sq_alu_instr_queue.v*, 2:21.)

119. As such, the shader may perform one of vertex operations or pixel operations depending, in part, on *aif_thread_type_q*.

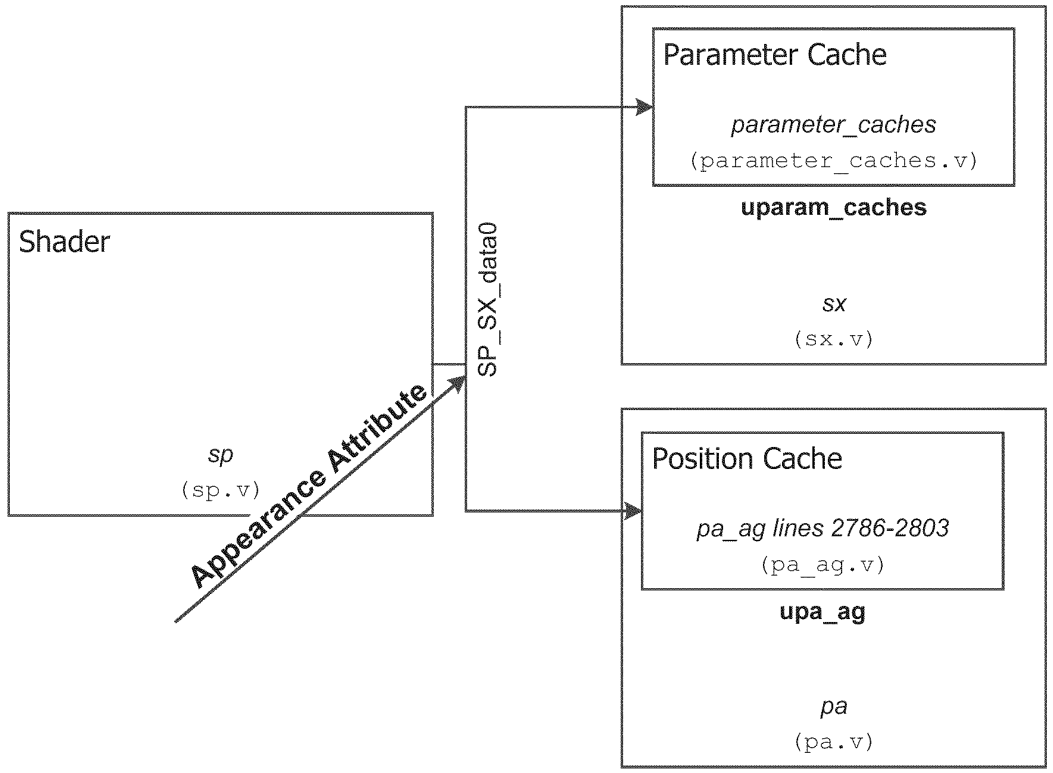
c. The shader also includes means for performing one of the vertex operations or pixel operations based on the selected one of the plurality of inputs.

120. The shader also includes the means for "*performing one of the vertex operations or pixel operations based on the selected one of the plurality of inputs.*"

121. As discussed in Section IX.A.2, the `sp` module receives 1) the `sq_sp_gpr_input_mux` signal which indicates to the *sp* module to perform vertex operations or pixel operations, and 2) the `SQ_SP_instruct` signal that provides the instructions of the selected operations. The `macc_gpr` module then performs the selected operation as discussed in Section IX.A.2, and is the means for performing the vertex operations or the pixel operations.

d. And the shader provides a appearance attribute.

122. And "*the shader provides a appearance attribute.*" Based on my understanding of the R400 RTL code, I have generated a figure below which represents my understanding of the components, and describe the code with reference to the figure.



123. Once the `vector` units complete processing, the `vector` units generate data called `sp_sx_data` (*see* `vector.v`, 4:5) and provide `sp_sx_data` to a shader export block, called the `sx` module (specified in `sx.v`).

124. For example, each `MACC` module generates a vector result, called `VectorResult0`, `VectorResult1`, `VectorResult2`, and `VectorResult3` respectively. (`vector.v`, 14:7, 14:25, 15:10, and 15:26.) In the `MACC` module the vector result is called `oResult`. (`macc.v`, 1:14, 3:6, and 29:1.) Signal `oResult` is generated based on the `iInstruction`, `iRegData`, and `iScalarData` that are inputs to the `MACC` module. (*Id.* at 1:16, 1:25, 2:50.)

The *MACC* module parses the `iInstruction` and assigns different bits of `iInstruction` to color (red, green, blue) and alpha (transparency) signals, as shown using R400 RTL code below:

```
  //-----------------------------------------------------------------
-----------
   //----------------------------------------------------------------
-----------
   //Registering the Instruction word (20 bits) in four
consecutive cycles
   //----------------------------------------------------------------
----------
   always@(posedge sclk)
     if(srst)
       q_Instruction0 <= 21'b0;
     else if(decode_SrcA)
       q_Instruction0 <= iInstruction;

   always@(posedge sclk)
     if(srst)
       q_Instruction1 <= 21'b0;
     else if(decode_SrcB)
       q_Instruction1 <= iInstruction;

    always@(posedge sclk)
     if(srst)
       q_Instruction2 <= 21'b0;
     else if(decode_SrcC)
       q_Instruction2 <= iInstruction;

   always@(posedge sclk)
     if(srst)
       q_Instruction3 <= 21'b0;
     else if(decode_Opcode)
       q_Instruction3 <= iInstruction;

   //grabing the export destination ID.
   //If we are dealing with an export instruction...this value
identifies which
   //attribute is being exported ...please refer to the shader
pipe spec for more details
   //on this
   //---------------------------------------------
   assign oExportDst = q_Instruction0[17:12];

   //----------------------------------------------------------------
------------
```

```
   //decoding the instruction word into a set of select/modify
signals used
   //for argument selection and input modification on the way to
MACC unit
   //---------------------------------------------------------------
-----------

   assign SrcASel = q_Instruction0[2:0];
   assign SrcANegate = q_Instruction0[3:3];
   assign SrcAAlphaSwizzle = q_Instruction0[11:10];
   assign SrcARedSwizzle = q_Instruction0[5:4];
   assign SrcAGreenSwizzle = q_Instruction0[7:6];
   assign SrcABlueSwizzle = q_Instruction0[9:8];

   assign SrcBSel = q_Instruction1[2:0];
   assign SrcBNegate = q_Instruction1[3:3];
   assign SrcBAlphaSwizzle = q_Instruction1[11:10];
   assign SrcBRedSwizzle = q_Instruction1[5:4];
   assign SrcBGreenSwizzle = q_Instruction1[7:6];
   assign SrcBBlueSwizzle = q_Instruction1[9:8];


   assign SrcCSel = q_Instruction2[2:0];
   assign SrcCNegate = q_Instruction2[3:3];
   assign SrcCAlphaSwizzle = q_Instruction2[11:10];
   assign SrcCRedSwizzle = q_Instruction2[5:4];
   assign SrcCGreenSwizzle = q_Instruction2[7:6];
   assign SrcCBlueSwizzle = q_Instruction2[9:8];



   assign VectorOpcode = q_Instruction3[4:0];
   assign ScalarOpcode = q_Instruction3[10:5];
   assign VectorClamp = q_Instruction3[11:11];
   assign ScalarClamp = q_Instruction3[12:12];
   assign VectorWriteMask = q_Instruction3[16:13];
   assign ScalarWriteMask = q_Instruction3[20:17];
```

(*Id.* at 10:12-13:6.)

125. The `MACC` module then uses these color and alpha signals to manipulate the `iRegData` as shown using the RTL code below:

```
//------------------------------------------------------------------
----------------
   //Argument Selectin for the three source operands going into
the MACC unit
```

```
   //All information required for the selection logic in embedded
into the ALU
   //Instrution Word. Please refer to the Shade Processor Spec
for a delailed
   //definition of the select fields for the three sources
   //------------------------------------------------------------
-------------------

   always@(SrcASel or iConstantData or iRegData or VectorData or
iScalarData)
     begin
      case(SrcASel)
        3'b000 : InputDataA = iConstantData;
        3'b100 : InputDataA = iRegData;
        3'b101 : InputDataA = iRegData;
        3'b110 : InputDataA = VectorData;
        3'b111 : InputDataA = iScalarData;
        default: InputDataA = iRegData;
      endcase // case(SrcASel)
     end // always@ (SrcASel or iConstantData or iRegData or
iVectorData or iScalarData)

    always@(SrcBSel or iConstantData or iRegData or VectorData or
iScalarData)
     begin
      case(SrcBSel)
        3'b000 : InputDataB = iConstantData;
        3'b100 : InputDataB = iRegData;
        3'b101 : InputDataB = iRegData;
        3'b110 : InputDataB = VectorData;
        3'b111 : InputDataB = iScalarData;
        default: InputDataB = iRegData;
      endcase // case(SrcBSel)
     end // always@ (SrcBSel or iConstantData or iRegData or
iVectorData or iScalarData)

    always@(SrcCSel or iConstantData or iRegData or VectorData or
iScalarData)
     begin
      case(SrcCSel)
        3'b000 : InputDataC = iConstantData;
        3'b100 : InputDataC = iRegData;
        3'b101 : InputDataC = iRegData;
        3'b110 : InputDataC = VectorData;
        3'b111 : InputDataC = iScalarData;
        default: InputDataC = iRegData;
      endcase // case(SrcCSel)
     end // always@ (SrcCSel or iConstantData or iRegData or
iVectorData or iScalarData)
   //------------------------------------------------------------
----------------------------
```

```
   //------------------------------------------------------------------
--------------------------
   //Input Modifiers ie. swizzle and negate are begin applied
   //------------------------------------------------------------------
-------------------------

   //Source A swizzling

   always@(InputDataA or SrcAAlphaSwizzle)
     case(SrcAAlphaSwizzle)
       2'b00: SrcAAlphaBus = InputDataA[127:96];
       2'b01: SrcAAlphaBus = InputDataA[95:64];
       2'b10: SrcAAlphaBus = InputDataA[63:32];
       2'b11: SrcAAlphaBus = InputDataA[31:0];
     endcase // case(SrcAAlphaSwizzle)


   always@(InputDataA or SrcARedSwizzle)
     case(SrcARedSwizzle)
       2'b00: SrcARedBus = InputDataA[95:64];
       2'b01: SrcARedBus = InputDataA[63:32];
       2'b10: SrcARedBus = InputDataA[31:0];
       2'b11: SrcARedBus = InputDataA[127:96];
     endcase // case(SrcARedSwizzle)



   always@(InputDataA or SrcAGreenSwizzle)
     case(SrcAGreenSwizzle)
       2'b00: SrcAGreenBus = InputDataA[63:32];
       2'b01: SrcAGreenBus = InputDataA[31:0];
       2'b10: SrcAGreenBus = InputDataA[127:96];
       2'b11: SrcAGreenBus = InputDataA[95:64];
     endcase // case(SrcAGreenSwizzle)

   always@(InputDataA or SrcABlueSwizzle)
     case(SrcABlueSwizzle)
       2'b00: SrcABlueBus = InputDataA[31:0];
       2'b01: SrcABlueBus = InputDataA[127:96];
       2'b10: SrcABlueBus = InputDataA[95:64];
       2'b11: SrcABlueBus = InputDataA[63:32];
     endcase // case(SrcAGreenSwizzle)

   //Source B swizzling

   always@(InputDataB or SrcBAlphaSwizzle)
     case(SrcBAlphaSwizzle)
       2'b00: SrcBAlphaBus = InputDataB[127:96];
       2'b01: SrcBAlphaBus = InputDataB[95:64];
       2'b10: SrcBAlphaBus = InputDataB[63:32];
       2'b11: SrcBAlphaBus = InputDataB[31:0];
```

```
    endcase

  always@(InputDataB or SrcBRedSwizzle)
    case(SrcBRedSwizzle)
      2'b00: SrcBRedBus = InputDataB[95:64];
      2'b01: SrcBRedBus = InputDataB[63:32];
      2'b10: SrcBRedBus = InputDataB[31:0];
      2'b11: SrcBRedBus = InputDataB[127:96];
    endcase // case(SrcBRedSwizzle)

  always@(InputDataB or SrcBGreenSwizzle)
    case(SrcBGreenSwizzle)
      2'b00: SrcBGreenBus = InputDataB[63:32];
      2'b01: SrcBGreenBus = InputDataB[31:0];
      2'b10: SrcBGreenBus = InputDataB[127:96];
      2'b11: SrcBGreenBus = InputDataB[95:64];
    endcase // case(SrcBGreenSwizzle)


  always@(InputDataB or SrcBBlueSwizzle)
    case(SrcBBlueSwizzle)
      2'b00: SrcBBlueBus = InputDataB[31:0];
      2'b01: SrcBBlueBus = InputDataB[127:96];
      2'b10: SrcBBlueBus = InputDataB[95:64];
      2'b11: SrcBBlueBus = InputDataB[63:32];
    endcase // case(SrcBGreenSwizzle)

  //Source C swizzling
    always@(InputDataC or SrcCAlphaSwizzle)
    case(SrcCAlphaSwizzle)
      2'b00: SrcCAlphaBus = InputDataC[127:96];
      2'b01: SrcCAlphaBus = InputDataC[95:64];
      2'b10: SrcCAlphaBus = InputDataC[63:32];
      2'b11: SrcCAlphaBus = InputDataC[31:0];
    endcase

  always@(InputDataC or SrcCRedSwizzle)
    case(SrcCRedSwizzle)
      2'b00: SrcCRedBus = InputDataC[95:64];
      2'b01: SrcCRedBus = InputDataC[63:32];
      2'b10: SrcCRedBus = InputDataC[31:0];
      2'b11: SrcCRedBus = InputDataC[127:96];
    endcase // case(SrcCRedSwizzle)

  always@(InputDataC or SrcCGreenSwizzle)
    case(SrcCGreenSwizzle)
      2'b00: SrcCGreenBus = InputDataC[63:32];
      2'b01: SrcCGreenBus = InputDataC[31:0];
      2'b10: SrcCGreenBus = InputDataC[127:96];
      2'b11: SrcCGreenBus = InputDataC[95:64];
    endcase // case(SrcCGreenSwizzle)
```

```
   always@(InputDataC or SrcCBlueSwizzle)
     case(SrcCBlueSwizzle)
       2'b00: SrcCBlueBus = InputDataC[31:0];
       2'b01: SrcCBlueBus = InputDataC[127:96];
       2'b10: SrcCBlueBus = InputDataC[95:64];
       2'b11: SrcCBlueBus = InputDataC[63:32];
     endcase // case(SrcCGreenSwizzle)



   //----------------------------------------------------------------
-------------
   //Modeling stages  for the Argument storing
   //----------------------------------------------------------------
-----------

   // always@(SrcAAlphaBus or decode_SrcA)
   //  if(decode_SrcA)
   //    SrcAAlphaBusLatch1 = SrcAAlphaBus;

   always@(posedge sclk)
     if(decode_SrcB)
     begin
        SrcAAlphaBusLatch1 <= SrcAAlphaBus;
           SrcARedBusLatch0 <= SrcARedBus;
        SrcAGreenBusLatch0 <= SrcAGreenBus;
        SrcABlueBusLatch0 <= SrcABlueBus;
     end

   always@(posedge sclk)
     if(decode_SrcC)
       begin
        SrcBAlphaBusLatch1 <= SrcBAlphaBus;
        SrcBRedBusLatch1 <= SrcBRedBus;
        SrcBGreenBusLatch0 <= SrcBGreenBus;
        SrcBBlueBusLatch0 <= SrcBBlueBus;
       end


   always@(posedge sclk)
     if(decode_Opcode)
       begin
        SrcCAlphaBusLatch1 <= SrcCAlphaBus;
        SrcCRedBusLatch1 <= SrcCRedBus;
        SrcCGreenBusLatch1 <= SrcCGreenBus;
        SrcCBlueBusLatch0 <= SrcCBlueBus;
       end

   //second level of latches
   always@(posedge sclk)
     if(decode_SrcA)
```

```
        begin
         SrcARedBusLatch1 <= SrcARedBusLatch0;
         SrcAGreenBusLatch1 <= SrcAGreenBusLatch0;
         SrcABlueBusLatch1 <= SrcABlueBusLatch0;
         SrcBGreenBusLatch1 <= SrcBGreenBusLatch0;
         SrcBBlueBusLatch1 <= SrcBBlueBusLatch0;
         SrcCBlueBusLatch1 <= SrcCBlueBusLatch0;
        end

    //-------------------------------------------------------------
------------------------
    // register the outputs from the latches into the MACC unit
    //-------------------------------------------------------------
-----------------------
    always@(posedge sclk)
      begin
       if(decode_SrcA)
         OperandA <= SrcAAlphaBusLatch1;
       else if(decode_SrcB)
         OperandA <= SrcARedBusLatch1;
       else if(decode_SrcC)
         OperandA <= SrcAGreenBusLatch1;
       else
         OperandA <= SrcABlueBusLatch1;
      end // always@ (sclk)


    always@(posedge sclk)
      begin
       if(decode_SrcA)
         OperandB <= SrcBAlphaBusLatch1;
       else if(decode_SrcB)
         OperandB <= SrcBRedBusLatch1;
       else if(decode_SrcC)
         OperandB <= SrcBGreenBusLatch1;
       else
         OperandB <= SrcBBlueBusLatch1;
      end // always@ (sclk)

    always@(posedge sclk)
      begin
       if(decode_SrcA)
         OperandC <= SrcCAlphaBusLatch1;
       else if(decode_SrcB)
         OperandC <= SrcCRedBusLatch1;
       else if(decode_SrcC)
         OperandC <= SrcCGreenBusLatch1;
       else
         OperandC <= SrcCBlueBusLatch1;
      end // always@ (sclk)

    //----------------------------------------------------------
```

```
   //Input Modifier ....NEGATE.
   //-----------------------------------------------------------
   always@(SrcANegate or OperandA)
     if(SrcANegate)
       OperandAMod[31:0]=
{OperandA[31]^SrcANegate,OperandA[30:0]};
     else
       OperandAMod = OperandA;

   always@(SrcBNegate or OperandB)
     if(SrcBNegate)
       OperandBMod[31:0]=
{OperandB[31]^SrcBNegate,OperandB[30:0]};
     else
       OperandBMod = OperandB;

   always@(SrcCNegate or OperandC)
     if(SrcCNegate)
       OperandCMod[31:0]=
{OperandC[31]^SrcCNegate,OperandC[30:0]};
     else
       OperandCMod = OperandC;
```

(*Id.* at 13:8-23:22.)

126. Then, the `MACC` module generates `oResult` which includes, for example, a color or alpha attribute as a results of instructions in the color and alpha parameters, using the R400 RTL code below:

```
//Floating point Multiply and Accumulate
   macc32 mad(OperandAMod, OperandBMod, OperandCMod,
VectorOpcode,MaccResult,sclk);


   //---------------------------------------------------------------
-----------------------
   //some of the opcodes do not have to be implemented via the
MACC unit
   //for example : MAX can be implemented via compares of the
exponents and/or mantissas of
   //the two numbers assuming that the numbers are normalized
   //this is a separate parallel pipeline from the MACC
   //---------------------------------------------------------------
---------------------------

   //MIN or MAX
   //revisit this logic for the case when exp = 0 ...ANDI
   always @(/*AUTOSENSE*/OperandAMod or OperandBMod)
     begin
      if(OperandAMod[30:0] >= OperandBMod[30:0])
        begin
           if(!OperandAMod[31])
             begin
              ResultMax = OperandAMod;
              ResultMin = OperandBMod;
             end
           else
             begin
              ResultMax = OperandBMod;
              ResultMin = OperandAMod;
             end
        end // if (OperandAMod[30:0] >= OperandBMod[30:0])
      else if (OperandBMod[30:0] >= OperandAMod[30:0])
        begin
           if(!OperandBMod[31])
             begin
              ResultMax = OperandBMod;
              ResultMin = OperandAMod;
             end
           else
             begin
              ResultMax = OperandAMod;
              ResultMin = OperandBMod;
             end
        end // if (OperandBMod[30:0] >= OperandAMod[30:0])
     end // always @ (...


   //choose  MIN vs. MAX
   assign ResultMaxMin = (opcode_mux_ctl[1]) ? ResultMax :
ResultMin;

   //delay the ResultMaxMin to match with the other path of the
pipeline that goes through the MACC
   always@(posedge sclk)
     begin
      q0_ResultMaxMin <= ResultMaxMin;
      q1_ResultMaxMin <= q0_ResultMaxMin;
      q2_ResultMaxMin <= q1_ResultMaxMin;
      q3_ResultMaxMin <= q2_ResultMaxMin;
     end

   reg [31:0] MaccResultMux;

   //---------------------------------------------------------------
-------------------------------------
   //Routing the Result into MaccResultMux based on the opcode
   //------------------------------------------------------------------
------------------------------------
   always @(/*AUTOSENSE*/MaccResult or q3_ResultMaxMin
         or q4_opcode_mux_ctl)
     begin
      case(q4_opcode_mux_ctl)
        2'b00: MaccResultMux = MaccResult;
        2'b01: MaccResultMux = q3_ResultMaxMin;
        2'b10: MaccResultMux = q3_ResultMaxMin;
        default : MaccResultMux = MaccResult;
      endcase // case(opcode_mux_ctl)
     end

   //------------------------------------------------------------------
-------------------------------------
   //Clamping the result and other output modifiers
   always@(/*AUTOSENSE*/MaccResultMux or ResultClamp)
     begin
      if(ResultClamp)
        begin
           if(MaccResultMux[31])
             MaccResultClamp = ZERO;
           else if(MaccResultMux[30:24] > GT_ONE_EXP)
               MaccResultClamp = ONE;
           else
             MaccResultClamp = MaccResultMux;
        end
      else
        MaccResultClamp = MaccResultMux;
     end // always@ (MaccResult or ResultClamp)

   //------------------------------------------------------------------
-------------------------------------
   //pipeline delays for the code....creating the 4 stage delay
for the VectorResult
   //------------------------------------------------------------------
-------------------------------------
   always@(posedge sclk)
     begin
      q0_MaccResultClamp <= MaccResultClamp;
      q1_MaccResultClamp <= q0_MaccResultClamp;
      q2_MaccResultClamp <= q1_MaccResultClamp;
     end

   assign VectorData = {q2_MaccResultClamp, q1_MaccResultClamp,
q0_MaccResultClamp, MaccResultClamp};
   assign oResult = {q2_MaccResultClamp, q1_MaccResultClamp,
q0_MaccResultClamp, MaccResultClamp} ;
```

(*Id.* at 24:25-29:2.)

127. Each one of *MACC* modules passes the *oResult* output to the respective *VectorResult0*, *VectorResult1*, *VectorResult2*, *and* *VectorResult3* signals, which include an appearance attribute. The shader thus generates an appearance attribute.

128. The vector result signals (*VectorResult0*, *VectorResult1*, *VectorResult2*, and *VectorResult3*) are assigned to *sp_sx_data* using the R400 RTL code below:

```
//--------------------------------------------------------------------
--------------------------------------------------------------------
------
   //Muxing the gpr vector results into one final vector result
conrolled by the phase_mux signal or a registered version of it
   //-----------------------------------------------------------------
--------------------------------------------------------------------
----------

   always @(/*AUTOSENSE*/VectorResult0 or VectorResult1
          or VectorResult2 or VectorResult3 or
sq_sp_gpr_phase_mux)
     begin
      case(sq_sp_gpr_phase_mux)
        2'b00: osp_sx_data = VectorResult0;
        2'b01: osp_sx_data = VectorResult1;
        2'b10: osp_sx_data = VectorResult2;
        2'b11: osp_sx_data = VectorResult3;
      endcase // case(sq_sp_gpr_phase_mux)
     end

   assign sp_sx_data = osp_sx_data;
```

(*vector.v*, 16:8-16:26.)

129. Additionally, the *vector* unit also sets the *sp_sx_exporting* and *sp_sx_exp_pvalid* signals which indicate that the shader in the *sp*

module will export `sp_sx_data` and that the data is valid. (`vector.v`, 19:23-20:22, 22:4, 22:7.)

130. The `vector` unit then passes the `sp_sx_data`, `sp_sx_exporting`, and `sp_sx_exp_pvalid` signals to the `sp` module. (*Id.* at 1:16-17.) The `sp` module assigns `sp_sx_data` from `uvector0` to `osp_sx_data0`, `sp_sx_data` from `uvector1` to `osp_sx_data1`, `sp_sx_data` from `uvector2` to `osp_sx_data2`, and `sp_sx_data` from `uvector3` to `osp_sx_data3`. (`sp.v`, 15:11, 16:16, 17:7, 17:26.) The `sp` module interface also provides `sp_sx_exporting` to `sp_exporting`, and `sp_sx_exp_pvalid` to `sp_exp_pvalid`. (*Id.* at 15:12, 15:15.)

131. The `sp` module then assigns the `osp_sx_data0`, `osp_sx_data1`, `osp_sx_data2`, and `osp_sx_data3` signals and the `sp_exporting`, and `sp_sx_exp_pvalid` signals to the SP_SX interface, using the R400 RTL code below:

```
 //-----------------------------------------------------------------
------------------------
   //SHADER(SP) - SX(SHADER EXPORT)
   //This interface represents pixel/parameter data being
exported out of the shader pipe
   //into the SX block
   //---------------------------------------------------------------
-------------------------
   output [127:0] SP_SX_data0,
SP_SX_data1,SP_SX_data2,SP_SX_data3;
```

```
   wire [127:0]   q_sp_sx_data0, q_sp_sx_data1 , q_sp_sx_data2 ,
q_sp_sx_data3;
   wire [127:0]   osp_sx_data0, osp_sx_data1 , osp_sx_data2 ,
osp_sx_data3;

   ati_dff_out #(128) usp_sx_data0(sclk,
osp_sx_data0,q_sp_sx_data0);
   ati_dff_out #(128) usp_sx_data1(sclk,
osp_sx_data1,q_sp_sx_data1);
   ati_dff_out #(128) usp_sx_data2(sclk,
osp_sx_data2,q_sp_sx_data2);
   ati_dff_out #(128) usp_sx_data3(sclk,
osp_sx_data3,q_sp_sx_data3);

   //export data going out to SX (shader export)
   assign    SP_SX_data0 = q_sp_sx_data0;
   assign    SP_SX_data1 = q_sp_sx_data1;
   assign    SP_SX_data2 = q_sp_sx_data2;
   assign    SP_SX_data3 = q_sp_sx_data3;
```

(*Id.* at 3:18-4:10.)

```
   output [3:0]   SP_SX_exp_pvalid;
   output [0:0]   SP_SX_exporting ;
   …
   wire [3:0]      sp_exp_pvalid;
   wire [0:0]      sp_exporting ;
   …
   wire [3:0]      q_sp_exp_pvalid;
   wire [0:0]      q_sp_exporting ;
   …
   ati_dff_out #(4)
usp_exp_pvalid(sclk,sp_exp_pvalid,q_sp_exp_pvalid);
   ati_dff_out #(1)
usp_exporting(sclk,sp_exporting,q_sp_exporting);
   …
   assign    SP_SX_exp_pvalid = q_sp_exp_pvalid;
   assign    SP_SX_exporting = q_sp_exporting ;
```

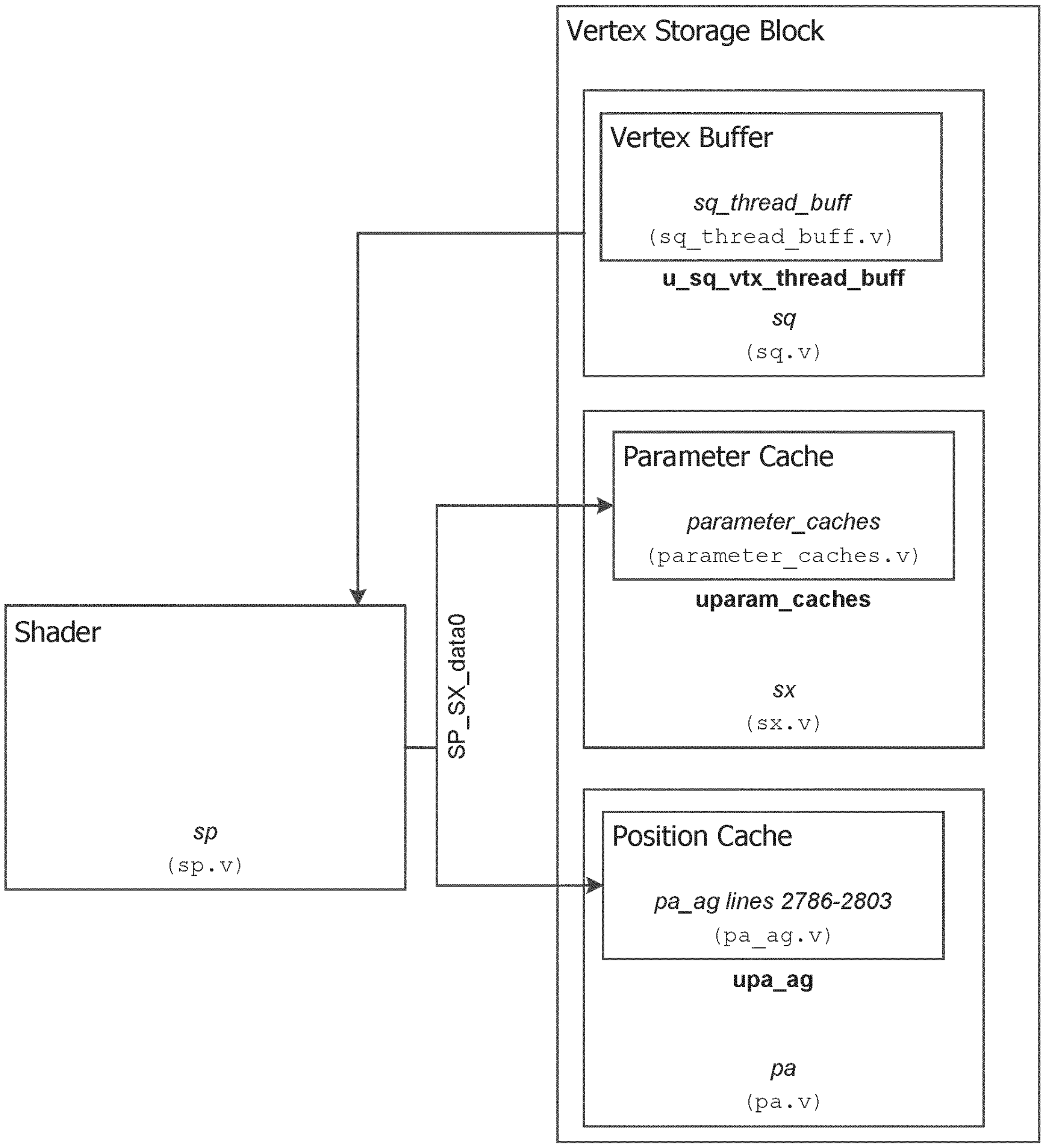
(*Id.* at 7:14-15, 19-20, 24-25, 8:4-5, 9-10.)

132. When the `SP_SX_exporting` and `SP_SX_exp_pvalid` parameters indicate that the `SP_SX_data0-SP_SX_data3` data is valid, the `sp` module exports the data to the shader export block (the *sx* module). Because

the `sp` module exports the `SP_SX_data0-SP_SX_data3` generated by the shader, the shader provides an appearance attribute.

B. Claim 2

133. Claim 2 recites the graphics processor of claim 1, "*further including a vertex storage block for maintaining vertex information.*" Based on my understanding of the R400 RTL code, I have generated a figure below which represents my understanding of the components, and describe the code with reference to the figure.

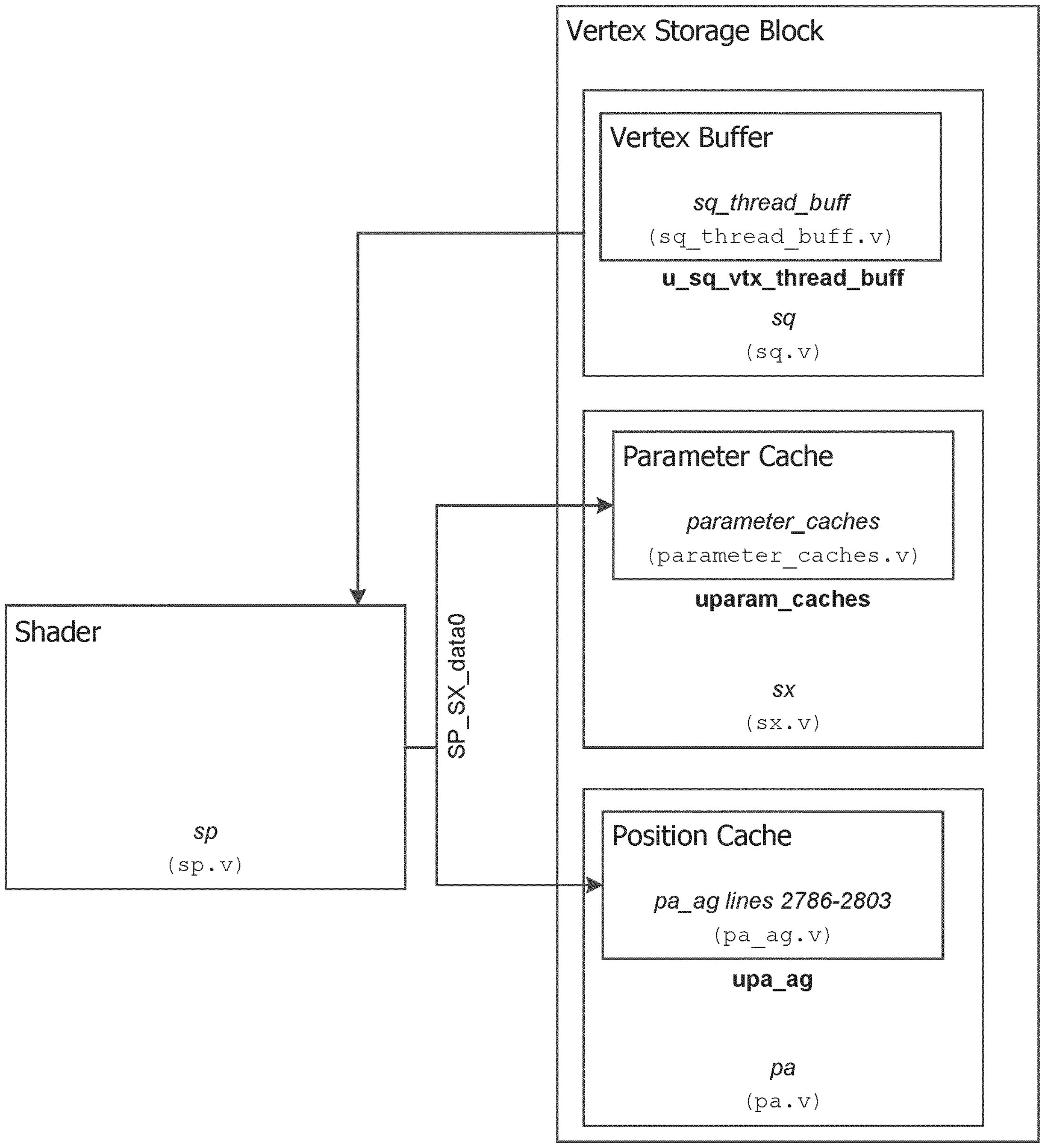


134. The vertex storage block includes a vertex thread buffer called *u_sq_vtx_thread_buff* (*sq.v*, 34:19-38:18), a parameter cache called *uparam_caches* (*sx.v*, 20:13-21:3), and a position cache

u_pos_dum_mem_p2 (*pa_ag.v*, 112:21-2113:5; also shown as lines 2786-2803 in my figure above).

C. Claim 3

135. Claim 3 recites a vertex storage block that further includes a parameter cache and a position cache. I have already identified where the vertex storage block includes the parameter cache and the position cache in my analysis of claim 2 in Section IX.B.



1. The vertex storage block further includes a parameter cache

136. Claim 3 recites a graphics processor of claim 2, "vertex storage block further includes a parameter cache operative to maintain appearance attribute data for a corresponding vertex." As discussed in Section IX.C, the vertex storage block

includes a parameter cache called uparam_caches. (*sx.v*, 20:13-21:3.) Module *uparam_caches* is an instance of a *parameter_caches* module and is defined in *parameter_caches.v*.

137. As I also described in Section IX.A.3, the shader provides appearance data in *SP_SX_data0-SP_SX_data3* to the shader export block included in the *sx* module. The *sx* module receives the *SP_SX_data0- SP_SX_data3* using the R400 RTL code replicated below:

```
module sx(/*AUTOARG*/
…
SQ_SX_pc_channel_mask, SP0_SX_data0, SP0_SX_data1, SP0_SX_data2,
   SP0_SX_data3, SP1_SX_data0, SP1_SX_data1, SP1_SX_data2,
   SP1_SX_data3, SP0_SX_exp_pvalid, SP1_SX_exp_pvalid,
…
input [127:0] P0_SX_data0,SP0_SX_data1,SP0_SX_data2,SP0_SX_data3;
input [127:0] P1_SX_data0,SP1_SX_data1,SP1_SX_data2,SP1_SX_data3;
```

(*sx.v*, 1:12, 2:8-10, 5:10-11.)

138. The the R400 RTL code above demonstrates that the shader export module receives *SP_SX_data0-SP_SX_data3* from two shaders, as *SP0_SX_data0- SP0_SX_data3* and *SP1_SX_data0-SP1_SX_data3*. For simplicity, I analyze the components here with respect to *SP_SX_data0-SP_SX_data3* as that is sufficient to meet the claim limitations.

139. The signals *SP_SX_data0-SP_SX_data3* include appearance attribute data for a corresponding vertex that was processed by the shader.

140. The shader export circuit passes *SP0_SX_data0* - *SP0_SX_data3* to *uparam_caches*. (*sx.v*, 20:24-26.) Module *uparam_caches* includes eight *uparam_cache_ctl1* - *uparam_cache_ctl7* instances of the *uparam_cache_ctl* module. The *uparam_cache_ctl* module is defined in *parameter_cache_ctl.v*.

141. The first four instances maintain *SP_SX_data0-SP_SX_data3*. (*parameter_caches.v*, 5:4-8:12.)

142. The *uparam_cache_ctl* module maintains *SP_SX_data0-SP_SX_data3* which the *uparam_cache_ctl* module receives as *SP_SX_data*. (*param_cache_ctl.v*, 1:19-20.) The *uparam_cache_ctl* module then reads and writes (maintains) *SP_SX* data in the memory module type "*rfsd2_128x128cm2sw0*" called "*u_pc*" as shown in using the R400 RTL code below:

```
always @(posedge sclk)
     begin
      if(srst)
        begin
           pc_ptr0 <= 11'b0;
           pc_ptr1 <= 11'b0;
           pc_ptr2 <= 11'b0;
           pc_wr_en <= 1'b0;
           pc_wr_addr <= 7'b0;
           pc_cmask <= 4'b0;
           vertex_data_in <= 127'b0;
           q0_vertex_data_out <= 127'b0;
        end
      else
```

```
        begin
           pc_ptr0 <= SQ_SX_ptr0;
           pc_ptr1 <= SQ_SX_ptr1;
           pc_ptr2 <= SQ_SX_ptr2;
           pc_wr_en <= SQ_SX_pc_wr_en;
           pc_wr_addr <= SQ_SX_pc_wr_addr;
           pc_cmask <=  SQ_SX_pc_cmask;
           vertex_data_in <= SP_SX_data;
           q0_vertex_data_out <= vertex_data_out;
        end
     end // always @ (posedge sclk)
…
rfsd2_128x128cm2sw0 u_pc
     //   tp_coord_fifo_ram utp_coord_fifo_ram_0
     //     (/*VRGIO tp_coord_fifo_ram cfifo_in cfifo_out
q_cfifo_wptr q_cfifo_rptr cfifo_ram_wen cfifo_ram_ren I0*/
     //      );
     (/*VRGIO rfsd2_128x128cm2sw0 vertex_data_in vertex_data_out
pc_wr_addr pc_index pc_wr_en pc_rd_en null*/
      // READ INTERFACE
      .CLKB(iSCLK), // Read Clock
      .OEB(pc_rd_en), // Output enable
      .MEB(vdd), // Read enable
      .ADRB0(pc_index[0]), .ADRB1(pc_index[1]),
.ADRB2(pc_index[2]), .ADRB3(pc_index[3]),  // Read Address
      .ADRB4(pc_index[4]), .ADRB5(pc_index[5]),
.ADRB6(pc_index[6]),  // Read Address
      .QB0(vertex_data_out[0]), .QB1(vertex_data_out[1]),
.QB2(vertex_data_out[2]), .QB3(vertex_data_out[3]),  // Read Data
      .QB4(vertex_data_out[4]), .QB5(vertex_data_out[5]),
.QB6(vertex_data_out[6]), .QB7(vertex_data_out[7]),  // Read Data
      .QB8(vertex_data_out[8]), .QB9(vertex_data_out[9]),
.QB10(vertex_data_out[10]), .QB11(vertex_data_out[11]),  // Read
Data
      .QB12(vertex_data_out[12]), .QB13(vertex_data_out[13]),
.QB14(vertex_data_out[14]), .QB15(vertex_data_out[15]),  // Read
Data
      .QB16(vertex_data_out[16]), .QB17(vertex_data_out[17]),
.QB18(vertex_data_out[18]), .QB19(vertex_data_out[19]),  // Read
Data
      .QB20(vertex_data_out[20]), .QB21(vertex_data_out[21]),
.QB22(vertex_data_out[22]), .QB23(vertex_data_out[23]),  // Read
Data
      .QB24(vertex_data_out[24]), .QB25(vertex_data_out[25]),
.QB26(vertex_data_out[26]), .QB27(vertex_data_out[27]),  // Read
Data
      .QB28(vertex_data_out[28]), .QB29(vertex_data_out[29]),
.QB30(vertex_data_out[30]), .QB31(vertex_data_out[31]),  // Read
Data
      .QB32(vertex_data_out[32]), .QB33(vertex_data_out[33]),
.QB34(vertex_data_out[34]), .QB35(vertex_data_out[35]),  // Read
Data
```

```
	.QB36(vertex_data_out[36]), .QB37(vertex_data_out[37]),
.QB38(vertex_data_out[38]), .QB39(vertex_data_out[39]),  // Read
Data
	.QB40(vertex_data_out[40]), .QB41(vertex_data_out[41]),
.QB42(vertex_data_out[42]), .QB43(vertex_data_out[43]),  // Read
Data
	.QB44(vertex_data_out[44]), .QB45(vertex_data_out[45]),
.QB46(vertex_data_out[46]), .QB47(vertex_data_out[47]),  // Read
Data
	.QB48(vertex_data_out[48]), .QB49(vertex_data_out[49]),
.QB50(vertex_data_out[50]), .QB51(vertex_data_out[51]),  // Read
Data
	.QB52(vertex_data_out[52]), .QB53(vertex_data_out[53]),
.QB54(vertex_data_out[54]), .QB55(vertex_data_out[55]),  // Read
Data
	.QB56(vertex_data_out[56]), .QB57(vertex_data_out[57]),
.QB58(vertex_data_out[58]), .QB59(vertex_data_out[59]),  // Read
Data
	.QB60(vertex_data_out[60]), .QB61(vertex_data_out[61]),
.QB62(vertex_data_out[62]), .QB63(vertex_data_out[63]),  // Read
Data
	.QB64(vertex_data_out[64]), .QB65(vertex_data_out[65]),
.QB66(vertex_data_out[66]), .QB67(vertex_data_out[67]),  // Read
Data
	.QB68(vertex_data_out[68]), .QB69(vertex_data_out[69]),
.QB70(vertex_data_out[70]), .QB71(vertex_data_out[71]),  // Read
Data
	.QB72(vertex_data_out[72]), .QB73(vertex_data_out[73]),
.QB74(vertex_data_out[74]), .QB75(vertex_data_out[75]),  // Read
Data
	.QB76(vertex_data_out[76]), .QB77(vertex_data_out[77]),
.QB78(vertex_data_out[78]), .QB79(vertex_data_out[79]),  // Read
Data
	.QB80(vertex_data_out[80]), .QB81(vertex_data_out[81]),
.QB82(vertex_data_out[82]), .QB83(vertex_data_out[83]),  // Read
Data
	.QB84(vertex_data_out[84]), .QB85(vertex_data_out[85]),
.QB86(vertex_data_out[86]), .QB87(vertex_data_out[87]),  // Read
Data
	.QB88(vertex_data_out[88]), .QB89(vertex_data_out[89]),
.QB90(vertex_data_out[90]), .QB91(vertex_data_out[91]),  // Read
Data
	.QB92(vertex_data_out[92]), .QB93(vertex_data_out[93]),
.QB94(vertex_data_out[94]), .QB95(vertex_data_out[95]),  // Read
Data
	.QB96(vertex_data_out[96]), .QB97(vertex_data_out[97]),
.QB98(vertex_data_out[98]), .QB99(vertex_data_out[99]),  // Read
Data
	.QB100(vertex_data_out[100]), .QB101(vertex_data_out[101]),
.QB102(vertex_data_out[102]), .QB103(vertex_data_out[103]),  //
Read Data
```

```
      .QB104(vertex_data_out[104]), .QB105(vertex_data_out[105]),
.QB106(vertex_data_out[106]), .QB107(vertex_data_out[107]),  //
Read Data
      .QB108(vertex_data_out[108]), .QB109(vertex_data_out[109]),
.QB110(vertex_data_out[110]), .QB111(vertex_data_out[111]),  //
Read Data
      .QB112(vertex_data_out[112]), .QB113(vertex_data_out[113]),
.QB114(vertex_data_out[114]), .QB115(vertex_data_out[115]),  //
Read Data
      .QB116(vertex_data_out[116]), .QB117(vertex_data_out[117]),
.QB118(vertex_data_out[118]), .QB119(vertex_data_out[119]),  //
Read Data
      .QB120(vertex_data_out[120]), .QB121(vertex_data_out[121]),
.QB122(vertex_data_out[122]), .QB123(vertex_data_out[123]),  //
Read Data
      .QB124(vertex_data_out[124]), .QB125(vertex_data_out[125]),
.QB126(vertex_data_out[126]), .QB127(vertex_data_out[127]),  //
Read Data
      // WRITE INTERFACE
      .CLKA(iSCLK), // Write Clock
      .WEA(pc_wr_en), // Write enable
      .MEA(vdd), // Memory enable
      .ADRA0(pc_wr_addr[0]), .ADRA1(pc_wr_addr[1]),
.ADRA2(pc_wr_addr[2]), .ADRA3(pc_wr_addr[3]),  // Write Address
      .ADRA4(pc_wr_addr[4]), .ADRA5(pc_wr_addr[5]),
.ADRA6(pc_wr_addr[6]),  // Write Address
      .DA0(vertex_data_in[0]), .DA1(vertex_data_in[1]),
.DA2(vertex_data_in[2]), .DA3(vertex_data_in[3]),  // Write Data
      .DA4(vertex_data_in[4]), .DA5(vertex_data_in[5]),
.DA6(vertex_data_in[6]), .DA7(vertex_data_in[7]),  // Write Data
      .DA8(vertex_data_in[8]), .DA9(vertex_data_in[9]),
.DA10(vertex_data_in[10]), .DA11(vertex_data_in[11]),  // Write
Data
      .DA12(vertex_data_in[12]), .DA13(vertex_data_in[13]),
.DA14(vertex_data_in[14]), .DA15(vertex_data_in[15]),  // Write
Data
      .DA16(vertex_data_in[16]), .DA17(vertex_data_in[17]),
.DA18(vertex_data_in[18]), .DA19(vertex_data_in[19]),  // Write
Data
      .DA20(vertex_data_in[20]), .DA21(vertex_data_in[21]),
.DA22(vertex_data_in[22]), .DA23(vertex_data_in[23]),  // Write
Data
      .DA24(vertex_data_in[24]), .DA25(vertex_data_in[25]),
.DA26(vertex_data_in[26]), .DA27(vertex_data_in[27]),  // Write
Data
      .DA28(vertex_data_in[28]), .DA29(vertex_data_in[29]),
.DA30(vertex_data_in[30]), .DA31(vertex_data_in[31]),  // Write
Data
      .DA32(vertex_data_in[32]), .DA33(vertex_data_in[33]),
.DA34(vertex_data_in[34]), .DA35(vertex_data_in[35]),  // Write
Data
```

```
        .DA36(vertex_data_in[36]), .DA37(vertex_data_in[37]),
.DA38(vertex_data_in[38]), .DA39(vertex_data_in[39]),  // Write
Data
        .DA40(vertex_data_in[40]), .DA41(vertex_data_in[41]),
.DA42(vertex_data_in[42]), .DA43(vertex_data_in[43]),  // Write
Data
        .DA44(vertex_data_in[44]), .DA45(vertex_data_in[45]),
.DA46(vertex_data_in[46]), .DA47(vertex_data_in[47]),  // Write
Data
        .DA48(vertex_data_in[48]), .DA49(vertex_data_in[49]),
.DA50(vertex_data_in[50]), .DA51(vertex_data_in[51]),  // Write
Data
        .DA52(vertex_data_in[52]), .DA53(vertex_data_in[53]),
.DA54(vertex_data_in[54]), .DA55(vertex_data_in[55]),  // Write
Data
        .DA56(vertex_data_in[56]), .DA57(vertex_data_in[57]),
.DA58(vertex_data_in[58]), .DA59(vertex_data_in[59]),  // Write
Data
        .DA60(vertex_data_in[60]), .DA61(vertex_data_in[61]),
.DA62(vertex_data_in[62]), .DA63(vertex_data_in[63]),  // Write
Data
        .DA64(vertex_data_in[64]), .DA65(vertex_data_in[65]),
.DA66(vertex_data_in[66]), .DA67(vertex_data_in[67]),  // Write
Data
        .DA68(vertex_data_in[68]), .DA69(vertex_data_in[69]),
.DA70(vertex_data_in[70]), .DA71(vertex_data_in[71]),  // Write
Data
        .DA72(vertex_data_in[72]), .DA73(vertex_data_in[73]),
.DA74(vertex_data_in[74]), .DA75(vertex_data_in[75]),  // Write
Data
        .DA76(vertex_data_in[76]), .DA77(vertex_data_in[77]),
.DA78(vertex_data_in[78]), .DA79(vertex_data_in[79]),  // Write
Data
        .DA80(vertex_data_in[80]), .DA81(vertex_data_in[81]),
.DA82(vertex_data_in[82]), .DA83(vertex_data_in[83]),  // Write
Data
        .DA84(vertex_data_in[84]), .DA85(vertex_data_in[85]),
.DA86(vertex_data_in[86]), .DA87(vertex_data_in[87]),  // Write
Data
        .DA88(vertex_data_in[88]), .DA89(vertex_data_in[89]),
.DA90(vertex_data_in[90]), .DA91(vertex_data_in[91]),  // Write
Data
        .DA92(vertex_data_in[92]), .DA93(vertex_data_in[93]),
.DA94(vertex_data_in[94]), .DA95(vertex_data_in[95]),  // Write
Data
        .DA96(vertex_data_in[96]), .DA97(vertex_data_in[97]),
.DA98(vertex_data_in[98]), .DA99(vertex_data_in[99]),  // Write
Data
        .DA100(vertex_data_in[100]), .DA101(vertex_data_in[101]),
.DA102(vertex_data_in[102]), .DA103(vertex_data_in[103]),  //
Write Data
```

```
       .DA104(vertex_data_in[104]), .DA105(vertex_data_in[105]),
.DA106(vertex_data_in[106]), .DA107(vertex_data_in[107]),  //
Write Data
       .DA108(vertex_data_in[108]), .DA109(vertex_data_in[109]),
.DA110(vertex_data_in[110]), .DA111(vertex_data_in[111]),  //
Write Data
       .DA112(vertex_data_in[112]), .DA113(vertex_data_in[113]),
.DA114(vertex_data_in[114]), .DA115(vertex_data_in[115]),  //
Write Data
       .DA116(vertex_data_in[116]), .DA117(vertex_data_in[117]),
.DA118(vertex_data_in[118]), .DA119(vertex_data_in[119]),  //
Write Data
       .DA120(vertex_data_in[120]), .DA121(vertex_data_in[121]),
.DA122(vertex_data_in[122]), .DA123(vertex_data_in[123]),  //
Write Data
       .DA124(vertex_data_in[124]), .DA125(vertex_data_in[125]),
.DA126(vertex_data_in[126]), .DA127(vertex_data_in[127]),  //
Write Data
```

(param_cache_ctl.v, 3:4-4-3, 6:17-11:12.)

143. For example `param_cache_ctl` provides input data to `vertex_data_in` and then stores the `SP_SX data` in the `u_pc` memory, or uses the `vertex_data_out` signal to reads `SP_SX data` from the `u_pc` memory. Because the parameter cache is operative to store and read the `SP_SX data` (the vertex data) from memory, the parameter cache is operative to maintain the appearance attribute data.

144. In this way, the vertex storage block includes a parameter cache operative to maintain appearance attribute data for a corresponding vertex.

2. *The vertex storage block and a position cache*

145. Claim 3 also recites a vertex storage block that includes *"a position cache operative to maintain position data for a corresponding vertex."* A position

cache, which is called the position memory in the R400 RTL code, is instantiated in *pa_ag.v*, as replicated below:

```
// Instantiate the position memory
// 1 64d x 128w
dum_mem_p2 #(
u_pos_ADDR_WIDTH ,
u_pos_DATA_WIDTH ,
u_pos_WORDS      ,
u_pos_DEBUG
)
u_pos_dum_mem_p2 (
      .iRCLK(sclk),
      .iWCLK(sclk),
      .iMER(pos_re),
      .iMEW(pos_mem_we),
      .iWEN(pos_mem_we),
      .iRADR(pos_raddr),
      .iWADR(pos_mem_waddr),
      .iD(pos_pntsz_ag_mem_data),
      .oQ(pos_rdata));
```

(*pa_ag.v*, 112 :13-113:5.)

146. The input data to the position cache is provided on signal *pos_pntsz_ag_mem_data* at, for example, 113:4 of *pa_ag.v* of the *pa_ag* module. The *pa_ag* module receives *pos_pntsz_ag_mem_data* at 3:13 and 8:3 of *pa_ag.v*.

147. The *pos_pntsz_ag_mem_data* signal is provided by the *pa* module as signal *ccg_ag_pos_pntsz_mem_wrdata*. (*pa.v*, 41:4.) The *pa module* receives the *ccg_ag_pos_pntsz_mem_wrdata* signal from the shader export block interface and clip code generator called the *upa_sxifccg*

module at 49:11-53:11 in *pa.v*. In particular, the *ccg_ag_pos_pntsz_mem_wrdata* is assigned from the *oposition_wrdata* signal. (*pa.v*, 52:23.)

148. The *upa_sxifccg* module is specified in *pa_sxifccg.v*. In the *upa_sxifccg* module, the *oposition_wrdata* is defined as an output. (*pa_sxifccg.v*, 8:22.)

149. The *oposition_wrdata* signal is assigned from signal *position_wrdata* at 17:24 in *pa_sxifccg.v*. The *position_wrdata* signal is defined as the output of the position memory at 26:2 of *pa_sxifccg.v*. The *position_wrdata* signal comes from *upa_ccg_sxifsm* (which is defined in *pa_ccg_sxifsm* module) on an output called *omem_position_wrdata*. (*pa_sxiccg.v*, 15:23-26:25.)

150. In the *pa_ccg_sxifsm* module *omem_position_wrdata* is defined at 4:7 and 7:12 (*pa_ccg_sxifsm.v*), and is assigned a value from *tcl_scratch_mem_position_data*. (*pa_ccg_sxifsm.v*, 18:13.)

151. The *tcl_scratch_mem_position_data* signal receives data from *sx_to_pa_vector* (*Id.* at 40:21, 41:21). The value in *sx_to_pa_vector* is provided by *isx_to_pa_vector_0* or

isx_to_pa_vector_1. (*Id.* at 17:5 or 17:9.) The *isx_to_pa_vector_0* or *isx_to_pa_vector_1* signals are inputs to the *pa_ccg_sxifsm* module. (*Id.* at 6:14, 6:18.) The *isx_to_pa_vector_0* or *isx_to_pa_vector_1* signals are connected to the signals *sx0_receive_fifo_rddata* and *sx1_receive_fifo_rddata* respectively. (*pa_sxifccg.v*, 25:3, 25:7). The *sx0_receive_fifo_rddata* and *sx1_receive_fifo_rddata* signals come from the *read_data* signals at the "receive fifos" (the first-in, first-out, buffers). (*Id.* at 20:3, 21:2). The "receive fifos" have *read_data* from the *sx0_receive_fifo_wrdata* and *sx1_receive_fifo_wrdata signals*. (*Id.* at 19:25, 20:24.) The *sx0_receive_fifo_wrdata* and *sx1_receive_fifo_wrdata signals* are provided by *isx0_receive_fifo_wrdata* and *isx1_receive_fifo_wrdata* (*id.* at 15:19, 15:22) which are defined as inputs at 6:21 and 6:24 of *pa_sxifccg.v*.

152. The isx0_receive_fifo_wrdata and *isx1_receive_fifo_wrdata* signals are provided by *SX0_PA_input_data_wrdata* and *SX1_PA_input_data_wrdata*. (*pa.v*, 50:13, 50:17.) The *SX0_PA_input_data_wrdata* and *SX1_PA_input_data_wrdata* signals are assigned the values of SX0_*PA_input_data_q* and *SX1_PA_input_data_q*. (*Id.* at 36:12-13,

36:18-19.) The *SX0_PA_input_data_q* and *SX1_PA_input_data_q* signals are assigned values from *SX0_PA_input_data* and *SX1_PA_input_data*. (*Id.* at 33:7, 33:13.) The *SX0_PA_input_data* and *SX1_PA_input_data* signals come from *u0_SX_PA_data* and *u1_SX_PA_data*, (*id.* at 36:9-10, 36:18-19), that are defined as coming from the shader export block. (*Id.* at 4:18, 5:2, 8:7, and 8:15.)

153. The *pa* module receives *u0_SX_PA_data* and *u1_SX_PA_data* from the shader export block. The shader export block includes *SX_PA_data*. (*sx.v*, 19:19.) The *SX_PA_data* signal comes from *q_sx_pa_data* (*sx.v*, 20:3) and *q_sx_pa_data* comes from *sx_pa_data* (*id.* at 19:25,) which comes from the signal of the same name in the *export_control* module. (*sx.v*, 22:1) The *sx_pa_data* signal is defined at 7:11 in export_control.v and provided from the *export_buffers* module as the *oclipp_data* signal. (*export_control.v*, 75:10.) The *oclipp_data* signal is defined at 2:25 of export_buffers.v and is assigned from *q_clipp_data*. (*Id.* at 83:16). The *q_clipp_data* signal is assigned from *clipp_data*. (*Id.* at 83:8.) The *clipp_data* signal is assigned in *export_buffers.v* at 78:6-14 and 76:23-77:22, which comes through a queue at 8:13-22 or 8:25-16:4. The data in the queue comes from the shader processor and becomes *ipixel_data0*. In

particular, `SP_SX_data0` is output from the `sp` module. (`sp.v`, 1:21, 3:23.). The `SP_SX_data0` signal provided by the *sp* module becomes *SP0_SX_data0* input in the `sx` module. (`sx.v`, 2:8, 5:10.) The `SP0_SX_data0` signal becomes `q_sp0_sx_data0` (*id.* at 5:16), and is provided to the `export_control` module as `sp0_sx_data0`. (*Id.* at 22:18.)

154. The export_control module is specified in `export_control.v`. The `sp0_sx_data0` signal is defined as an input in `export_control.v` and is then propagated to the export_buffers module as `ipixel_data0`. (`export_control.v`, 75:15.)

155. In this way, a position cache is operative to maintain position data for a corresponding vertex.

D. Claim 5

1. The appearance attribute is position

156. Claim 3 recites "*wherein the appearance attribute is position*." In my analysis of claim 1 in Section IX.A.3, I explained how a shader provides an appearance attribute. As I described in my analysis of claim 3, the appearance attribute is position when the selected input is vertex data.

157. Also, when the *MACC* module parses `iInstruction`, as I discussed in Section IX.A.3, the *MACC* module parses out an export destination called `oExportDst`, as shown using R400 RTL code, replicated below:

```
//grabing the export destination ID.
   //If we are dealing with an export instruction...this value
identifies which
   //attribute is being exported ...please refer to the shader
pipe spec for more details
   //on this
   //-----------------------------------------------
   assign oExportDst = q_Instruction0[17:12];
```

(`macc.v`, 11:15-20.)

158. The `oExportDst` parameter determines whether the data exported from the *sp* module is pixel data or vertex data, and is outputted from the *MACC* module to the `mac_gpr` module as `oexport_dst`. (*Id.* at 3:19.) The `mac_gpr` module then outputs `oexport_dst` to the `vector` unit as `sq_sp_exp_dst`. (`macc_gpr.v`, 14:8.) The `vector` unit outputs `sq_sp_exp_dst`, to the `sp` module as shown using the R400 RTL code below:

```
q0_sq_exp_dst <= sq_sp_exp_dst;
 q1_sq_exp_dst <= q0_sq_exp_dst;
 q2_sq_exp_dst <= q1_sq_exp_dst;
 q3_sq_exp_dst <= q2_sq_exp_dst;
 q4_sq_exp_dst <= q3_sq_exp_dst;
 q5_sq_exp_dst <= q4_sq_exp_dst;
 q6_sq_exp_dst <= q5_sq_exp_dst;
 q7_sq_exp_dst <= q6_sq_exp_dst;
 q8_sq_exp_dst <= q7_sq_exp_dst;
 q9_sq_exp_dst <= q8_sq_exp_dst;
 q10_sq_exp_dst <= q9_sq_exp_dst;
 q11_sq_exp_dst <= q10_sq_exp_dst;
 …
 assign sp_sx_exp_dst   = q10_sq_exp_dst;
```

(vector.v, 21:12-23, 22:7.)

159. The `sp` module outputs `sp_sx_exp_dst` to the shader export block (the `sx` module) as `SP_SX_exp_dest`. (*See* `sp.v`, 1:24, 7:22, 8:12.)

160. The `sx` module receives `SP_SX_exp_dest` from one of two shaders as the `SP0_SX_exporting` and `SP1_SX_exporting` parameter and propagates the `SP0_SX_exporting` and `SP1_SX_exporting` parameters to `uexport_control` defined in the `export_control` module, which is shown using the R400 RTL code below:

```
module sx(/*AUTOARG*/
…
   SP1_SX_exporting, SP0_SX_exp_dest, SP1_SX_exp_dest,
…
input [5:0]       SP0_SX_exp_dest, SP1_SX_exp_dest; //these are
coming straight from the destination pointer of the ALU
instruction
                                                     //SP does
nothing else other than pipelining them through.
…
ati_dff_in #(6)
usp0_sx_exp_dst(sclk,SP0_SX_exp_dest,q_sp0_sx_exp_dest);
ati_dff_in #(6)
usp1_sx_exp_dst(sclk,SP1_SX_exp_dest,q_sp1_sx_exp_dest);
…
export_control uexport_control(
…
.sp0_sx_exp_dest(q_sp0_sx_exp_dest),
.sp1_sx_exp_dest(q_sp1_sx_exp_dest),
…
);
```

(`sx.v`, 1:12, 2:12, 6:7-9, 6:24-25, 21:5, 22:16-17, 23:29.)

161. The `export_control` module identifies the types of appearance attributes included in the shader export block (*sx* module) that are received from the shader based on the `sp0_sx_exp_dest` parameter. This is shown using R400 RTL code below:

```
//00:no export
//01:vertex export
//10:pixel export
assign     export_type = (sp0_sx_exp_alu_id) ?
{sp0_sx_exporting[0] & q_exp_pix_alu1,sp0_sx_exporting[0] &
~q_exp_pix_alu1}:
           {sp0_sx_exporting[0] &
q_exp_pix_alu0,sp0_sx_exporting[0] & ~q_exp_pix_alu0};

   always @(/*AUTOSENSE*/`COLOR0 or `COLOR1 or `COLOR2 or `COLOR3
or `COLORFOG0 or `COLORFOG1 or `COLORFOG2 or `COLORFOG3 or
`PIXEL_EXPORT or `POSITION or `SPRITE_EDGE
or `VERTEX_EXPORT or `Z_DATA or export_type or sp0_sx_exp_dest)
     begin
     case(export_type)
       `PIXEL_EXPORT:
         begin
            position_aux = 1'b0;
            case(sp0_sx_exp_dest)
            `COLOR0:attribute_offset = 3'h0;
            `COLOR1:attribute_offset = 3'h1;
            `COLOR2:attribute_offset = 3'h2;
            `COLOR3:attribute_offset = 3'h3;
            `COLORFOG0:attribute_offset = 3'h0;
            `COLORFOG1:attribute_offset = 3'h1;
            `COLORFOG2:attribute_offset = 3'h2;
            `COLORFOG3:attribute_offset = 3'h3;
            `Z_DATA:attribute_offset = 3'h4;
            endcase // case(sp0_sx_exp_dest)
         end // case: VERTEX
       `VERTEX_EXPORT:
         begin
            case(sp0_sx_exp_dest)
            `POSITION:
              begin
                 attribute_offset = 3'h0;    // + count of the
position vectors that have been exported so far
                 position_aux = 1'b0;
              end
            `SPRITE_EDGE:
                begin
                   attribute_offset = 3'h4; //starting offset is
      always relative position 4
                   position_aux = 1'b1;
                end
               endcase // case(sp0_sx_exp_dest
            end // case: VERTEX
          default : attribute_offset = 3'h0;
        endcase // case(sp0_sx_exporting)
       end // always @ (...
```

(`export_control.v`, 34:5-38:12.)

162. At 35:12, the `sp0_sx_exp_dest` parameter can be "POSITION," for and `export type` "VERTEX_EXPORT" which indicates that the vertex data has an appearance attribute that is position. As such, the appearance attribute included in `SX_SP_data` is position.

2. The position attribute is associated with a corresponding vertex

163. Claim 5 also recites "the position attribute is associated with a corresponding vertex when the selected one of the plurality of inputs is vertex data." As I discussed in Section IX.A.1, the selected input includes vertex data. When the selected input is vertex data, the shader generates a position attribute using the corresponding `SQ_SP_vsr_data` and SQ_`SP_instruct` inputs of the selected one of the plurality of inputs, which is included in `SP_SX_Data`.

E. Claim 6

1. The appearance attribute is color

164. Claim 6 depends from claim 5 and recites the graphics processing system of claim 5 "*wherein the appearance attribute is color*." As I discussed in Section VI.A, the appearance attribute is color. In particular, the attribute is color, when the selected input is pixel data.

165. The `export_control` module identifies the types of appearance attributes included in the shader export block (`sx` module) received from the shader based on the `sp0_sx_exp_dest` parameter as shown using R400 RTL code below:

```
//00:no export
//01:vertex export
//10:pixel export
assign     export_type = (sp0_sx_exp_alu_id) ?
{sp0_sx_exporting[0] & q_exp_pix_alu1,sp0_sx_exporting[0] &
~q_exp_pix_alu1}:
           {sp0_sx_exporting[0] &
q_exp_pix_alu0,sp0_sx_exporting[0] & ~q_exp_pix_alu0};

   always @(/*AUTOSENSE*/`COLOR0 or `COLOR1 or `COLOR2 or `COLOR3
or `COLORFOG0 or `COLORFOG1 or `COLORFOG2 or `COLORFOG3 or
`PIXEL_EXPORT or `POSITION or `SPRITE_EDGE
or `VERTEX_EXPORT or `Z_DATA or export_type or sp0_sx_exp_dest)
      begin
       case(export_type)
         `PIXEL_EXPORT:
          begin
             position_aux = 1'b0;
             case(sp0_sx_exp_dest)
             `COLOR0:attribute_offset = 3'h0;
             `COLOR1:attribute_offset = 3'h1;
             `COLOR2:attribute_offset = 3'h2;
             `COLOR3:attribute_offset = 3'h3;
             `COLORFOG0:attribute_offset = 3'h0;
```

```
            `COLORFOG1:attribute_offset = 3'h1;
            `COLORFOG2:attribute_offset = 3'h2;
            `COLORFOG3:attribute_offset = 3'h3;
            `Z_DATA:attribute_offset = 3'h4;
            endcase // case(sp0_sx_exp_dest)
        end // case: VERTEX
      `VERTEX_EXPORT:
        begin
            case(sp0_sx_exp_dest)
            `POSITION:
              begin
                  attribute_offset = 3'h0;     // + count of the
position vectors that have been exported so far
                  position_aux = 1'b0;
              end
            `SPRITE_EDGE:
              begin
                  attribute_offset = 3'h4; //starting offset is
always relative position 4
                  position_aux = 1'b1;
              end
            endcase // case(sp0_sx_exp_dest
        end // case: VERTEX
      default : attribute_offset = 3'h0;
    endcase // case(sp0_sx_exporting)
   end // always @ (...
```

(*export_control.v*, 34:5-38:12.)

166. At 34:24-35:5, the *sp0_sx_exp_dest* parameter can be "*COLOR0*," "*COLOR1*," "*COLOR2*," "*COLOR3*," "*COLORFOG0*," "*COLORFOG1*," "*COLORFOG2*," and "*COLORFOG3*" for and *export type* "*PIXEL_EXPORT*" which indicates that the pixel data has an appearance attribute that is color. As such, the appearance attribute included in the *SX_SP_data0-3* is color.

167. Further, as I described in Section IX.C.1, the *sx* module receives the *SX_SP_data0* data as *SP0_SX_data0*. (*sx.v*, 2:8, 5:10.) The

`SP0_SX_data0` becomes `q_sp0_sx_data0` (*Id.* at 5:16.), and is provided to the `export_control` module. (*Id.* at 22:18.)

168. The `export_control` module receives `q_sp0_sx_data0` as `sp0_sx_data0` (`export_control.v`, 2:3, 3:13), and assigns `sp0_sx_data0` to `q0_sp0_data0`. (*Id.* at 9:4.) The `q0_sp0_data0` signal is provided to `uexport_buffers` (defined in `export_buffers.v` as `export_buffers` module) as `ipixel_data0`. (*Id.* at 75:15.) The `ipixel_data0` signal indicates that the data is pixel data.

169. In the `export_buffers` module, `ipixel_data0` is passed through a queue, and outputted as `buff0_out`. (`export_buffers.v`, 8:13-22 or 8:25-16:4.) The `buff0_out` parameters is assigned to `bank0_data0` (*id.* at 72:20) and then to `rb0_data`. (*Id.* at 78:21.) The `rb0_data` is assigned to `orb0_data` (*id* at 83:12), which is defined as an output of `export_buffers` module. (*Id.* at 2:24.)

170. The `export_control` module receives the `export_buffers` parameter from `uexport_buffers` as `sx_rb0_color_data`. (`export_control.v`, 75:4.) The `sx_rb0_color_data` signal is provided as an output of the `export_control` module and to the `sx` module. (*Id.* at 1:19,

5:9) The *sx* module receives the *sx_rb0_color_data* signal as *sx_rb0_color_data*. (*sx.v*, 21:20.)

171. The *sx_rb0_color_data* is assigned to *q_sx_rb0_color_data* (*id.* at 15:26) and *q_sx_rb0_color_data* is assigned to SX_RB0_color_data, (*id.* at 16:9), which is an output of the *sx* module. (*Id.* at 1:22, 15:13.)

172. The same analysis also applies to *SP0_SX_data1*, *SP0_SX_data2*, and *SP0_SX_data3* using corresponding signal names.

173. Because *SP0_SX_data0*, which includes an appearance attribute, is converted to *q_sx_rb0_color_data* which includes color, the appearance attribute is color.

2. The color attribute is associated with a corresponding pixel.

174. Claim 6 also recites "*the color attribute is associated with a corresponding pixel when the selected one of the plurality of inputs is pixel data.*" When the selected input is pixel data, the shader (*sp* module) generates a color attribute using the corresponding *SX_SP_data0-3* and *SQ_SP_instruct* inputs of the selected one of the plurality of inputs, which is included in *SP_SX_Data0-3*.

175. Further as I discussed in Section IX.E.1, the color is associated with the pixel data.

F. Claim 8

176. Claim 8 recites the graphics processing system of claim 1 "*wherein the appearance value is depth*." As I discussed in Section IX.A.3, the graphics processor of claim 1 provides an appearance attribute, and as I discussed in Section IX.E.1 and IX.E.2, the appearance attribute is associated with a corresponding pixel.

177. Also, as I discussed in Section IX.E.1, the `export_control` module identifies the types of appearance attributes included in the shader export block (*sx* module), and received from the shader based on the `sp0_sx_exp_dest` parameter as shown using R400 RTL code below:

```
//00:no export
//01:vertex export
//10:pixel export
assign     export_type = (sp0_sx_exp_alu_id) ?
{sp0_sx_exporting[0] & q_exp_pix_alu1,sp0_sx_exporting[0] &
~q_exp_pix_alu1}:
           {sp0_sx_exporting[0] &
q_exp_pix_alu0,sp0_sx_exporting[0] & ~q_exp_pix_alu0};

   always @(/*AUTOSENSE*/`COLOR0 or `COLOR1 or `COLOR2 or `COLOR3
or `COLORFOG0 or `COLORFOG1 or `COLORFOG2 or `COLORFOG3 or
`PIXEL_EXPORT or `POSITION or `SPRITE_EDGE
or `VERTEX_EXPORT or `Z_DATA or export_type or sp0_sx_exp_dest)
     begin
      case(export_type)
        `PIXEL_EXPORT:
          begin
             position_aux = 1'b0;
```

```
                case(sp0_sx_exp_dest)
                `COLOR0:attribute_offset = 3'h0;
                `COLOR1:attribute_offset = 3'h1;
                `COLOR2:attribute_offset = 3'h2;
                `COLOR3:attribute_offset = 3'h3;
                `COLORFOG0:attribute_offset = 3'h0;
                `COLORFOG1:attribute_offset = 3'h1;
                `COLORFOG2:attribute_offset = 3'h2;
                `COLORFOG3:attribute_offset = 3'h3;
                `Z_DATA:attribute_offset = 3'h4;
                endcase // case(sp0_sx_exp_dest)
            end // case: VERTEX
          `VERTEX_EXPORT:
            begin
                case(sp0_sx_exp_dest)
                `POSITION:
                  begin
                      attribute_offset = 3'h0;    // + count of the
position vectors that have been exported so far
                      position_aux = 1'b0;
                  end
                `SPRITE_EDGE:
                  begin
                      attribute_offset = 3'h4; //starting offset is
always relative position 4
                      position_aux = 1'b1;
                  end
                endcase // case(sp0_sx_exp_dest
            end // case: VERTEX
          default : attribute_offset = 3'h0;
        endcase // case(sp0_sx_exporting)
      end // always @ (...
```

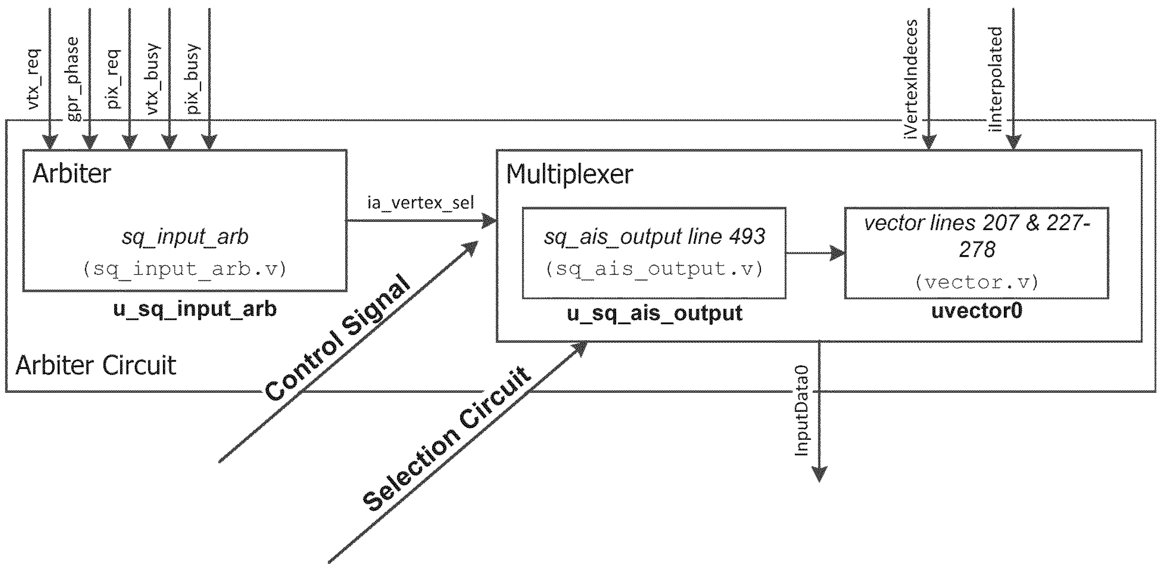
(`export_control.v`, 34:5-38:12.)

178. At 35:6, the `sp0_sx_exp_dest` parameter can be "Z_DATA," which indicates that the pixel data has a depth parameter. As such, the pixel data has appearance value that is depth.

G. Claim 9

179. Claim 9 recites the selection circuit and a control signal provided by an arbiter. Based on my understanding of the R400 RTL code, I have generated a

figure below which represents my understanding of the components, and describe the code with the reference to the figure.



1. *The selection circuit*

180. Claim 9 recites a graphics processor of claim 1, "*further including a selection circuit, wherein the selection circuit is a multiplexer.*" As discussed in Section IX.A.3, the input arbiter called *u_sq_input_arb* selects between a vertex request (*vtx_req*) and a request (*pix_req*) and passes the *vtx_sel* signal as *ia_vertex_sel* to the *u_sq_ais_output*.

181. The *u_sq_ais_output* module receives the *ia_vertex_sel* control signal from the arbiter and generates a *SQ_SP_gpr_input_sel* signal

using a multiplexer decoder stage on 21:7 (indicated as line 493 in the figure in Section IX.G), as shown using the R400 RTL code below:

```
    // SQ_SP_gpr_phase
    // SQ_SP_gpr_input_sel
    //
    always @(posedge clk)
      begin
       SQ_SP_gpr_phase <= gpr_phase;
       SQ_SP_gpr_input_sel <= {ia_vertex_sel, ~ia_vertex_sel}; //
00: cnt, 01: pix, 10: vtx (fix needed for count)
      end
```

(`sq_ais_output.v`, 21:1-9.)

182. The `SQ_SP_gpr_input_sel` is provided to the `sq` module as `sq_sp_gpr_input_mux`. The signal `sq_sp_gpr_input_mux` is the control signal to the multiplexer. The multiplexer is included inside each of the vector units `uvector0-3` of the `sp` module.

183. The `sp` module receives the `sq_sp_gpr_input_mux` as `sq_sp_gpr_input_mux` (`sp.v`, 2:7, 9:11). The `sp` module converts `sq_sp_gpr_input_mux` to `q_sq_gpr_phase_mux` (*id.* at 17:16) and propagates `q_sq_gpr_phase_mux` to each vector unit `uvector0-3`. (*Id.* at 16:1, 16:25, 17:16, 18:8.)

184. The `vector` units `uvector0-3` receive `q_sq_gpr_phase_mux` as `sq_sp_gpr_phase_mux`. (`vector.v` , 1:21, 3:2.) The selection circuitry in `vector` units `uvector0-3` uses `sq_sp_gpr_phase_mux` as a control

signal to a selection circuit that selects one of a plurality of inputs from the vertex indicies (which are the vertex data) and the interpolated pixel inputs (which are the pixel data). For example, the vector module uses the `sq_sp_gpr_input_mux` to select the vertex data input (`iVertexIndices`) or the pixel data input (`iInterpolated`), using the R400 RTL code replicated below:

```
//--------------------------------------------------------------
---------------------------------------------------------
   //Muxing logic to select from data comming from the
Interpolators(in reality more than just interpolated
data....there can be
   //also faceness and XY data), AutoCount data and Vertex
Indices comming from the staging registers.
   //Each MACC unit has its own mux logic since the controls are
phased out by one cycle from one MACC to the other.
   //-----------------------------------------------------------
---------------------------------------------------------
   //muxing logic for the inputs of the first MACC
   always @(/*AUTOSENSE*/iAutoCount or iInterpolated or
iVertexIndices
          or sq_sp_gpr_input_mux)
     begin
      case(sq_sp_gpr_input_mux)
        2'b00: InputData0 = iAutoCount ;
        2'b01: InputData0 = iInterpolated ;
        2'b10: InputData0 = iVertexIndices ;
        default: InputData0 = iInterpolated;
      endcase // case(sq_sp_gpr_input_mux)
     end

   //muxing logic for the inputs of the second MACC
   always @(/*AUTOSENSE*/iAutoCount or iInterpolated or
iVertexIndices
          or q0_gpr_input_mux)
     begin
      case(q0_gpr_input_mux)
        2'b00: InputData1 = iAutoCount ;
        2'b01: InputData1 = iInterpolated ;
        2'b10: InputData1 = iVertexIndices ;
        default: InputData1 = iInterpolated;
      endcase // case(q0_gpr_input_mux)
     end
   //muxing logic for the inputs of the third MACC
   always @(/*AUTOSENSE*/iAutoCount or iInterpolated or
iVertexIndices
          or q1_gpr_input_mux)
     begin
      case(q1_gpr_input_mux)
        2'b00: InputData2 = iAutoCount ;
        2'b01: InputData2 = iInterpolated ;
        2'b10: InputData2 = iVertexIndices ;
        default: InputData2 = iInterpolated;
      endcase // case(q1_gpr_input_mux)
     end

   //muxing logic for the inputs of the fourth MACC
   always @(/*AUTOSENSE*/iAutoCount or iInterpolated or
iVertexIndices
          or q2_gpr_input_mux)
     begin
      case(q2_gpr_input_mux)
        2'b00: InputData3 = iAutoCount ;
        2'b01: InputData3 = iInterpolated ;
        2'b10: InputData3 = iVertexIndices ;
        default: InputData3 = iInterpolated;
      endcase // case(q2_gpr_input_mux)
     end
```

(`vector.v`, 10:2-12:6.)

185. The selected input is provided as `InputData0`, `InputData1`, `InputData2`, and `InputData3`.

186. In this way, the R400 RTL code includes a selection circuit that is a multiplexer.

2. *The control signal*

187. Claim 9 also recites "*the control signal is provided by an arbiter, wherein the arbiter is coupled to the multiplexer*." As discussed above, the `u_sq_input_arb` input arbiter includes the arbiter that provides an

`ia_vertex_sel` signal which is the control signal. As discussed in Section IX.A.2, the *`u_sq_ais_output`* receives the *`ia_vertex_sel`* `signal` (*`sq.v`*, 79:8) and converts *`ia_vertex_sel`* to *`SQ_SP_gpr_input_sel`*. (*Id.* at 80:6.) The *`u_sq_ais_output`* signal provides the signal to the shader included in the *`sp`* module as *`Sq_sp_gpr_input_mux`*. (*Id.* at 2:16, 9:4, 80:6.) In this way, the control signal is provided by the arbiter.

3. The arbiter is coupled to the multiplexer.

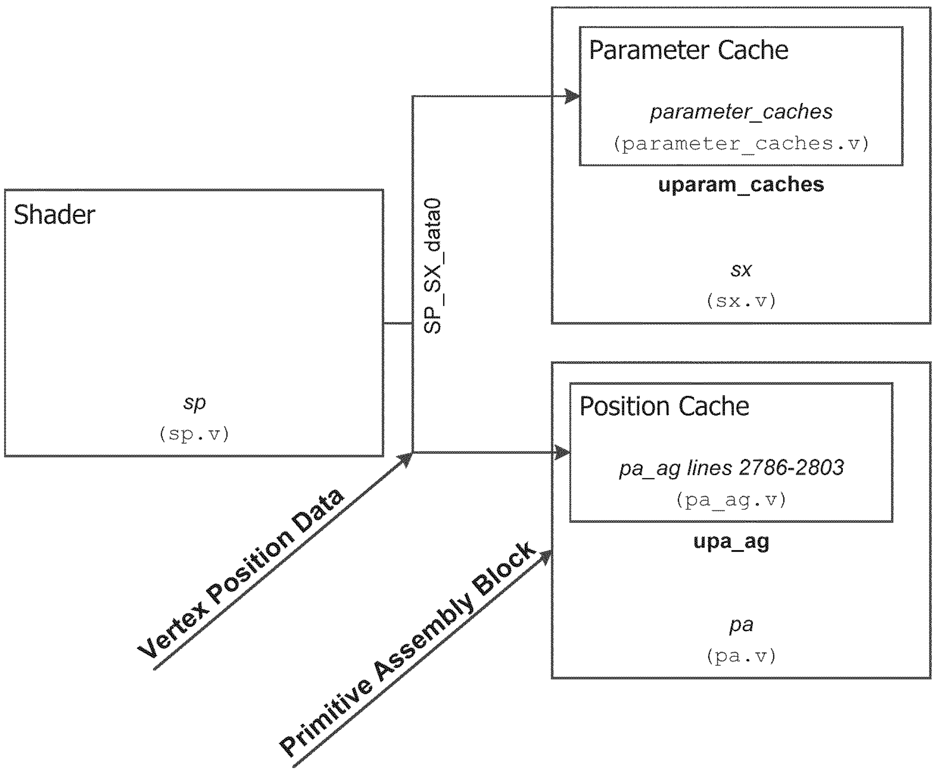
188. Claim 9 also recites "wherein the arbiter is coupled to the multiplexer."

189. The multiplexer is included in the vector unit as shown in Section IX.A.2. The *`u_sq_ais_output`* signal is generated by the arbiter is converted to *`sq_sp_gpr_input_mux`* in the *`sp`* module and to *`sq_sp_gpr_input_mux`* in the vector unit. This is a control signal to the multiplexer in the vector unit and shows that the arbiter is coupled to the multiplexer.

H. Claim 10

190. Claim 10 recites the graphics processor of claim 1. Based on my understanding of the R400 RTL code, I have generated a figure below which

represents my understanding of the components, and describe the code with reference to the figure.



1. The vertex position data

191. Claim 10 recites the graphics processor of claim 1, "*wherein the shader provides vertex position data*." As discussed in Section IX.C.2, the *sp* module provides *SP_SX_data0*, *SP_SX_data1*, *SP_SX_data2*, and *SP_SX_data3* which includes vertex position data. The *SP_SX_data0*, *SP_SX_data1*, *SP_SX_data2*, and *SP_SX_data3* signals are provided to the shader export block.

192. For simplicity, I limit my discussion to the shader providing `SP_SX_data0` which is sufficient to practice the claim.

2. The primitive assembly block coupled to the shader

193. The primitive assembly block in the graphics processing system is included in the `pa` module. The `pa` module is defined in `pa.v`. The primitive assembly block converts the vertex position data into a list of primitives. The `pa` module is coupled to the shader (included in the `sp` module) through one of the shader export blocks (the `sx` modules).The coupling between the shader and the shader export block (the `sx` module) is discussed in Section IX.A.3 and IX.C.1.

194. The primitive assembly block (the `pa` module) includes a `PA_SX/SX_PA` interface to the two shader export blocks. Here, for simplicity, I limit my discussion to a single shader export block which is sufficient to practice the claim. The R400 RTL code for the `PA_SX/SX_PA` interface is replicated below.

```
// -----------------------------------------
// interface to the shader export 0 block
// -----------------------------------------
u0_SX_PA_send,
u0_SX_PA_data,
u0_PA_SX_req,
u0_PA_SX_sp_id,
u0_PA_SX_offset,
u0_PA_SX_aux,
u0_PA_SX_last,
```

(*pa.v*, 4:14-23.)

195. The primitive assembly block requests data from the *sx* module (the shader export block) using *u0_PA_SX_req*, *u0_PA_SX_sp_id*, *u0_PA_SX_offset*, *u0_PA_SX_aux*, and *u0_PA_SX_last*.

196. In response, the primitive assembly block receives the data from the *sx* module using the *u0_SX_PA_send* and *u0_SX_PA_data* interface.

197. The *u0_SX_PA_data* data is provided to the export buffers *uexport_buffers* of the *sx* module. The *sx* module receives the request from the *pa* module using the *PA_SX_req*, *PA_SX_sp_id*, *PA_SX_offset*, *PA_SX_aux*, *PA_SX_last* (*sx.v*, 2:23-24) and propagates the request to *uexport_control*, using the R400 RTL code below:

```
export_control uexport_control(
…
.pa_sx_req(q_pa_sx_req), .pa_sx_sp_id(q_pa_sx_sp_id),
.pa_sx_offset(q_pa_sx_offset),.pa_sx_aux(q_pa_sx_aux),
            .pa_sx_last(q_pa_sx_last)
  );
```

(*Id.* at 21:5, 23:26-29.)

198. The *export_control* module passes the request to *uexport_buffers*, and receives 1) the *sx_pa_data* which includes the vertex position data and 2) the *sx_pa_send* signals. (*export_control.v*,

75:10.) The `export_control` module then propagates those signals to the *sx* module using the R400 RTL code below:

```
module export_control(/*AUTOARG*/
   // Outputs
…
sx_pa_send, sx_pa_data,
```

(*Id.* at 1:13-14, 1:23.)

199. The `sx` module transmits the `sx_pa_send`, `sx_pa_data` as `SX_PA_data` and `SX_PA_send` to the primitive assembly block (the `pa` module). (`sx.v`, 1:25-2:1.)

200. The primitive assembly block receives `SX_PA_data` as `u0_SX_PA_data` and `u1_SX_PA_data`.

201. As explained above, the primitive assembly block (the `pa` module) is coupled to the shader (included in the `sp` module) through the shader export shader block (the `sx` module).

3. *The primitive assembly block is operative to generate primitives.*

202. Claim 10 also recites the primitive assembly block is "*operative to generate primitives in response to the vertex position data.*" The primitive assembly block (the `pa` module) receives `SX_PA_data` that includes the vertex position data. The `pa` module then generates primitives as coded in the *pa*

module. The primitive assembly block is an old component of the graphics processing system that has been adapted to receive data from the shader export block, and is necessary to generate primitives from the vertex position data.

203. Once the primitive assembly block generates primitives, the primitive assembly block provides the primitives to the raster engine (also referred to as the scan converter and defined in the *sc* module), using the *PA_SC* interface below:

```
module pa (
…
// -------------------------------------------
    // interface to the scan converter
    // -------------------------------------------
    PA_SC_p0,
    PA_SC_p1,
    PA_SC_p2,
    PA_SC_p3,
    PA_SC_p4,
    PA_SC_xy0,
    PA_SC_xy1,
    PA_SC_xy2,
    PA_SC_zminmax,
    PA_SC_cntl,
    PA_SC_phase,
    PA_SC_valid,
    PA_SC_v0_indx,
    SC_PA_earlyfrz
    );
…
    // interface to scan converter
    output [17:0]  PA_SC_xy0;
    output [17:0]  PA_SC_xy1;
    output [17:0]  PA_SC_xy2;
    output [31:0]  PA_SC_p0;
    output [39:0]  PA_SC_p1;
    output [31:0]  PA_SC_p2;
    output [31:0]  PA_SC_p3;
    output [31:0]  PA_SC_p4;
    output [13:0]  PA_SC_zminmax;
    output [29:0]  PA_SC_cntl;
    output  [1:0]  PA_SC_phase;
    output         PA_SC_valid;
output  [1:0]  PA_SC_v0_indx;
```

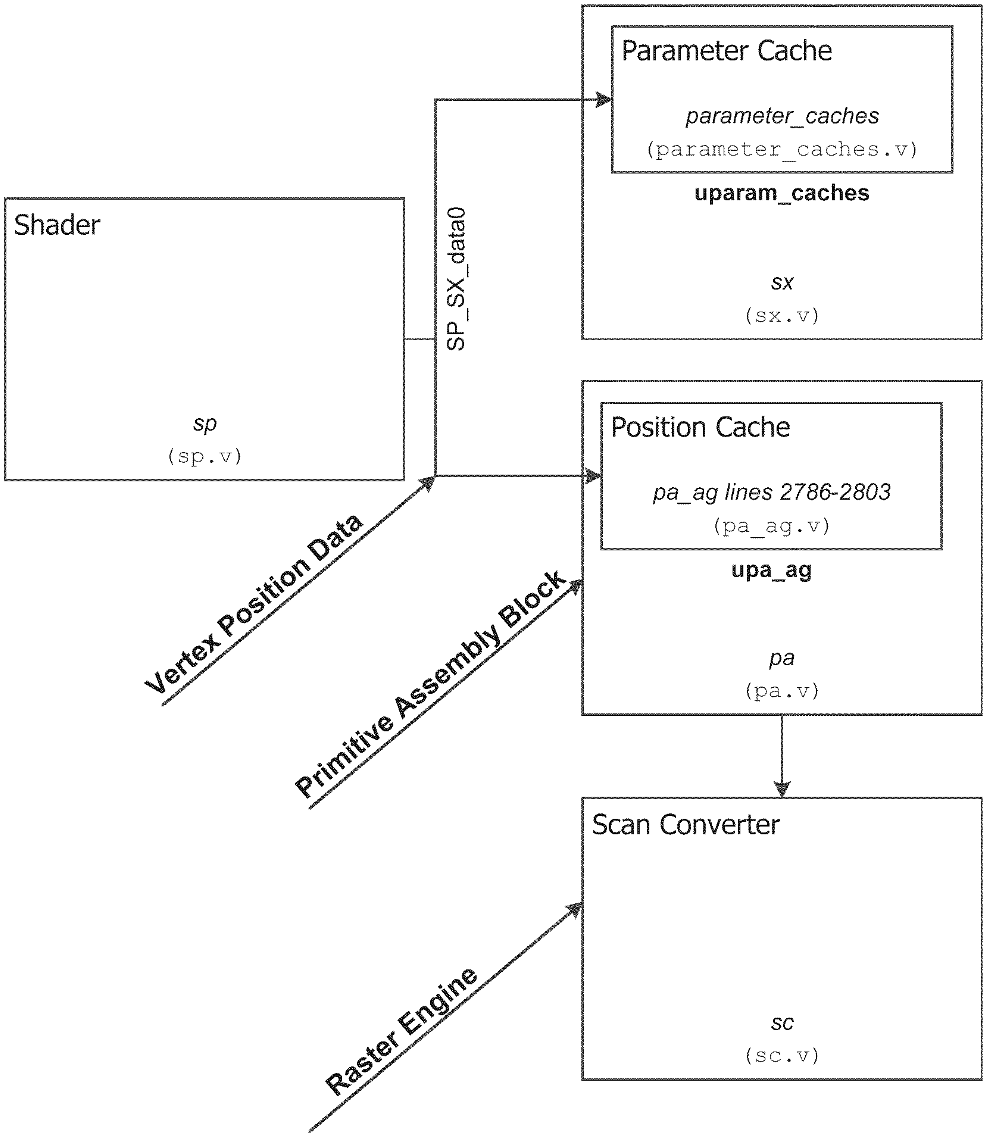
(`pa.v`, 2:2, 5:8-25, 8:21-9:9.)

204. In particular, the primitive assembly block transmits the generated primitives in at least the *PA_SC_p0*, *PA_SC_p1*, *PA_SC_p2*, *PA_SC_p3*, *PA_SC_p4*, *PA_SC_xy0*, *PA_SC_xy1*, *or PA_SC_xy2* signals to the raster engine (the *sc* module).

205. As explained above, the primitive assembly block (the *pa* module) is operative to generate primitives from the vertex position data.

I. Claim 11

206. Claim 11 recites the graphics processor of claim 10. Based on my understanding of the R400 RTL code, I have generated a figure below which represents my understanding of the components, and describes the code with reference to the figure.



1. The Raster Engine

207. Claim 11 recites the graphics processor of claim 10, "*further including a raster engine.*" The raster engine in the graphics processing system is included in the *sc* module (also referred to as a scan converter). The `sc` module receives the primitives from the primitive assembly block (the `pa` module), as I

have described in Section IX.C.2. Replicated below, is the R400 RTL code showing the raster engine receiving the primitive data:

```
module sc (
…
     // -------------------------------------------
    // Interface to the PA Setup Unit
    // -------------------------------------------
    PA_SC_p0,
    PA_SC_p1,
    PA_SC_p2,
    PA_SC_p3,
    PA_SC_p4,
    PA_SC_xy0,
    PA_SC_xy1,
    PA_SC_xy2,
    PA_SC_zminmax,
    PA_SC_cntl,
    PA_SC_phase,
    PA_SC_v0_indx,
    PA_SC_valid,
    SC_PA_earlyfrz,
…
// -------------------------------------------
    // Interface to the PA Setup Unit
    // -------------------------------------------
    input  [17:0]  PA_SC_xy0;
    input  [17:0]  PA_SC_xy1;
    input  [17:0]  PA_SC_xy2;
    input  [31:0]  PA_SC_p0;
    input  [39:0]  PA_SC_p1;
    input  [31:0]  PA_SC_p2;
    input  [31:0]  PA_SC_p3;
    input  [31:0]  PA_SC_p4;
    input  [13:0]  PA_SC_zminmax;
    input  [29:0]  PA_SC_cntl;
    input   [1:0]  PA_SC_phase;
    input   [1:0]  PA_SC_v0_indx;
    input          PA_SC_valid;
    output         SC_PA_earlyfrz;
```

(*sc.v*, 2:4, 3:24-4:15, 9:20-10:11.)

208. Through the *PS_SC* interface, the raster engine (the *sc* module) is coupled to the primitive assembly bock (the *pa* module).

2. *Generating the pixel parameter*

209. The *sc* module is operative to generate pixel parameter data using the primitive data received from the primitive assembly block. Example pixel parameter data that the raster engine generates is shown using the R400 RTL code below:

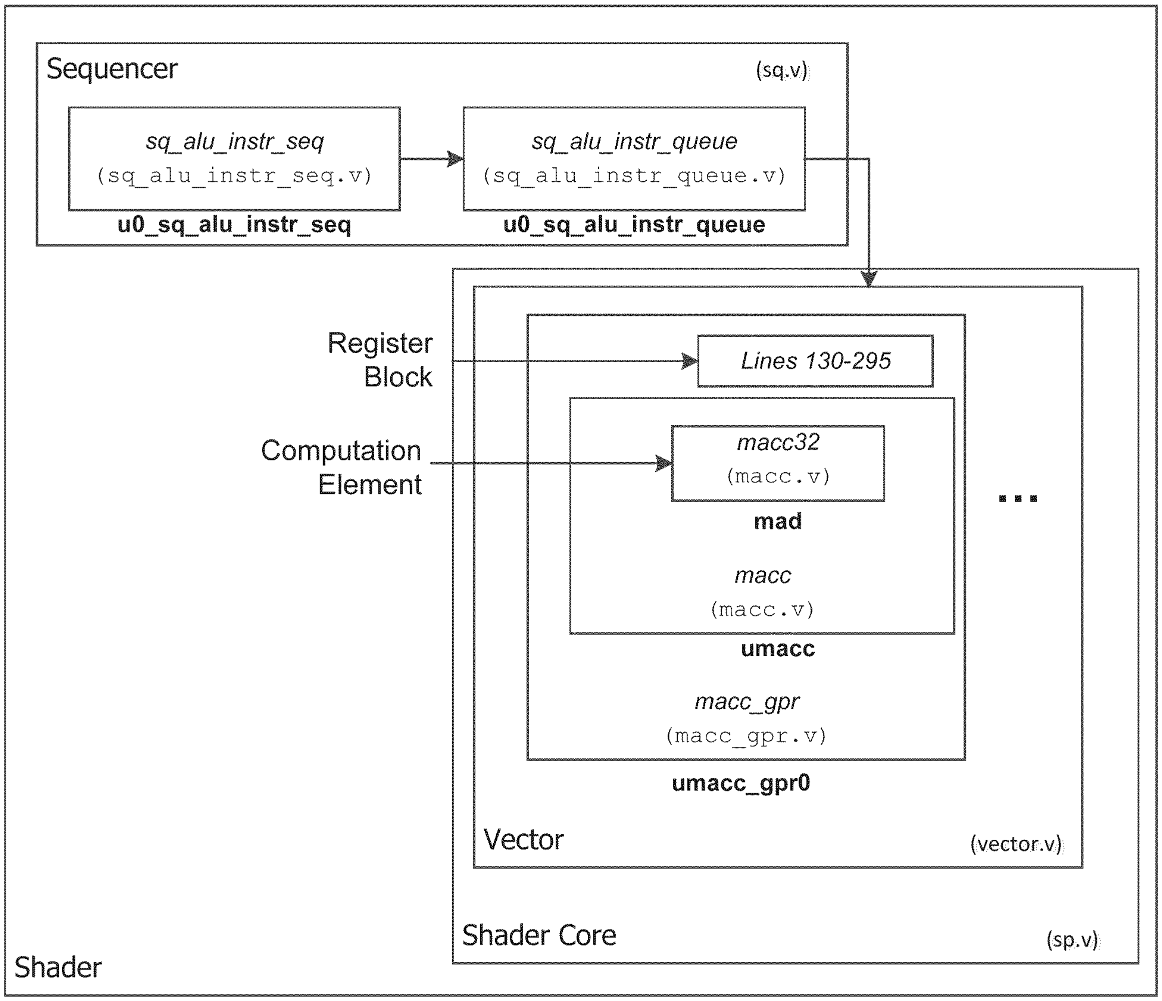
```
// Concatenate outputs of sc_quadmask to create write data for
tile fifo.
    assign tile_ff_wr_data[`SC_TD_LAST_TILE]    = qm_last_tile;
    assign tile_ff_wr_data[`SC_TD_ZMASK_NEEDED] =
qm_z_mask_needed;
    assign tile_ff_wr_data[`SC_TD_EVENT]        = qm_event;
    assign tile_ff_wr_data[`SC_TD_XMIN]         = qm_xmin;
    assign tile_ff_wr_data[`SC_TD_XMAX]         = qm_xmax;
    assign tile_ff_wr_data[`SC_TD_YMIN]         = qm_ymin;
    assign tile_ff_wr_data[`SC_TD_YMAX]         = qm_ymax;
    assign tile_ff_wr_data[`SC_TD_BBFRACTBITS]  =
qm_bb_fract_bits;
    assign tile_ff_wr_data[`SC_TD_XDIR]         = qm_xdir;
    assign tile_ff_wr_data[`SC_TD_YDIR]         = qm_ydir;
    assign tile_ff_wr_data[`SC_TD_TILEX]        = qm_tilex;
    assign tile_ff_wr_data[`SC_TD_TILEY]        = qm_tiley;
    assign tile_ff_wr_data[`SC_TD_TILEX_M3]     = qm_tilex_m3;
    assign tile_ff_wr_data[`SC_TD_TILEY_M3]     = qm_tiley_m3;
    assign tile_ff_wr_data[`SC_TD_XMAJOR]       = qm_xmajor;
    assign tile_ff_wr_data[`SC_TD_E0_SAMPLE]    = qm_e0;
    assign tile_ff_wr_data[`SC_TD_E1_SAMPLE]    = qm_e1;
    assign tile_ff_wr_data[`SC_TD_E2_SAMPLE]    = qm_e2;
    assign tile_ff_wr_data[`SC_TD_E0_DX]        = qm_dxe0;
    assign tile_ff_wr_data[`SC_TD_E0_DY]        = qm_dye0;
    assign tile_ff_wr_data[`SC_TD_E1_DX]        = qm_dxe1;
    assign tile_ff_wr_data[`SC_TD_E1_DY]        = qm_dye1;
    assign tile_ff_wr_data[`SC_TD_E2_DX]        = qm_dxe2;
    assign tile_ff_wr_data[`SC_TD_E2_DY]        = qm_dye2;
    assign tile_ff_wr_data[`SC_TD_STIPPLE_MASK] = 8'b11111111;
```

(*Id.* at 83:19-84:19.)

210. As such, the primitive assembly block (the `pa` module) is operative to generate primitives from the vertex position data.

J. Claim 13

211. Claim 13 recites three components, a register block, a computational element, and a sequencer. Below, I have generated a figure based on my understanding of the R400 RTL code that shows the relationship between these components.



1. The register block

212. Claim 13 recites a graphics processor of claim 1, "*wherein the shader includes a register block for maintaining the selected one of the plurality of inputs*." As discussed in Section IX.A.3, each vector unit instantiates four register blocks, *umacc_gpr0*, *umacc_gpr1*, *umacc_gpr2*, and *umacc_gpr3*, that are defined in the *macc_gpr* module. The *macc_gpr* module is specified in macc_gpr.v. One or more of *umacc_gpr0*, *umacc_gpr1*, *umacc_gpr2*, and *umacc_gpr3* form a register block.

213. Each *macc_gpr* module receives one instance of input data (*InputData0*, *InputData1*, *InputData2*, and *InputData3*) as the *iInterpolated* parameter, which is the selected one of a plurality of inputs, as shown using the R400 RTL code below.

```
module macc_gpr(
           /*AUTOARG*/
   // Outputs
   oScalarInput, oScalarOpcode, oVectorOutput, oRegData,
oexport_dst,
   // Inputs
   sq_sp_instruct, sq_sp_instruct_start, sq_sp_gpr_rd_addr,
   sq_sp_gpr_wr_addr, sq_sp_gpr_phase_mux, sq_sp_mem_wr_ena,
   sq_sp_mem_rd_ena, sq_sp_wr_ena, sq_sp_gpr_cmask,
iInterpolated,
   sq_sp_constant, iScalarData, tp_sp_data, tp_sp_gpr_dst,
   tp_sp_gpr_cmask, tp_sp_data_valid, sclk, srst
   );
```

(*macc_gpr.v*, 1:13-23.)

214. The `macc_gpr` module also includes a memory called "“`ugpr_mem`” of module type “`rfsd2_128x128cm2sw1.`” (*Id.* at 8:1; also as shown in my figure in Section.IX.J.as lines 130-295.) The `macc_gpr` module stores the data in the `iInterpolated` in this memory. To store the data in the `iInterpolated` into the memory, the `macc_gpr` module includes a multiplexer that controls the input to the memory, as shown below, and selects an input (such as `iInterpolated`) that is stored in the “`ugpr_mem`” memory as `InputGPR`:

```
/----------------------------------------------------------------
-------------------------------------------------------------/
   //The phase mux controlling the write input port into GPRs
(register file write port)
   //-------------------------------------------------------------
-------------------------------------------------------------------/
   always@(/*AUTOSENSE*/VectorResult or iInterpolated or
iScalarData
        or sq_sp_gpr_phase_mux or tp_sp_data)
     begin
      case(sq_sp_gpr_phase_mux)
        2'b00: InputGPR = iInterpolated;
        2'b01: InputGPR = tp_sp_data;
        2'b10: InputGPR = VectorResult;
        2'b11: InputGPR = iScalarData;
        default: InputGPR = iInterpolated;
      endcase // case(sq_sp_gpr_phase_mux)
     end // always@ (…
```

(*Id.* at 5:18-6:7.)

215. The `macc_gpr` module writes the selected input into the memory “`ugpr_mem`” at a specified address, as shown using the R400 RTL code below:

```
// WRITE INTERFACE
       .CLKA(iSCLK), // Write Clock
       .WEA(gpr_wr_ena), // Write enable
       .MEA(vdd), // Memory enable
       .ADRA0(gpr_wr_addr[0]), .ADRA1(gpr_wr_addr[1]),
.ADRA2(gpr_wr_addr[2]), .ADRA3(gpr_wr_addr[3]),  // Write Address
       .ADRA4(gpr_wr_addr[4]), .ADRA5(gpr_wr_addr[5]),
.ADRA6(gpr_wr_addr[6]),  // Write Address
       .DA0(InputGPR[0]), .DA1(InputGPR[1]), .DA2(InputGPR[2]),
.DA3(InputGPR[3]),  // Write Data
       .DA4(InputGPR[4]), .DA5(InputGPR[5]), .DA6(InputGPR[6]),
.DA7(InputGPR[7]),  // Write Data
       .DA8(InputGPR[8]), .DA9(InputGPR[9]), .DA10(InputGPR[10]),
.DA11(InputGPR[11]),  // Write Data
       .DA12(InputGPR[12]), .DA13(InputGPR[13]),
.DA14(InputGPR[14]), .DA15(InputGPR[15]),  // Write Data
       .DA16(InputGPR[16]), .DA17(InputGPR[17]),
.DA18(InputGPR[18]), .DA19(InputGPR[19]),  // Write Data
       .DA20(InputGPR[20]), .DA21(InputGPR[21]),
.DA22(InputGPR[22]), .DA23(InputGPR[23]),  // Write Data
       .DA24(InputGPR[24]), .DA25(InputGPR[25]),
.DA26(InputGPR[26]), .DA27(InputGPR[27]),  // Write Data
       .DA28(InputGPR[28]), .DA29(InputGPR[29]),
.DA30(InputGPR[30]), .DA31(InputGPR[31]),  // Write Data
       .DA32(InputGPR[32]), .DA33(InputGPR[33]),
.DA34(InputGPR[34]), .DA35(InputGPR[35]),  // Write Data
       .DA36(InputGPR[36]), .DA37(InputGPR[37]),
.DA38(InputGPR[38]), .DA39(InputGPR[39]),  // Write Data
       .DA40(InputGPR[40]), .DA41(InputGPR[41]),
.DA42(InputGPR[42]), .DA43(InputGPR[43]),  // Write Data
       .DA44(InputGPR[44]), .DA45(InputGPR[45]),
.DA46(InputGPR[46]), .DA47(InputGPR[47]),  // Write Data
       .DA48(InputGPR[48]), .DA49(InputGPR[49]),
.DA50(InputGPR[50]), .DA51(InputGPR[51]),  // Write Data
       .DA52(InputGPR[52]), .DA53(InputGPR[53]),
.DA54(InputGPR[54]), .DA55(InputGPR[55]),  // Write Data
       .DA56(InputGPR[56]), .DA57(InputGPR[57]),
.DA58(InputGPR[58]), .DA59(InputGPR[59]),  // Write Data
       .DA60(InputGPR[60]), .DA61(InputGPR[61]),
.DA62(InputGPR[62]), .DA63(InputGPR[63]),  // Write Data
       .DA64(InputGPR[64]), .DA65(InputGPR[65]),
.DA66(InputGPR[66]), .DA67(InputGPR[67]),  // Write Data
       .DA68(InputGPR[68]), .DA69(InputGPR[69]),
.DA70(InputGPR[70]), .DA71(InputGPR[71]),  // Write Data
       .DA72(InputGPR[72]), .DA73(InputGPR[73]),
.DA74(InputGPR[74]), .DA75(InputGPR[75]),  // Write Data
       .DA76(InputGPR[76]), .DA77(InputGPR[77]),
.DA78(InputGPR[78]), .DA79(InputGPR[79]),  // Write Data
       .DA80(InputGPR[80]), .DA81(InputGPR[81]),
.DA82(InputGPR[82]), .DA83(InputGPR[83]),  // Write Data
       .DA84(InputGPR[84]), .DA85(InputGPR[85]),
.DA86(InputGPR[86]), .DA87(InputGPR[87]),  // Write Data
```

```
      .DA88(InputGPR[88]), .DA89(InputGPR[89]),
.DA90(InputGPR[90]), .DA91(InputGPR[91]),  // Write Data
      .DA92(InputGPR[92]), .DA93(InputGPR[93]),
.DA94(InputGPR[94]), .DA95(InputGPR[95]),  // Write Data
      .DA96(InputGPR[96]), .DA97(InputGPR[97]),
.DA98(InputGPR[98]), .DA99(InputGPR[99]),  // Write Data
      .DA100(InputGPR[100]), .DA101(InputGPR[101]),
.DA102(InputGPR[102]), .DA103(InputGPR[103]),  // Write Data
      .DA104(InputGPR[104]), .DA105(InputGPR[105]),
.DA106(InputGPR[106]), .DA107(InputGPR[107]),  // Write Data
      .DA108(InputGPR[108]), .DA109(InputGPR[109]),
.DA110(InputGPR[110]), .DA111(InputGPR[111]),  // Write Data
      .DA112(InputGPR[112]), .DA113(InputGPR[113]),
.DA114(InputGPR[114]), .DA115(InputGPR[115]),  // Write Data
      .DA116(InputGPR[116]), .DA117(InputGPR[117]),
.DA118(InputGPR[118]), .DA119(InputGPR[119]),  // Write Data
      .DA120(InputGPR[120]), .DA121(InputGPR[121]),
.DA122(InputGPR[122]), .DA123(InputGPR[123]),  // Write Data
      .DA124(InputGPR[124]), .DA125(InputGPR[125]),
.DA126(InputGPR[126]), .DA127(InputGPR[127]),  // Write Data
```

(*Id.* at 10:13-12:20.)

216. The `macc_gpr` module also retrieves the selected input from the "`ugpr_mem`" memory. For example, upon request, the `macc_gpr` module may read the selected input from memory an store the selected input in the `RegData` register, as shown using R400 RTL code below:

```
      .CLKB(iSCLK), // Read Clock
      .OEB(sq_sp_gpr_rd_ena), // Output enable
      .MEB(vdd), // Read enable
      .ADRB0(sq_sp_gpr_rd_addr[0]), .ADRB1(sq_sp_gpr_rd_addr[1]),
.ADRB2(sq_sp_gpr_rd_addr[2]), .ADRB3(sq_sp_gpr_rd_addr[3]),  //
Read Address
      .ADRB4(sq_sp_gpr_rd_addr[4]), .ADRB5(sq_sp_gpr_rd_addr[5]),
.ADRB6(sq_sp_gpr_rd_addr[6]),  // Read Address
      .QB0(RegData[0]), .QB1(RegData[1]), .QB2(RegData[2]),
.QB3(RegData[3]),  // Read Data
      .QB4(RegData[4]), .QB5(RegData[5]), .QB6(RegData[6]),
.QB7(RegData[7]),  // Read Data
      .QB8(RegData[8]), .QB9(RegData[9]), .QB10(RegData[10]),
.QB11(RegData[11]),  // Read Data
```

```
      .QB12(RegData[12]), .QB13(RegData[13]), .QB14(RegData[14]),
.QB15(RegData[15]),  // Read Data
      .QB16(RegData[16]), .QB17(RegData[17]), .QB18(RegData[18]),
.QB19(RegData[19]),  // Read Data
      .QB20(RegData[20]), .QB21(RegData[21]), .QB22(RegData[22]),
.QB23(RegData[23]),  // Read Data
      .QB24(RegData[24]), .QB25(RegData[25]), .QB26(RegData[26]),
.QB27(RegData[27]),  // Read Data
      .QB28(RegData[28]), .QB29(RegData[29]), .QB30(RegData[30]),
.QB31(RegData[31]),  // Read Data
      .QB32(RegData[32]), .QB33(RegData[33]), .QB34(RegData[34]),
.QB35(RegData[35]),  // Read Data
      .QB36(RegData[36]), .QB37(RegData[37]), .QB38(RegData[38]),
.QB39(RegData[39]),  // Read Data
      .QB40(RegData[40]), .QB41(RegData[41]), .QB42(RegData[42]),
.QB43(RegData[43]),  // Read Data
      .QB44(RegData[44]), .QB45(RegData[45]), .QB46(RegData[46]),
.QB47(RegData[47]),  // Read Data
      .QB48(RegData[48]), .QB49(RegData[49]), .QB50(RegData[50]),
.QB51(RegData[51]),  // Read Data
      .QB52(RegData[52]), .QB53(RegData[53]), .QB54(RegData[54]),
.QB55(RegData[55]),  // Read Data
      .QB56(RegData[56]), .QB57(RegData[57]), .QB58(RegData[58]),
.QB59(RegData[59]),  // Read Data
      .QB60(RegData[60]), .QB61(RegData[61]), .QB62(RegData[62]),
.QB63(RegData[63]),  // Read Data
      .QB64(RegData[64]), .QB65(RegData[65]), .QB66(RegData[66]),
.QB67(RegData[67]),  // Read Data
      .QB68(RegData[68]), .QB69(RegData[69]), .QB70(RegData[70]),
.QB71(RegData[71]),  // Read Data
      .QB72(RegData[72]), .QB73(RegData[73]), .QB74(RegData[74]),
.QB75(RegData[75]),  // Read Data
      .QB76(RegData[76]), .QB77(RegData[77]), .QB78(RegData[78]),
.QB79(RegData[79]),  // Read Data
      .QB80(RegData[80]), .QB81(RegData[81]), .QB82(RegData[82]),
.QB83(RegData[83]),  // Read Data
      .QB84(RegData[84]), .QB85(RegData[85]), .QB86(RegData[86]),
.QB87(RegData[87]),  // Read Data
      .QB88(RegData[88]), .QB89(RegData[89]), .QB90(RegData[90]),
.QB91(RegData[91]),  // Read Data
      .QB92(RegData[92]), .QB93(RegData[93]), .QB94(RegData[94]),
.QB95(RegData[95]),  // Read Data
      .QB96(RegData[96]), .QB97(RegData[97]), .QB98(RegData[98]),
.QB99(RegData[99]),  // Read Data
      .QB100(RegData[100]), .QB101(RegData[101]),
.QB102(RegData[102]), .QB103(RegData[103]),  // Read Data
      .QB104(RegData[104]), .QB105(RegData[105]),
.QB106(RegData[106]), .QB107(RegData[107]),  // Read Data
      .QB108(RegData[108]), .QB109(RegData[109]),
.QB110(RegData[110]), .QB111(RegData[111]),  // Read Data
      .QB112(RegData[112]), .QB113(RegData[113]),
.QB114(RegData[114]), .QB115(RegData[115]),  // Read Data
```

```
      .QB116(RegData[116]), .QB117(RegData[117]),
.QB118(RegData[118]), .QB119(RegData[119]),  // Read Data
      .QB120(RegData[120]), .QB121(RegData[121]),
.QB122(RegData[122]), .QB123(RegData[123]),  // Read Data
      .QB124(RegData[124]), .QB125(RegData[125]),
.QB126(RegData[126]), .QB127(RegData[127]),  // Read Data
```

(*Id.* at 8:5-10:12.)

217. The storage and retrieval of the selected input to the "`ugpr_mem`" memory allows the `macc_gpr` module to maintain the selected one of the plurality of inputs as recited in claim 13.

218. Each vector unit `uvector0-3` includes four instances of a `macc_gpr` module. The vector units are included in the `sp` module, within the shader. As such, the shader includes a register block.

2. The computation element

219. Claim 13 also recites "a computation element operative to perform arithmetic and logical operations on the data maintained in the register block." Each instance of the `macc_gpr` module includes a MACC module called umacc, which is replicated using the R400 RTL code below:

```
macc umacc(.oResult(VectorResult), .oScalarOpcode(oScalarOpcode)
,.oScalarInput(oScalarInput),.oExportDst(oexport_dst),
.iRegData(q_RegData),.iConstantData(sq_sp_constant),.iScalarData(
iScalarData), .iInstruction(sq_sp_instruct),
.iInstStart(sq_sp_instruct_start), .sclk(sclk), .srst(srst));
```

(*Id.* at 3:17-21.)

220. The *MACC* module receives *q_RegData* (which it converts to *iRegData*) which is the data maintained in the register block in *macc_gpr* module as *oRegData*. The *iRegData* signal is converted to *OperandAMod*, *OperandBMod*, and/or *OperandCMod* as shown in the *MACC* module at 13:8-23:22.

221. The *MACC* module also receives the instruction from the sequencer using the *sq_sp_instruct* parameter and converts *sq_sp_instruct* to *iInstruction*. The *MACC* module then parses *iInstruction* and retrieves *VectorOpcode*, as shown using the R400 RTL code below:

```
reg [20:0]        q_Instruction0, q_Instruction1, q_Instruction2,
q_Instruction3;
…
wire [4:0]        VectorOpcode;
…
   //-----------------------------------------------------------------
-------------
   //-----------------------------------------------------------------
------------
   //Registering the Instruction word (20 bits) in four
consecutive cycles
   //-----------------------------------------------------------------
----------
   always@(posedge sclk)
     if(srst)
       q_Instruction0 <= 21'b0;
     else if(decode_SrcA)
       q_Instruction0 <= iInstruction;

   always@(posedge sclk)
     if(srst)
       q_Instruction1 <= 21'b0;
     else if(decode_SrcB)
       q_Instruction1 <= iInstruction;

    always@(posedge sclk)
    if(srst)
      q_Instruction2 <= 21'b0;
    else if(decode_SrcC)
      q_Instruction2 <= iInstruction;

  always@(posedge sclk)
    if(srst)
      q_Instruction3 <= 21'b0;
    else if(decode_Opcode)
      q_Instruction3 <= iInstruction;
…
  //---------------------------------------------------------------
------------
  //decoding the instruction word into a set of select/modify
signals used
  //for argument selection and input modification on the way to
MACC unit
  //---------------------------------------------------------------
-----------

  assign SrcASel = q_Instruction0[2:0];
  assign SrcANegate = q_Instruction0[3:3];
  assign SrcAAlphaSwizzle = q_Instruction0[11:10];
  assign SrcARedSwizzle = q_Instruction0[5:4];
  assign SrcAGreenSwizzle = q_Instruction0[7:6];
  assign SrcABlueSwizzle = q_Instruction0[9:8];

  assign SrcBSel = q_Instruction1[2:0];
  assign SrcBNegate = q_Instruction1[3:3];
  assign SrcBAlphaSwizzle = q_Instruction1[11:10];
  assign SrcBRedSwizzle = q_Instruction1[5:4];
  assign SrcBGreenSwizzle = q_Instruction1[7:6];
  assign SrcBBlueSwizzle = q_Instruction1[9:8];


  assign SrcCSel = q_Instruction2[2:0];
  assign SrcCNegate = q_Instruction2[3:3];
  assign SrcCAlphaSwizzle = q_Instruction2[11:10];
  assign SrcCRedSwizzle = q_Instruction2[5:4];
  assign SrcCGreenSwizzle = q_Instruction2[7:6];
  assign SrcCBlueSwizzle = q_Instruction2[9:8];



  assign VectorOpcode = q_Instruction3[4:0];
  assign ScalarOpcode = q_Instruction3[10:5];
  assign VectorClamp = q_Instruction3[11:11];
  assign ScalarClamp = q_Instruction3[12:12];
  assign VectorWriteMask = q_Instruction3[16:13];
  assign ScalarWriteMask = q_Instruction3[20:17];
```

(`macc.v`, 4:8, 4:10, 10:12-11:13, 11:22-13:6.)

222. In particular, the `MACC` module retrieves `VectorOpcode`, which includes instructions used to perform arithmetic and logical operations on the data maintained in the register block (the `oRegData`).

223. The computation element is included in the `MACC` module, and is called a `mad` unit. The `mad` unit is instantiated in the `macc32` module. The R400 RTL code that instantiates the `mad` unit in the `MACC` module is replicated below:

```
//Floating point Multiply and Accumulate
   macc32 mad(OperandAMod, OperandBMod, OperandCMod,
VectorOpcode,MaccResult,sclk);
```

(*Id.* at 24:25-25:2.)

224. The `macc32` module receives `OperandAMod`, `OperandBMod`, and `OperandCMod` as operands which include data maintained in the register block (the `oRegData`), and `VectorOpcode` which includes instructions that manipulate the data. The `macc32` module is then operative to use `OperandAMod`, `OperandBMod`, `OperandCMod` (data) and `VectorOpcode` (instructions) to perform 1) floating point operations which are arithmetic operations, and 2) minimum, maximum, and compare operations, which are logical operations as specified in *macc32.mc.* The `MACC` module, receives the results from `macc32` module as `MaccResult`.

225. Also, the MACC module includes a listing of additional operations that macc32 module is operative to perform, that may be included in the floating point operations. These operations as listed using R400 RTL code are replicated below:

```
//ALU opcodes declared as parameters
   //this definition is subject to change as more
   //opcodes are added. for the latest definition
   //please refer to Shader Pipe Spec: ALU instruction definition
   parameter [4:0] ADD = 5'h00,
               MUL = 5'h01,
               MAX = 5'h02,
               MIN = 5'h03,
               SETE = 5'h04,
               SETGT = 5'h05,
               SETGE = 5'h06,
               SETNE = 5'h07,
               FRACT = 5'h08,
               TRUNC = 5'h09,
               FLOOR = 5'h0a,
               MULADD = 5'h0b,
               CNDE = 5'h0c,
               CNDGE = 5'h0d,
               CNDGT = 5'h0e;
```

(*Id.* at 3:11-4:4.)

226. This listing of arithmetical opcodes further indicates that the *macc32* module is operative to perform arithmetic and logical operations on the data in the register block.

227. In this way, the R400 RTL code demonstrates how a computation element is operative to perform arithmetic and logical operation on the data maintained in the register block.

3. *The Sequencer*

228. Claim 13 also recites a "*sequencer for maintaining the instructions that are executed by the computation element*." The file, `sq.v`, instantiates a `sq` module which is a hardware block of the graphics processor component which includes a sequencer. The sequencer (the `sq` module) maintains instructions and provides the instructions to the shader (included in the `sp` module) using an `SQ_SP_instruct` parameter, as described in detail below. (`sq.v`, 2:17, 9:11, 80:11.)

229. The sequencer maintains instructions in the instruction store. The instruction store is instantiated as `sq_instruction_store` using the `sq_instruction_store` module (*Id.* at 86:23-88:2). The `sq_instruction_store` module is defined in the `sq_instruction_store.v`. It consists of 4096 instruction words which are each 96-bits wide.

230. The `u0_sq_alu_instr_fetch` and `u1_sq_alu_instr_fetch` units defined in `sq_target_instr_fetch` module retrieve the instruction from the `sq_instruction_store` module. For simplicity, I focus on the `u0_sq_alu_instr_fetch` unit. For example, the

`sq_target_instr_fetch` module includes an interface with the `sq_instruction_store` module, as replicated below:

```
   // instruction store interface
   is_read_addr,// instruction store read address
   is_read_data,// instruction store read data
   is_phase,         // instruction store phase
   alu_phase,        // alu phase (alu0 and alu1 share the alu
is_phase)
```

(`sq_target_instr_fetch.v`, 3:7-11.)

231. The `sq_target_instr_fetch` module uses the `is_read_addr` interface to send the instruction pointer that communicates the address of the instruction to the `sq_instruction_store` module, using the R400 RTL code below:

```
output [11:0]   is_read_addr;
```

(*Id.* at 5:13.)

```
assign is_read_addr = tip_q;
```

(*Id.* at 8:8.)

```
always @(posedge clk)
     begin
      if ( ld_tip )             tip_q <= cfs_instr_ptr;
      else if ( inc_tip )
         if ( vtx_wrap )        tip_q <= inst_base_vtx;
         else if ( pix_wrap )   tip_q <= inst_base_pix;
         else                   tip_q <= tip_q + 1;
      else                      tip_q <= tip_q;
     end
```

(Id. at 9:3-11.)

232. The `q_instruction_store` receives an `is_read_addr` request, from `u0_sq_alu_instr_fetch`, using the interface below:

```
input [11:0] i_alu0_addr;
```

(sq_instruction_store.v, 2:20.)

233. In response to the `is_read_addr` request, the `q_instruction_store` module retrieves the instruction and outputs the instruction as the `o_is_data` signal, using the R400 RTL code below:

```
output [95:0] o_is_data;
```

(*Id.* at 3:2.)

```
wire [95:0]  o_is_data = read_data;
```

(*Id.* at 3:19.)

```
assign        mem_read_data = d_addr[11] ? mem1_rd_data :
mem0_rd_data;
```

(*Id.* at 7:25.)

```
// register instantiation
always @(posedge i_clk)
  begin
    if (i_reset)
      begin
     we              <= 1'b0;
//   addr            <= 12'd0;
     read_data       <= 96'd0;
     o_rtr           <= 1'b0;
     wrt_data        <= 96'd0;
     q_rbi_addr_in   <= 12'd0;
      end
    else
      begin
     we              <= d_we;
//   addr            <= d_addr;
     read_data       <= mem_read_data;
     o_rtr           <= d_rtr;
     wrt_data        <= d_wrt_data;
     q_rbi_addr_in   <= d_rbi_addr_in;
      end
  end
```

(*Id.* at 15:26-16:19.)

234. The `sq_target_instr_fetch` module receives the instruction from the `q_instruction_store` module using the R400 RTL code below:

```
   input  [95:0] is_read_data;
```

(*sq_target_instr_fetch.v*, 5:14.)

235. The instruction is loaded into the `sq_target_instr_fetch` module's `tif_instr_q` register, as shown using the R400 RTL code below:

```
   // ------------------------------------
   // -- Target Instruction Register (TIR) --
   // ------------------------------------
   // - loaded with data read from instruction store
   // - the TIR is output to the target instruction queue (which
does some decode in front of the queue)

   always @(posedge clk)
     begin
      if (ld_tir)  tif_instr_q <= is_read_data;
      else         tif_instr_q <= tif_instr_q;
     end
```

(*Id.* at 12:17-13:2.)

236. The `sq_target_instr_fetch` module transmits the instruction to an `sq_alu_instr_queue` module using the interface below:

```
// outputs to the target instruction decoder (in the TIQ module)
…
   tif_thread_type_q, // vert:1, pix:0
   tif_thread_id_q,   // the target thread id
   tif_instr_q,       // the target instruction register (TIR)
```

(*Id.* at 3:19-21.)

237. With respect to the shader, the `sq_target_instr_fetch` module passes the instruction to the `sq_alu_instr_queue` module. The

sq_alu_instr_queue module calculates the gpr address (the address where the data is located that requires execution). The R400 RTL code for the *sq_alu_instr_queue* module is included in *sq_alu_instr_queue*.v. The *sq.v* file instantiates two instances of *sq_alu_instr_queue* module called *u0_sq_alu_instr_queue* and *u1_sq_alu_instr_queue*, associated with each instance of a shader. (*sq.v*, 68:6-69:24.) For simplicity, I address only *u0_sq_alu_instr_queue* as that is sufficient to meet the claim limitations.

238. The *sq_alu_instr_queue* module receives the instruction from the *sq_target_instr_fetch* module, using the interface below:

```
// inputs from AIF (ALU Instruction Fetch)
…
   aif_thread_type_q, // vector type (0: pixel, 1: vertex)
   aif_thread_id_q,   // thread id
…
   aif_instr_q,       // instruction register (registered read
from IS - 96 bits)
```

(*sq_alu_instr_queue*.v, 2:15, 2:21-22, 2:24.)

239. The *sq_alu_instr_queue* modules pass the instruction to the *u_sq_ais_output* module. The R400 RTL code for the *u_sq_ais_output* module is included in sq_ais_output.v. The *sq.v* instantiates an instance of the *u_sq_ais_output* module called *u_sq_ais_output*. (*sq.v*, 77:20-81:4.)

240. The *u_sq_ais_output* module receives the instruction from the *u0_sq_alu_instr_queue* using the interface below:

```
// inputs from the AIQs
…
   aiq0_instr,        // instruction
```

(*sq_ais_output.v*, 2:9, 2:14.)

241. The *u_sq_ais_output* module converts the *aiq0_instr* instruction into a *SQ_SP* interface format, and then provides the instruction to the shader, as shown using the R400 RTL code below:

```
   // --------------------------------
   // -- SP instruction, write_mask --
   // --------------------------------
   // - valid with instruction start

   always @(posedge clk)
    begin
      case (gpr_phase)
       `SQ_SRCB_PHASE: begin
         case (alu_phase)
        LO: begin
             SQ_SP_instr <= {3'b000, aiq0_instr[06:00],
aiq0_instr[55:48], aiq0_instr[58], aiq0_instr[101:99]};
             u0_SQ_SP_write_mask <= aiq0_valid_bits  [3:0];
u1_SQ_SP_write_mask <= aiq0_valid_bits  [7:4];
             u2_SQ_SP_write_mask <= aiq0_valid_bits [11:8];
u3_SQ_SP_write_mask <= aiq0_valid_bits [15:12];
           end
      …
          endcase
        end
       `SQ_SRCC_PHASE: begin
         case (alu_phase)
        LO: begin
             SQ_SP_instr <= {aiq0_instr[15:08], aiq0_instr[47:40],
aiq0_instr[57], aiq0_instr[98:96]};
             u0_SQ_SP_write_mask <= aiq0_valid_bits [19:16];
u1_SQ_SP_write_mask <= aiq0_valid_bits [23:20];
```

```
              u2_SQ_SP_write_mask <= aiq0_valid_bits [27:24];
u3_SQ_SP_write_mask <= aiq0_valid_bits [31:28];
            end
      …
           endcase
         end
         `SQ_FA_PHASE: begin
           case (alu_phase)
          LO: begin
              SQ_SP_instr <= {aiq0_instr[23:16], aiq0_instr[39:32],
aiq0_instr[56], aiq0_instr[95:93]};
              u0_SQ_SP_write_mask <= aiq0_valid_bits [35:32];
u1_SQ_SP_write_mask <= aiq0_valid_bits [39:36];
              u2_SQ_SP_write_mask <= aiq0_valid_bits [43:40];
u3_SQ_SP_write_mask <= aiq0_valid_bits [47:44];
            end
      …
           endcase
         end
         `SQ_SRCA_PHASE: begin
           case (alu_phase)
          LO: begin
              SQ_SP_instr <= {aiq0_instr[23:16], aiq0_instr[25:24],
aiq0_instr[31:26], aiq0_instr[92:88]};
              u0_SQ_SP_write_mask <= aiq0_valid_bits [51:48];
u1_SQ_SP_write_mask <= aiq0_valid_bits [55:52];
              u2_SQ_SP_write_mask <= aiq0_valid_bits [59:56];
u3_SQ_SP_write_mask <= aiq0_valid_bits [63:60];
            end
           endcase
         end
        endcase
     end
```

(*Id.* at 16:9-17:20, 17:29-18:11, 18:20-19:15 .)

242. The *SQ_SP* interface which includes instruction that the sequencer passed tothe shader is replicated below:

```
     // outputs to SP
     SQ_SP_gpr_wr_addr,
     SQ_SP_gpr_wr_en,
     SQ_SP_gpr_rd_addr,
     SQ_SP_gpr_rd_en,
     SQ_SP_gpr_phase,
     SQ_SP_gpr_input_sel,
     SQ_SP_gpr_channel_mask,
…
   SQ_SP_instr,
   SQ_SP_const,
…
   //
   SQ_SP_exporting,
   SQ_SP_exp_id,
   u0_SQ_SP_write_mask,
   u1_SQ_SP_write_mask,
   u2_SQ_SP_write_mask,
   u3_SQ_SP_write_mask,
```

(*Id.* at 4:4-11, 4:15-16, 4:18-24.)

243. In particular, the interface includes the *SQ_SP_instruct* parameter which provides the instruction.

244. The shader (included in the *sp* module) receives the instruction using the *SQ_SP* interface, and converts the instruction into *q_sq_instruct*, as shown using the R400 RTL code below:

```
input [20:0]   SQ_SP_instruct;
```

(*sp.v*, 6:13.)

```
ati_dff_in #(21) sq_instruct(sclk,SQ_SP_instruct,q_sq_instruct);
```

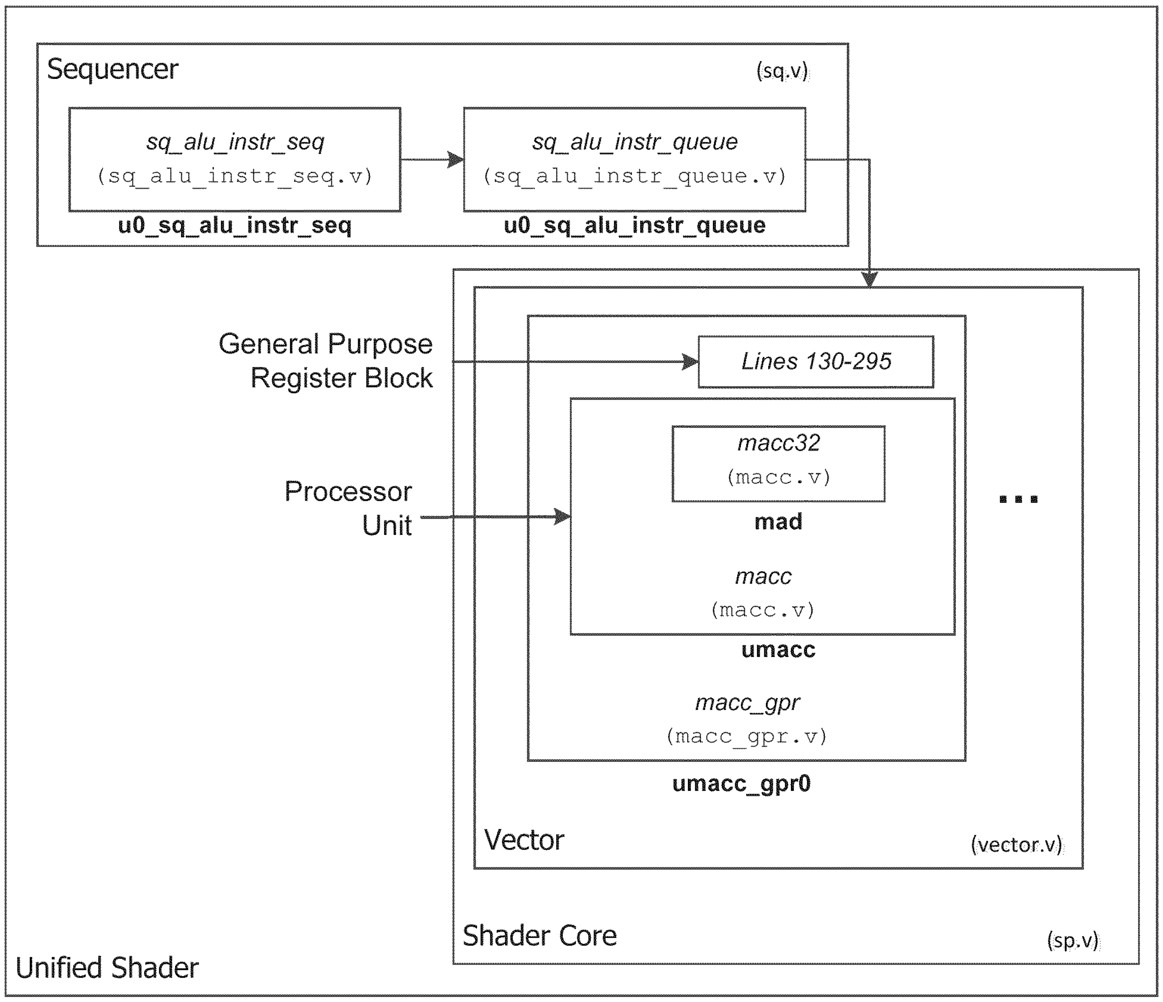
(*Id.* at 7:4.)

245. The *sp* module passes the instruction to the vector units *uvector0*, *uvector1*, *uvector2*, and *uvector3*, which pass the instruction to the *macc_gpr* module, the *MACC* module and to the *mad* unit (which is the computation unit) as described in Sections IX.A.3 and IX.J.3.

246. In this way, the instruction maintained by the sequencer is executed by the computation element.

K. Claim 15

247. Claim 15 recites “[a] unified shader” comprising three components: a general purpose register, a processor unit and a sequencer. Below, I have generated a figure based on my understanding of the R400 RTL code that shows the relationship between these components.



1. A general purpose register block

248. Claim 15 recites a unified shader comprising "*a general purpose register block for maintaining data*." As I discussed in Section IX.A.3, the shader instantiates `umacc_gpr0`, `umacc_gpr1`, `umacc_gpr2`, and `umacc_gpr3` units using the `macc_gpr` module. The `macc_gpr` module is specified in

macc_gpr.v. One or more of the *umacc_gpr0*, *umacc_gpr1*, *umacc_gpr2*, and *umacc_gpr3* form a register block.

249. The register block is a general purpose register block because it stores different types of data. For example, the data can be vector data (*VectorResult*), scalar data (*iScalarData*), texture data (*tp_sp_data*) or interpolated data (*iInterpolated*). As shown, *macc_gpr* module selects the type of data for storage in the general purpose register, using the R400 RTL code below:

```
//------------------------------------------------------------------
------------------------------------------------------------/
   //The phase mux controlling the write input port into GPRs
(register file write port)
   //---------------------------------------------------------------
----------------------------------------------------------------/
   always@(/*AUTOSENSE*/VectorResult or iInterpolated or
iScalarData
        or sq_sp_gpr_phase_mux or tp_sp_data)
     begin
      case(sq_sp_gpr_phase_mux)
        2'b00: InputGPR = iInterpolated;
        2'b01: InputGPR = tp_sp_data;
        2'b10: InputGPR = VectorResult;
        2'b11: InputGPR = iScalarData;
        default: InputGPR = iInterpolated;
      endcase // case(sq_sp_gpr_phase_mux)
     end // always@ (...
```

(*macc_gpr.v*, 5:18-6:7.)

250. The *macc_gpr* module then writes the selected data using the *InputGPR* signal into a memory called *ugpr_mem*, as shown in Section IX.A.3.

251. Additionally, the `macc_gpr` module reads the data from `ugpr_mem` using the `RegData` signal as shown in Section IX. A. The reading and writing to a `ugpr_mem` memory maintains the data in the general purpose register block as recited in claim 13.

2. The processor unit

252. Claim 13 also recites "*a processor unit.*" The *MACC* module and the macc32 module, which I described in Section IX.A.3, together form a processor unit.

3. The Sequencer

253. Claim 13 also recites "*a sequencer, coupled to the general purpose register block and the processor unit.*" I described in Section IX.J.3 that a sequencer is the `sq` module.

a. Coupled to the general purpose register

254. The sequencer is coupled to the general purpose register block and the processor unit. In particular, the sequencer is coupled to the processor unit by the `SQ_SP` interface, and the general purpose register block by the `SQ_SP_GPR` interface. The sequencer side of the interface, included in the *sq* module, is replicated using the R400 RTL code below:

```
//-----------------------------------------------------------
```

```
// SQ-SP GPR control Interface
//--------------------------------------------------------------
output [6:0] SQ_SP_gpr_wr_addr;
output [0:0] u0_SQ_SP_gpr_wr_en;
output [0:0] u1_SQ_SP_gpr_wr_en;
output [0:0] u2_SQ_SP_gpr_wr_en;
output [0:0] u3_SQ_SP_gpr_wr_en;
output [6:0] SQ_SP_gpr_rd_addr;
output [0:0] SQ_SP_gpr_rd_en;
output [1:0] SQ_SP_gpr_phase_mux;
output [3:0] SQ_SP_channel_mask;

output [3:0] u0_SQ_SP_pix_mask;
output [3:0] u1_SQ_SP_pix_mask;
output [3:0] u2_SQ_SP_pix_mask;
output [3:0] u3_SQ_SP_pix_mask;

output [1:0]  Sq_sp_gpr_input_mux;
output [11:0] SQ_SP_auto_count;
```

(*sq.v*, 8:11-9:5.)

255. The *sp* module also includes the *SQ_SP* interface, which receives the above parameters using the R400 RTL code, replicated below:

```
//-----------------------------------------------------------/
   //SEQUENCER(SQ)-SHADER(SP)
   //GPR control and auto-counter interface
   //-----------------------------------------------------------/
   input [6:0]     SQ_SP_gpr_wr_addr;
   input [6:0]     SQ_SP_gpr_rd_addr;
   input [0:0]     SQ_SP_gpr_rd_en,SQ_SP_gpr_wr_en;
//these to read/write enable signals

//are used to enable the TP - GPR write path also
   input [1:0]     SQ_SP_gpr_phase_mux;
//control into GPR write port
   input [3:0]     SQ_SP_channel_mask;
   input [3:0]     SQ_SP_pix_mask;
   input [1:0]     Sq_sp_gpr_input_mux;
   input [11:0]   SQ_SP_auto_count;


   wire [6:0]      q_sq_gpr_wr_addr;
   wire [6:0]      q_sq_gpr_rd_addr;
   wire [0:0]      q_sq_gpr_rd_en,q_sq_gpr_wr_en;
//these to read/write enable signals
```

```
//are used to enable the TP - GPR write path also
   wire [1:0]      q_sq_gpr_phase_mux;
//control into GPR write port
   wire [3:0]      q_sq_channel_mask;
   wire [3:0]      q_sq_pix_mask;
   wire [1:0]      q_sq_gpr_input_mux;
   wire [11:0]     q_sq_auto_count;


   ati_dff_in #(7)
sq_gpr_wr_addr(sclk,SQ_SP_gpr_wr_addr,q_sq_gpr_wr_addr);
   ati_dff_in #(7)
sq_gpr_rd_addr(sclk,SQ_SP_gpr_rd_addr,q_sq_gpr_rd_addr);
   ati_dff_in #(1)
sq_gpr_rd_en(sclk,SQ_SP_gpr_rd_en,q_sq_gpr_rd_en);
   ati_dff_in #(1)
sq_gpr_wr_en(sclk,SQ_SP_gpr_wr_en,q_sq_gpr_wr_en);
   ati_dff_in #(2)
sq_gpr_phase_mux(sclk,SQ_SP_gpr_phase_mux,q_sq_gpr_phase_mux);
   ati_dff_in #(4)
sq_channel_mask(sclk,SQ_SP_channel_mask,q_sq_channel_mask);
   ati_dff_in #(4)
sq_pix_mask(sclk,SQ_SP_pix_mask,q_sq_pix_mask);
   ati_dff_in #(2)
sq_gpr_input_mux(sclk,Sq_sp_gpr_input_mux,q_sq_gpr_input_mux
);
   ati_dff_in #(12)
sq_auto_count(sclk,SQ_SP_auto_count,q_sq_auto_count);
```

(*sp.v*, 8:24-10:8)

256. The *sp* module provides the above `SQ_SP_GPR` interface to vector units: *uvector0*, *uvector1*, *uvector2*, and *uvector3*. (*See sp.v*, 15:6-18:16.) Each of the vector units includes the general purpose registers called *umacc_gpr0*, *umacc_gpr1*, *umacc_gpr2*, and *umacc_gpr3*, as I described in Section IX.A.3.

257. As such, the sequencer is coupled to the general purpose register block.

258. Because the general purpose register block is included in the processor unit, the sequencer is also coupled to the processor unit. Additionally, as I discussed in Section IX.A.3, the sequencer is also coupled to the processor unit through the `SQ_SP` instruction interface.

b. The sequencer maintains instructions

259. Claim 13 also recites "the sequencer maintaining instructions operative to cause the processor unit to execute vertex calculation and pixel calculation operations on selected data maintained in the general purpose register block."

260. As I discussed in Section IX.J.3, the sequencer maintains instructions. As I also discussed in Section IX.A.3, these instructions include pixel manipulation instructions and vertex manipulation instruction. When the vector unit passes the instruction to the `MACC` module, the `MACC` module executes the vertex calculation or the pixel calculation (as shown in macc.v and macc32.mc), depending on whether the instruction includes vertex manipulations or pixel manipulations.

261. The instruction also performs the operations on the selected data maintained in the general purpose register. As discussed in Section IX.K.1, the general purpose register maintains data, and the selected data is read from the `ugpr_mem` memory using the `RegData` signal.

262. The data selected in the *RegData* is stored as *q_RegData* and *oRegData*. (*macc_gpr.v*, 15:13, 15:17.) The *q_RegData* and instruction (*q_sp_instruct*) are passed to the *MACC* module called *umacc* that performs operations as described in macc.v, and generates a *VectorResult* that includes the result of the operations. (*macc_gpr.v*, 3:17-21.)

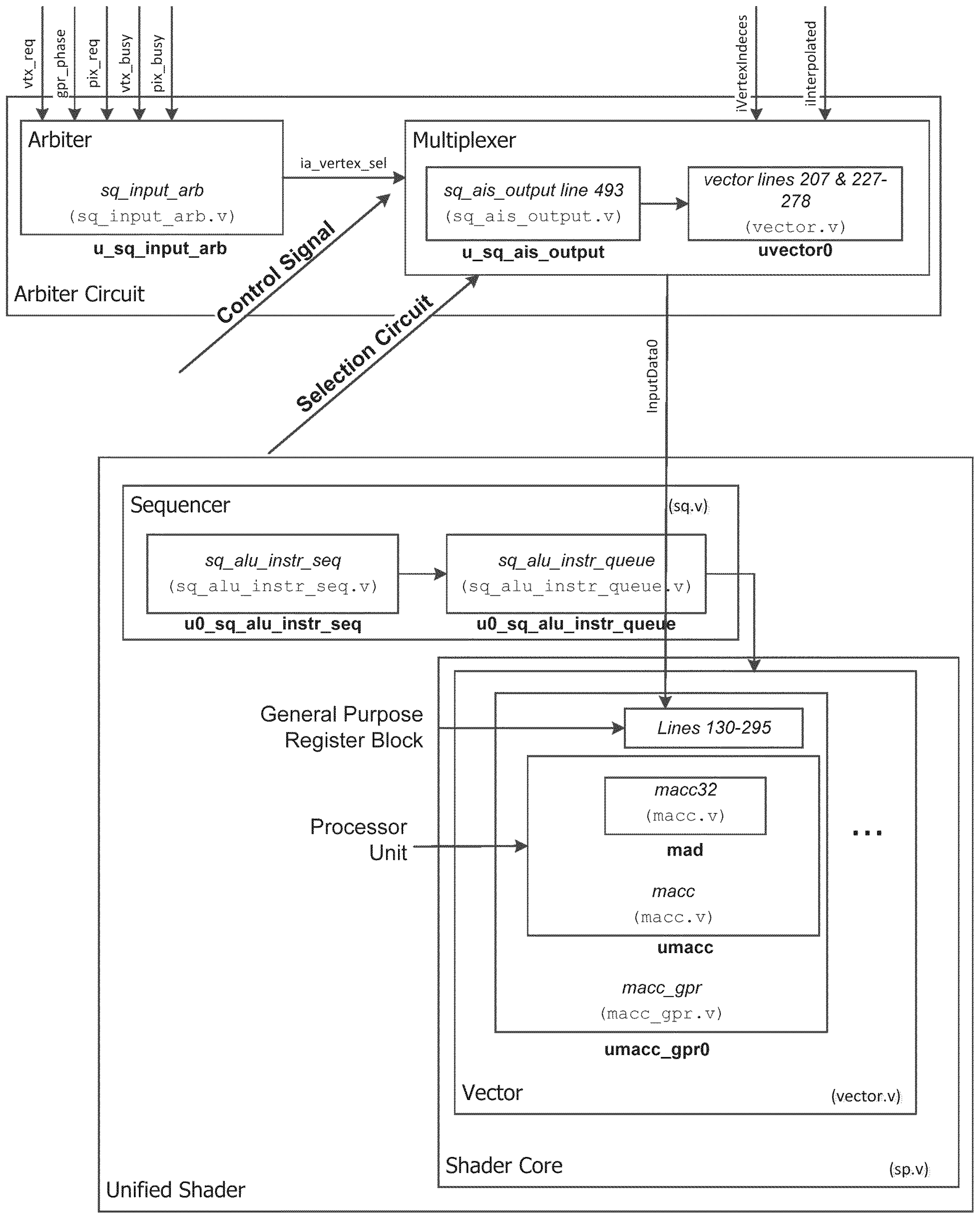
263. As such the instructions cause the processor unit to execute vertex and pixel calculations operations on the selected data maintained in the general purpose register block.

L. Claim 17

264. Claim 17 recites "a selection circuit operative to provide information to the general purpose block in response to a control signal."

265. I already discussed a selection circuit in Section IX.A. I have also discussed how the selection circuit provides an *SQ_SP_gpr_input_sel* signal to the *sp* module as the *sq_sp_gpr_input_mux* signal in response to a control signal. And I have described in Section IX.K.1 that the *sp* module includes the general purpose register block implemented as *umacc_gpr0*, *umacc_gpr1*, *umacc_gpr2*, and *umacc_gpr3* and that these blocks process data based on whether the *sq_sp_gpr_input_mux* signal is set to process the vertex

operations or the pixel operations. In this way, the selection circuit is operative to provide information to the general purpose block in response to a control signal.


vtx_req
gpr_phase
pix_req
vtx_busy
pix_busy
iVertexIndeces
iInterpolated
Arbiter
sq_input_arb
(sq_input_arb.v)
u_sq_input_arb
ia_vertex_sel
Multiplexer
sq_ais_output line 493
(sq_ais_output.v)
u_sq_ais_output
vector lines 207 & 227-278
(vector.v)
uvector0
Arbiter Circuit
Control Signal
Selection Circuit
InputData0
Sequencer
(sq.v)
sq_alu_instr_seq
(sq_alu_instr_seq.v)
u0_sq_alu_instr_seq
sq_alu_instr_queue
(sq_alu_instr_queue.v)
u0_sq_alu_instr_queue
General Purpose Register Block
Lines 130-295
Processor Unit
macc32
(macc.v)
mad
macc
(macc.v)
umacc
macc_gpr
(macc_gpr.v)
umacc_gpr0
...
Vector
(vector.v)
Shader Core
(sp.v)
Unified Shader

M. Claim 18

266. Claim 18 recites a shader of claim 17, "*wherein the selection circuit is a multiplexer and the control signal is provided by an arbiter.*" I already discussed a selection circuit that is a multiplexer in Section IX.A.2. And I have discussed a control signal that is provided by the arbiter in Section IX.A.2.

N. Claim 20

267. Claim 20 recites a shader of claim 15, "*wherein the processor unit executes vertex calculations while the pixel calculations are still in progress.*" I already discussed that the processor unit executes vertex calculations and pixel calculations. As I describe in Section IX.K.1, the general purpose register block described in `macc_gpr.v` can store the `InputGPR` data which can be either pixel data or vertex data. The general purpose register block in `macc_gpr` does not differentiate between the different data, and can store the pixel and vertex data at the same time.

268. A POSA would understand that in this architecture, the pixel calculations can stall while still in progress when the pixel threads wait for texture data. In this case, vertex calculations begin to execute, while the pixel calculations are stalled, but are still in progress – i.e., the pixel calculations still have pixel data that requires processing.

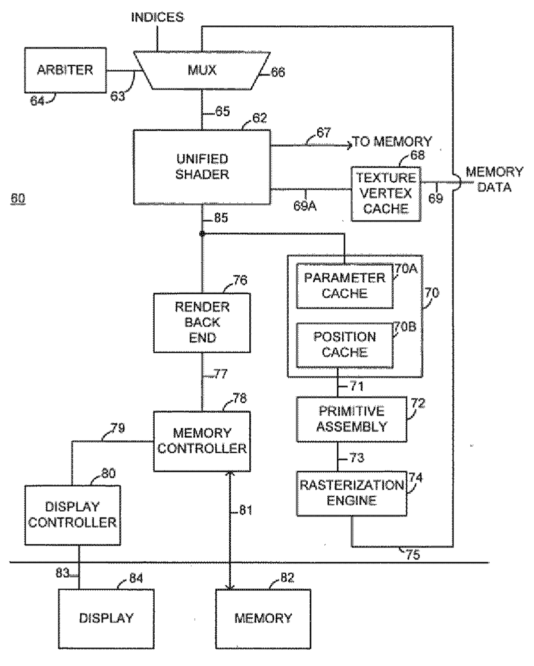
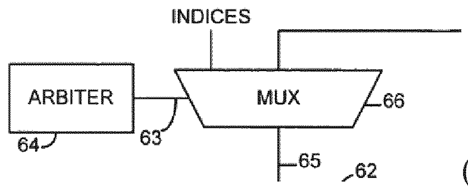
269. In this way, the R400 RTL code discloses the processor unit that executes vertex calculations while the pixel calculations are still in progress.

X. THE CLAIMS OF THE '871 PATENT ARE SUPPORTED BY THE PRIORITY DOCUMENT

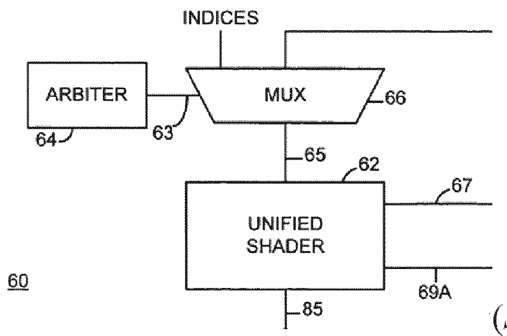
270. I understand that a specification must contain written description of the invention. I also understand that the purpose of this requirement is to satisfy the inventor's obligation to disclose to the public the technologic knowledge upon which the patent is based and also to demonstrate that the inventor was in possession of the claimed invention.

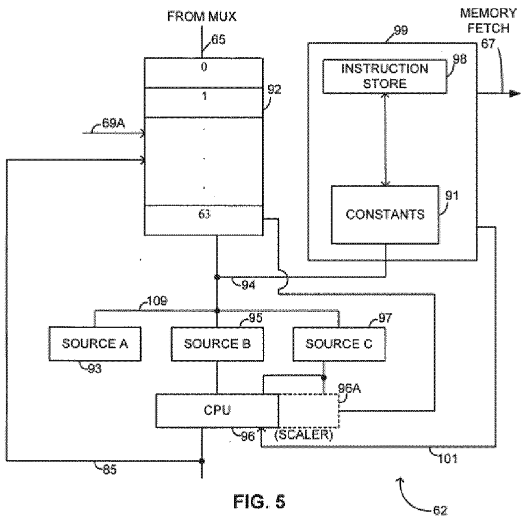
271. The '871 patent was filed on November 20, 2003 as the '318 application. I have examined the specification and figures of the '318 application. Based on my examination of the '318 application, I have generated a claim chart which demonstrates that the '318 application has written description support for all the instituted claims.

Support for the '871 Patent Claims in U.S. Patent Application No. 10/718,318	
'871 Patent Claim	
1. A graphics processor, comprising:	"The present invention generally relates to graphics processors and, more particularly, to a graphics processor architecture employing a single shader." (Ex. 2076, ¶ 1.) "FIG. 4A is a schematic block diagram of a graphics processor architecture according to the present

	invention." (*Id.* at 3.)  INDICES ARBITER MUX UNIFIED SHADER TO MEMORY TEXTURE VERTEX CACHE MEMORY DATA PARAMETER CACHE POSITION CACHE PRIMITIVE ASSEMBLY RASTERIZATION ENGINE RENDER BACK END MEMORY CONTROLLER DISPLAY CONTROLLER DISPLAY MEMORY 60 62 63 64 65 66 67 68 69 69A 70 70A 70B 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 FIG. 4A (*Id.* at FIG. 4A.) "Briefly stated, the present invention is directed to a graphics processor that employs a unified shader." (*Id.* at 4.) "A graphics processing architecture employing a single shader is disclosed." (*Id.* at 18.)
1a. an arbiter circuit for selecting one of a plurality of inputs in response to a control signal; and	"The architecture includes a circuit operative to select one of a plurality of inputs in response to a control signal." (*Id.*)  (*See id.* at FIG. 4A.) "[V]ertex information . . . is coupled to the first input of multiplexer 66." (*Id.* at 11.) "The resulting pixel data from the rasterization engine

	block 74 is the interpolated pixel parameter data that is transmitted to the second input of the multiplexer 66 on line 75." (*Id.* at 10.) "In an exemplary embodiment, a graphics processor according to the present invention includes an arbiter circuit for selecting one of a plurality of inputs for processing in response to a control signal." (*Id.* at 4.) "Referring now to FIG. 4A, in an exemplary embodiment, the graphics processor 60 of the present invention includes a multiplexer 66 having vertex (e.g. indices) data provided at a first input thereto and interpolated pixel parameter (e.g. position) data and attribute data from a rasterization engine 74 provided at a second input. A control signal generated by an arbiter 64 is transmitted to the multiplexer 66 on line 63. The arbiter 64 determines which of the two inputs to the multiplexer 66 is transmitted to a unified shader 62 for further processing. The arbitration scheme employed by the arbiter 64 is as follows: the vertex data on the first input of the multiplexer 66 is transmitted to the unified shader 62 on line 65 if there is enough resources available in the unified shader to operate on the vertex data; otherwise, the interpolated pixel parameter data present on the second input will be passed to the unified shader 62 for further processing." (*Id.* at 7.)
1b. a shader, coupled to the arbiter circuit, operative to process the selected one of the plurality of inputs, the shader including means for performing vertex operations and pixel operations, and performing one of the vertex operations or pixel operations based	"The architecture includes . . . a shader, coupled to the arbiter, operative to process the selected one of the plurality of inputs, the shader including means for performing vertex operations and pixel operations, and wherein the shader performs one of the vertex operations or pixel operations based on the selected one of the plurality of inputs. The shader includes a register block which is used to store the plurality of selected inputs, a sequencer which maintains vertex manipulation and pixel manipulations instructions and a processor capable of executing both floating point arithmetic and logical operations on the selected inputs in response to

on the selected one of the plurality of inputs, wherein the shader provides a appearance attribute.	the instructions maintained in the sequencer." (*Id.* at 18.) "Briefly stated, the present invention is directed to a graphics processor that employs a unified shader that is capable of performing both the vertex operations and the pixel operations in a space saving and computationally efficient manner." (*Id.* at 4.) "In an exemplary embodiment, a graphics processor according to the present invention includes . . . a shader, coupled to the arbiter, operative to process the selected one of the plurality of inputs, the shader including means for performing vertex operations and pixel operations, and wherein the shader performs one of the vertex operations or pixel operations based on the selected one of the plurality of inputs." (*Id.*)  INDICES ARBITER MUX 66 64 63 65 62 67 UNIFIED SHADER 60 69A 85 (*See id.* at FIG. 4A.) "FIG. 5 is an exploded schematic block diagram of the unified shader employed in the graphics processor illustrated in FIG. 4A." (*Id.* at 4.)

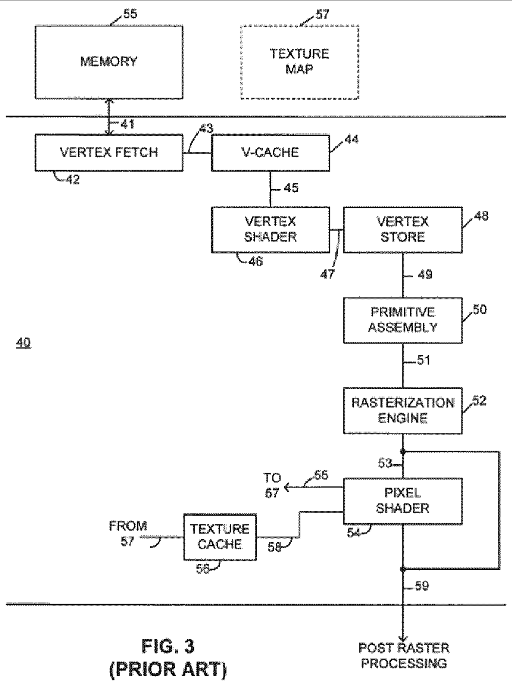
	 FROM MUX 65 0 1 63 92 69A MEMORY FETCH 99 67 98 INSTRUCTION STORE 91 CONSTANTS 94 109 95 97 SOURCE A SOURCE B SOURCE C 93 96A CPU 96 (SCALER) 101 85 62 FIG. 5 (*Id.* at FIG. 5.) "The shader includes a general purpose register block for storing at least the plurality of selected inputs, a sequencer for storing logical and arithmetic instructions that are used to perform vertex and pixel manipulation operations and a processor capable of executing both floating point arithmetic and logical operations on the selected inputs according to the instructions maintained in the sequencer. The shader of the present invention is referred to as a 'unified' shader because it is configured to perform both vertex and pixel operations." (*Id.* at 4.) "[T]he unified shader is more computationally efficient because it allows the shader to be flexibly allocated to pixels or verticies based on workload." (*Id.* at 5.) "[A]s illustrated [in FIG. 5], the unified shader 62 includes a general purpose register block 92, a plurality of source registers: including source register A 93, source register B 95, and source register C 97, a processor (e.g. CPU) 96 and a sequencer 99. The general purpose register block 92 includes sixty four registers, or available entries, for storing the information transmitted from the multiplexer 66 on line 65 or any other information to be maintained within the unified shader. The data present in the general purpose register block 92 is transmitted to the plurality of source

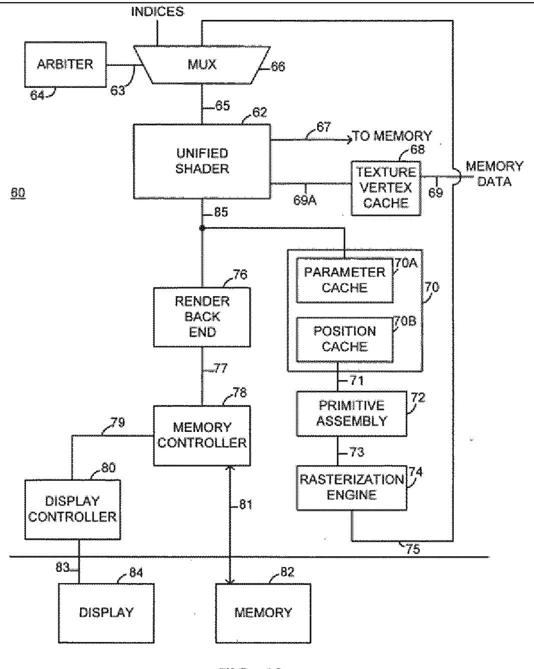
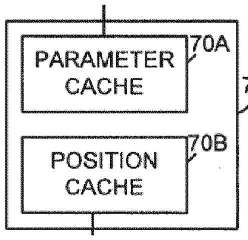
	registers via line 109." (*Id.* at 7.) "The processor 96 may be comprised of a dedicated piece of hardware or can be configured as part of a general purpose computing device (i.e. personal computer). In an exemplary embodiment, the processor 96 is adapted to perform 32-bit floating point arithmetic operations as well as a complete series of logical operations on corresponding operands. As shown, the processor is logically partitioned into two sections. Section 96 is configured to execute, for example, the 32-bit floating point arithmetic operations of the unified shader. The second section, 96A, is configured to perform scaler operations (e.g. log, exponent, reciprocal square root) of the unified shader." (*Id.* at 8.) "The sequencer 99 includes constants block 91 and an instruction store 98. The constants block 91 contains, for example, the several transformation matrices used in connection with vertex manipulation operations. The instruction store 98 contains the necessary instructions that are executed by the processor 96 in order to perform the respective arithmetic and logic operations on the data maintained in the general purpose register block 92 as provided by the source registers 93-95. The instruction store 98 further includes memory fetch instructions that, when executed, causes the unified shader 62 to fetch texture and other types of data, from memory 82 (FIG. 4A). In operation, the sequencer 99 determines whether the next instruction to be executed (from the instruction store 98) is an arithmetic or logical instruction or a memory (e.g. texture fetch) instruction. If the next instruction is a memory instruction or request, the sequencer 99 sends the request to a fetch block (not shown) which retrieves the required information from memory 82 (FIG. 4A). The retrieved information is then transmitted to the sequencer 99, through the vertex texture cache 68 (FIG. 4A)." (*Id.*) "If the next instruction to be executed is an arithmetic or

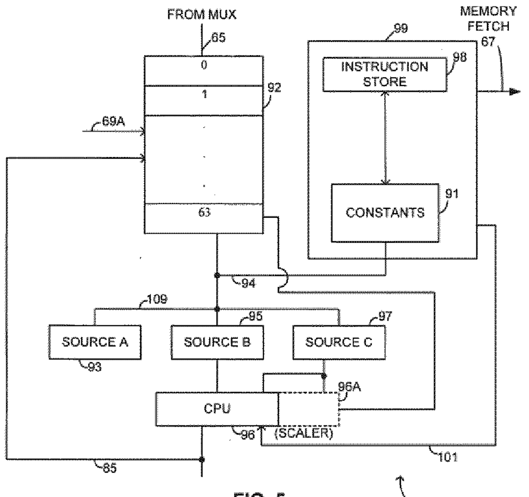
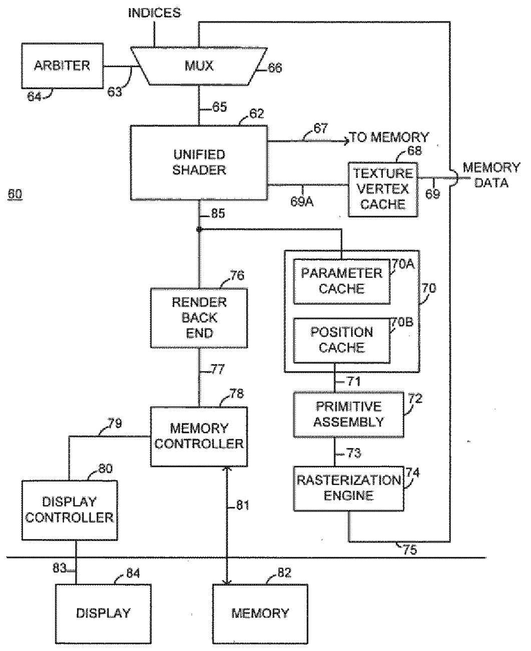
	logical instruction, the sequencer 99 causes the appropriate operands to be transferred from the general purpose register block 92 into the appropriate source registers (93, 95, 97) for execution, and an appropriate signal is sent to the processor 96 on line 101 indicating what operation or series of operations are to be executed on the several operands present in the source registers. At this point, the processor 96 executes the instructions on the operands present in the source registers and provides the result on line 85. The information present on line 85 may be transmitted back to the general purpose register block 92 for storage, or transmitted to succeeding components of the graphics processor 60." (*Id.* at 9.) "[T]he instruction store 98 maintains both vertex manipulation instructions and pixel manipulation instructions. Therefore, the unified shader 99 of the present invention is able to perform both vertex and pixel operations, as well as execute memory fetch operations. As such, the unified shader 62 of the present invention is able to perform both the vertex shading and pixel shading operations on data in the context of a graphics controller based on information passed from the multiplexer. By being adapted to perform memory fetches, the unified shader of the present invention is able to perform additional processes that conventional vertex shaders cannot perform; while at the same time, perform pixel operations." (*Id.*) "The unified shader 62 has ability to simultaneously perform vertex manipulation operations and pixel manipulation operations at various degrees of completion by being able to freely switch between such programs or instructions, maintained in the instruction store 98, very quickly. In application, vertex data to be processed is transmitted into the general purpose register block 92 from multiplexer 66. The instruction store 98 then passes the corresponding control signals to the processor 96 on line 101 to perform such vertex

	operations. However, if the general purpose register block 92 does not have enough available space therein to store the incoming vertex data, such information will not be transmitted as the arbitration scheme of the arbiter 64 is not satisfied. In this manner, any pixel calculation operations that are to be, or are currently being, performed by the processor 96 are continued, based on the instructions maintained in the instruction store 98, until enough registers within the general purpose register block 92 become available. Thus, through the sharing of resources within the unified shader 62, processing of image data is enhanced as there is no down time associated with the processor 96." (*Id.*) "[T]he graphics processor 60 of the present invention incorporates a unified shader 62 which is capable of performing both vertex manipulation operations and pixel manipulation operations based on the instructions stored in the instruction store 98." (*Id.* at 11.) "[A] as the unified shader 62 is capable of alternating between performing vertex manipulation operations and pixel manipulation operations, graphics processing efficiency is enhanced as one type of data operations is not dependent upon another type of data operations." (*Id.* at 11-12.) "[A] conventional shader 10 can be represented as a processing block 12 that accepts a plurality of bits of input data, such as, for example, object shape data (14) in object space (x,y,z); material properties of the object, such as color (16); texture information (18); luminance information (20); and viewing angle information (22) and provides output data (28) representing the object with texture and other appearance properties applied thereto (x', y', z')." (*Id.* at 1.) "[V]ertex shader . . . accepts as inputs the data representing, for example, vertices Vx, Vv and Vz, among others of cube 30 and providing angularly oriented vertices Vx,Vv and Vz, including any

	appearance attributes of corresponding cube 30'." (*Id.* at 2.) "[T]he pixel shader 54 generates the color and additional appearance attributes that are to be applied to a given pixel, and applies the appearance attributes to the respective pixels . . . . The generated pixel appearance attribute is then combined with a base color, as provided by the rasterization engine on line 53, to thereby provide a pixel color to the pixel corresponding at the position of interest. The pixel appearance attribute present on line 59 is then transmitted to post raster processing blocks (not shown)." (*Id.* at 6.)
2. The graphics processor of claim 1, further including a vertex storage block for maintaining vertex information.	"[Referring to FIG. 3,] [a]fter performing the transformation operation, the data representing the transformed vertices are then provided to a vertex store 48 on line 47. The vertex store 48 then transmits the modified vertex information contained therein to a primitive assembly block 50 on line 49. The primitive assembly block 50 assembles, or converts, the input vertex information into a plurality of primitives to be subsequently processed. Suitable methods of assembling the input vertex information into primitives [are] known in the art and will not be discussed in greater detail here. The assembled primitives are then transmitted to a rasterization engine 52, which converts the previously assembled primitives into pixel data through a process referred to as walking. The resulting pixel data is then transmitted to a pixel shader 54 on line 53." (*Id.* at 6.)

	 55 MEMORY 57 TEXTURE MAP 41 43 VERTEX FETCH V-CACHE 44 42 45 VERTEX SHADER VERTEX STORE 48 46 47 49 PRIMITIVE ASSEMBLY 50 40 51 RASTERIZATION ENGINE 52 53 TO 57 55 PIXEL SHADER FROM 57 TEXTURE CACHE 54 58 56 59 POST RASTER PROCESSING **FIG. 3 (PRIOR ART)** (*Id.* at FIG. 3.) "Referring back to FIG. 4A, the graphics processor 60 further includes a cache block 70, including a parameter cache 70A and a position cache 70B which accepts the [vertex] based output of the unified shader 62 on line 85 and stores the respective [vertex] parameter and position information in the corresponding cache. The [vertex] information present in the cache block 70 is then transmitted to the primitive assembly block 72 on line 71. The primitive assembly block 72 is responsible for assembling the information transmitted thereto from the cache block 70 into a series of triangles, or other suitable primitives, for further processing. The assembled primitives are then transmitted on line 73 to rasterization engine block 74, where the transmitted primitives are then converted into individual pixel data information through a walking process, or any other suitable pixel generation process. The resulting pixel data from the rasterization engine block 74 is the interpolated pixel parameter data that is transmitted to the second input of the multiplexer 66 on line 75." (*Id.* at 10.)

	 INDICES ARBITER MUX UNIFIED SHADER TO MEMORY TEXTURE VERTEX CACHE MEMORY DATA PARAMETER CACHE POSITION CACHE RENDER BACK END PRIMITIVE ASSEMBLY MEMORY CONTROLLER RASTERIZATION ENGINE DISPLAY CONTROLLER DISPLAY MEMORY 60 62 63 64 65 66 67 68 69 69A 70 70A 70B 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 FIG. 4A (*Id.* at FIG. 4A.)
3. The graphics processor of claim 2, wherein the vertex storage block further includes a parameter cache operative to maintain appearance attribute data for a corresponding vertex and a position cache operative to maintain position data for a corresponding vertex.	"Referring back to FIG. 4A, the graphics processor 60 further includes a cache block 70, including a parameter cache 70A and a position cache 70B which accepts the [vertex] based output of the unified shader 62 on line 85 and stores the respective [vertex] parameter and position information in the corresponding cache." (*Id.* at 10.)  (*See id.* at FIG. 4A.)
5. The graphics processor of claim 1, wherein the appearance attribute is position, and the position attribute is associated with a corresponding vertex when the selected one of	"[Refering to FIG. 5,] vertex data to be processed is transmitted into the general purpose register block 92 from multiplexer 66. The instruction store 98 then passes the corresponding control signals to the processor 96 on line 101 to perform such vertex operations." (*Id.* at 9.)

the plurality of inputs is vertex data.	 FROM MUX 65 MEMORY FETCH 67 99 98 INSTRUCTION STORE 0 1 92 69A 63 CONSTANTS 91 94 109 95 97 SOURCE A SOURCE B SOURCE C 93 96A CPU 96 (SCALER) 101 85 62 FIG. 5 (*Id.* at FIG. 5.) "Referring back to FIG. 4A, the graphics processor 60 further includes a cache block 70, including a parameter cache 70A and a position cache 70B which accepts the [vertex] based output of the unified shader 62 on line 85 and stores the respective [vertex] parameter and position information in the corresponding cache." (*Id.* at 10.)  INDICES ARBITER MUX 66 64 63 65 62 UNIFIED SHADER 67 TO MEMORY 68 TEXTURE VERTEX CACHE MEMORY DATA 69 69A 60 85 76 RENDER BACK END PARAMETER CACHE 70A 70 POSITION CACHE 70B 71 77 78 PRIMITIVE ASSEMBLY 72 79 MEMORY CONTROLLER 73 80 RASTERIZATION ENGINE 74 DISPLAY CONTROLLER 81 75 83 84 82 DISPLAY MEMORY FIG. 4A (*Id.* at FIG. 4A.)

"FIG. 3 is a schematic block diagram of a conventional graphics processor architecture." (*Id.* at 3.)

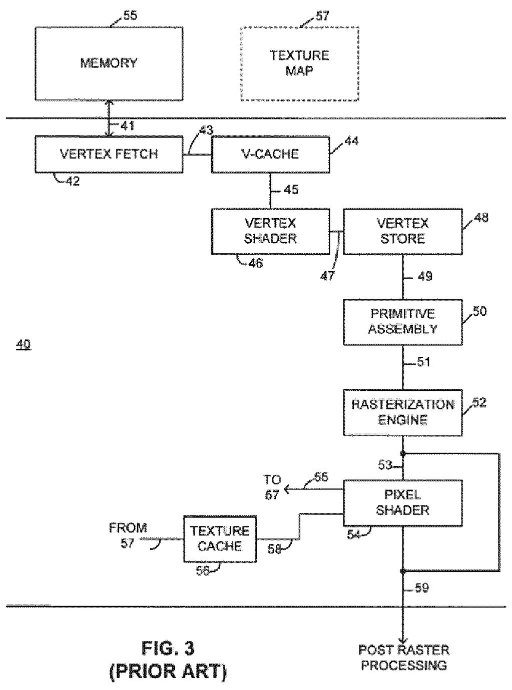


FIG. 3
(PRIOR ART)

(*Id.* at FIG. 3.)

"[In reference to FIG. 3,] the vertex data maintained in the vertex cache 44 is transmitted to a vertex shader 46 on line 45. [A]n example of the information that is requested by and transmitted to the vertex shader 46 includes the object shape, material properties (e.g. color), texture information, and viewing angle. Generally, the vertex shader 46 is a programmable mechanism which applies a transformation position matrix to the input position information (obtained from the vertex cache 44), thereby providing data representing a perspectively corrected image of the object to be rendered, along with any texture or color coordinates thereof." (*Id.* at 5.)

"[A] conventional shader 10 can be represented as a processing block 12 that accepts a plurality of bits of input data, such as, for example, object shape data (14) in object space (x,y,z); material properties of the object, such as color (16); texture information (18); luminance

	information (20); and viewing angle information (22) and provides output data (28) representing the object with texture and other appearance properties applied thereto (x', y', z')." (*Id.* at 1.) "[V]ertex shader . . . accepts as inputs the data representing, for example, vertices Vx, Vv and Vz, among others of cube 30 and providing angularly oriented vertices Vx,Vv and Vz, including any appearance attributes of corresponding cube 30'." (*Id.* at 2.)
6. The graphics processor of claim 5, wherein the appearance attribute is color, and the color attribute is associated with a corresponding pixel when the selected one of the plurality of inputs is pixel data.	"[A]ny pixel calculation operations that are to be, or are currently being, performed by the processor 96 are continued, based on the instructions maintained in the instruction store 98." (*Id.* at 10.) "In those situations when [pixel] data is transmitted to the unified shader 62 through the multiplexer 66, the resulting [pixel] data generated by the processor 96, is transmitted to a render back end block 76 which converts the resulting [pixel] data into at least one of several formats suitable for later display on display device 84. For example, if a stained glass appearance effect is to be applied to an image, the information corresponding to such appearance effect is associated with the appropriate position data by the render back end 76." (*Id.* at 11.)

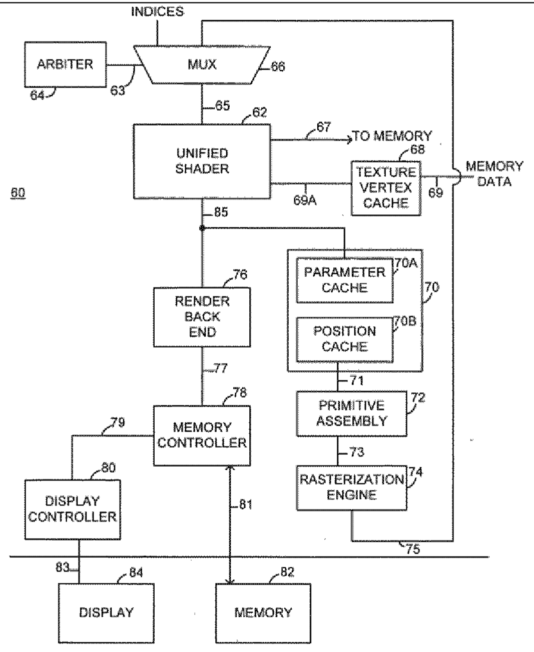


FIG. 4A

(*Id.* at FIG. 4A.)

"FIG. 3 is a schematic block diagram of a conventional graphics processor architecture." (*Id.* at 3.)

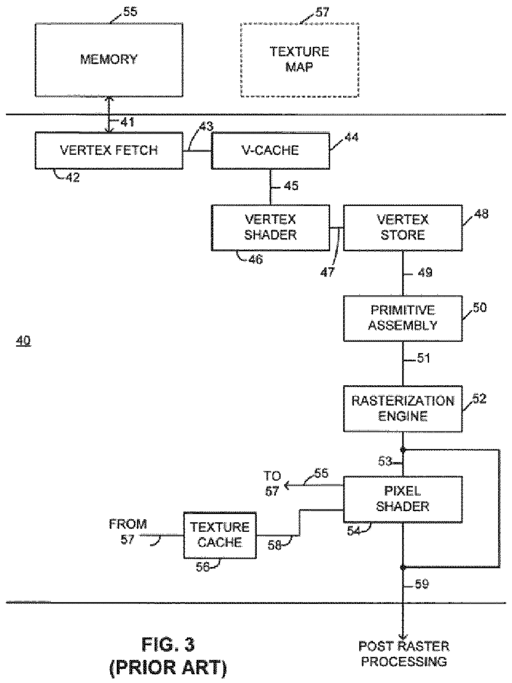


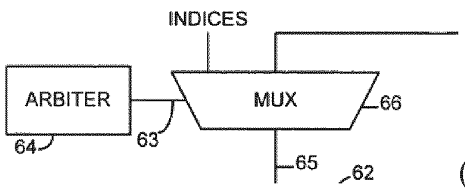
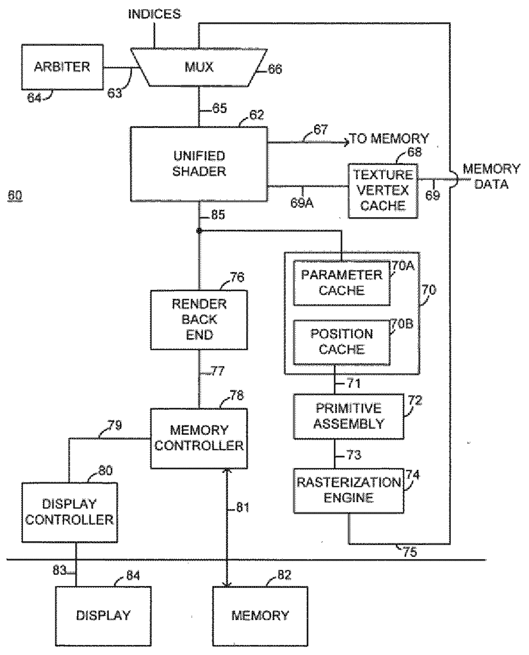
FIG. 3 (PRIOR ART)

(*Id.* at FIG. 3.)

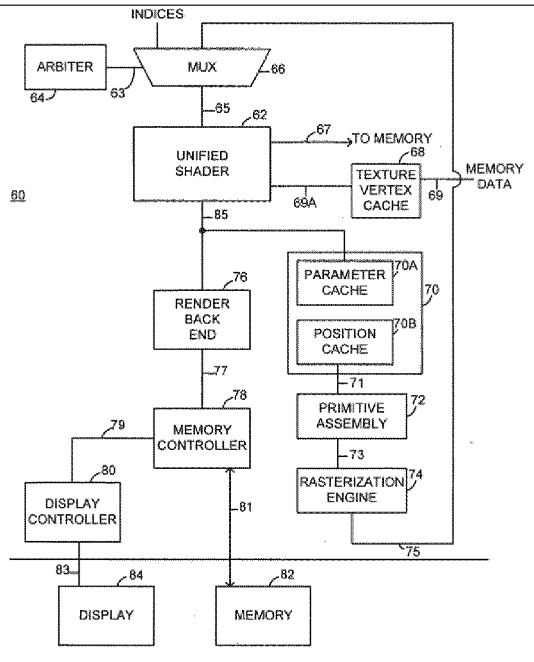
"[In reference to FIG. 3,] [c]olor and texture are then

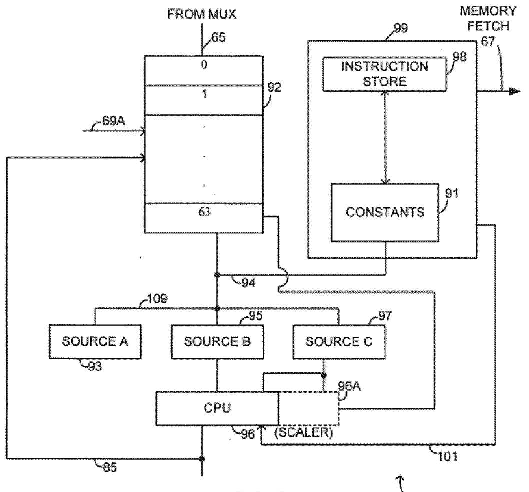
	applied to the individual pixels that comprise the shape based on their location within the primitive and the primitives orientation with respect to the generated shape; thereby generating the object that is rendered to a corresponding display for subsequent viewing." (*Id.* at 1.) "[A] conventional shader 10 can be represented as a processing block 12 that accepts a plurality of bits of input data, such as, for example, . . . material properties of the object, such as color (16) . . . and provides output data (28) representing the object with texture and other appearance properties applied thereto (x', y', z')." (*Id.*) "The pixel shader 54 generates the color and additional appearance attributes that are to be applied to a given pixel, and applies the appearance attributes to the respective pixels . . . . The generated pixel appearance attribute is then combined with a base color, as provided by the rasterization engine on line 53, to thereby provide a pixel color to the pixel corresponding at the position of interest. The pixel appearance attribute present on line 59 is then transmitted to post raster processing blocks (not shown)." (*Id.* at 6.) "Generally, the pixel shader provides the color value associated with each pixel of a rendered object." (*Id.* at 2.)
8. The graphics processor of claim 1, wherein the appearance value is depth.	"[A] conventional shader 10 can be represented as a processing block 12 that accepts a plurality of bits of input data, such as, for example, object shape data (14) in object space (x,y,z) . . . and provides output data (28) representing the object with texture and other appearance properties applied thereto (x', y', z')." (*Id.*) "[T]he shader accepts the vertex coordinate data representing cube 30 (FIG. 2A) as inputs and provides data representing, for example, a perspectively corrected view of the cube 30' (FIG. 2B) as an output. The corrected view may be provided, for example, by

	applying an appropriate transformation matrix to the data representing the initial cube 30. More specifically, the representation illustrated in FIG. 2B is provided by a vertex shader that accepts as inputs the data representing, for example, vertices Vx, Vv and Vz, among others of cube 30 and providing angularly oriented vertices Vx, Vv and Vz, including any appearance attributes of corresponding cube 30'." (*Id.*)
9. The graphics processor of claim 1, further including a selection circuit, wherein the selection circuit is a multiplexer, and the control signal is provided by an arbiter, wherein the arbiter is coupled to the multiplexer.	"The architecture includes a circuit operative to select one of a plurality of inputs in response to a control signal." (*Id.* at 18.) "In an exemplary embodiment, a graphics processor according to the present invention includes an arbiter circuit for selecting one of a plurality of inputs for processing in response to a control signal." (*Id.* at 4.) "[V]ertex information . . . is coupled to the first input of multiplexer 66." (*Id.* at 11.) "The resulting pixel data from the rasterization engine block 74 is the interpolated pixel parameter data that is transmitted to the second input of the multiplexer 66 on line 75." (*Id.* at 10.) "Referring now to FIG. 4A, in an exemplary embodiment, the graphics processor 60 of the present invention includes a multiplexer 66 having vertex (e.g. indices) data provided at a first input thereto and interpolated pixel parameter (e.g. position) data and attribute data from a rasterization engine 74 provided at a second input. A control signal generated by an arbiter 64 is transmitted to the multiplexer 66 on line 63. The arbiter 64 determines which of the two inputs to the multiplexer 66 is transmitted to a unified shader 62 for further processing. The arbitration scheme employed by the arbiter 64 is as follows: the vertex data on the first input of the multiplexer 66 is transmitted to the unified shader 62 on line 65 if there is enough resources available in the unified shader to operate on the vertex

	data; otherwise, the interpolated pixel parameter data present on the second input will be passed to the unified shader 62 for further processing." (*Id.* at 7.)  (*See id.* at FIG. 4A.)
10. The graphics processor of claim 1, wherein the shader provides vertex position data and further including a primitive assembly block, coupled to the shader, operative to generate primitives in response to the vertex position data.	"[T]he processor 96 executes the instructions on the operands present in the source registers and provides the result on line 85. The information present on line 85 may be . . . transmitted to succeeding components of the graphics processor 60." (*Id.* at 9.)  FIG. 4A (*Id.* at FIG. 4A.) "Referring back to FIG. 4A, the graphics processor 60 further includes a cache block 70, including a parameter cache 70A and a position cache 70B which accepts the [vertex] based output of the unified shader 62 on line 85 and stores the respective [vertex] parameter and position information in the corresponding cache. The [vertex]

	information present in the cache block 70 is then transmitted to the primitive assembly block 72 on line 71. The primitive assembly block 72 is responsible for assembling the information transmitted thereto from the cache block 70 into a series of triangles, or other suitable primitives, for further processing." (*Id.* at 10.)
11. The graphics processor of claim 10, further including a raster engine, coupled to the primitive assembly block, operative to generate pixel parameter data in response to the assembled vertex data.	"Referring back to FIG. 4A, the graphics processor 60 further includes a cache block 70, including a parameter cache 70A and a position cache 70B which accepts the pixel based output of the unified shader 62 on line 85 and stores the respective pixel parameter and position information in the corresponding cache. The pixel information present in the cache block 70 is then transmitted to the primitive assembly block 72 on line 71. The primitive assembly block 72 is responsible for assembling the information transmitted thereto from the cache block 70 into a series of triangles, or other suitable primitives, for further processing. The assembled primitives are then transmitted on line 73 to rasterization engine block 74, where the transmitted primitives are then converted into individual pixel data information through a walking process, or any other suitable pixel generation process. The resulting pixel data from the rasterization engine block 7 4 is the interpolated pixel parameter data that is transmitted to the second input of the multiplexer 66 on line 75." (*Id.*)

	 INDICES ARBITER MUX UNIFIED SHADER TO MEMORY TEXTURE VERTEX CACHE MEMORY DATA PARAMETER CACHE POSITION CACHE RENDER BACK END PRIMITIVE ASSEMBLY MEMORY CONTROLLER RASTERIZATION ENGINE DISPLAY CONTROLLER DISPLAY MEMORY 60 62 63 64 65 66 67 68 69 69A 70 70A 70B 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 FIG. 4A (*Id.* at FIG. 4A.)
13. The graphics processor of claim 1, wherein the shader includes a register block for maintaining the selected one of the plurality of inputs, a computation element operative to perform arithmetic and logical operations on the data maintained in the register block, and sequencer for maintaining the instructions that are executed by the computation element.	"The shader includes a register block which is used to store the plurality of selected inputs, a sequencer which maintains vertex manipulation and pixel manipulations instructions and a processor capable of executing both floating point arithmetic and logical operations on the selected inputs in response to the instructions maintained in the sequencer." (*Id.* at 18.) "FIG. 5 is an exploded schematic block diagram of the unified shader employed in the graphics processor illustrated in FIG. 4A." (*Id.* at 4.)

	 FROM MUX 65 MEMORY FETCH 67 99 98 INSTRUCTION STORE 0 1 92 69A 63 91 CONSTANTS 94 109 95 97 SOURCE A SOURCE B SOURCE C 93 96A CPU 96 (SCALER) 101 85 62 FIG. 5 (*Id.* at FIG. 5.) "The shader includes a general purpose register block for storing at least the plurality of selected inputs, a sequencer for storing logical and arithmetic instructions that are used to perform vertex and pixel manipulation operations and a processor capable of executing both floating point arithmetic and logical operations on the selected inputs according to the instructions maintained in the sequencer. The shader of the present invention is referred to as a 'unified' shader because it is configured to perform both vertex and pixel operations." (*Id.* at 4.) "[T]he unified shader 62 includes a general purpose register block 92, a plurality of source registers: including source register A 93, source register B 95, and source register C 97, a processor (e.g. CPU) 96 and a sequencer 99. The general purpose register block 92 includes sixty four registers, or available entries, for storing the information transmitted from the multiplexer 66 on line 65 or any other information to be maintained within the unified shader. The data present in the general purpose register block 92 is transmitted to the plurality of source registers via line 109." (*Id.* at 7.) "The processor 96 may be comprised of a dedicated piece of hardware or can be configured as part of a general purpose computing device (i.e. personal

	computer). In an exemplary embodiment, the processor 96 is adapted to perform 32-bit floating point arithmetic operations as well as a complete series of logical operations on corresponding operands. As shown, the processor is logically partitioned into two sections. Section 96 is configured to execute, for example, the 32-bit floating point arithmetic operations of the unified shader. The second section, 96A, is configured to perform scaler operations (e.g. log, exponent, reciprocal square root) of the unified shader." (*Id.* at 8.) "The sequencer 99 includes constants block 91 and an instruction store 98. The constants block 91 contains, for example, the several transformation matrices used in connection with vertex manipulation operations. The instruction store 98 contains the necessary instructions that are executed by the processor 96 in order to perform the respective arithmetic and logic operations on the data maintained in the general purpose register block 92 as provided by the source registers 93-95. The instruction store 98 further includes memory fetch instructions that, when executed, causes the unified shader 62 to fetch texture and other types of data, from memory 82 (FIG. 4A). In operation, the sequencer 99 determines whether the next instruction to be executed (from the instruction store 98) is an arithmetic or logical instruction or a memory (e.g. texture fetch) instruction. If the next instruction is a memory instruction or request, the sequencer 99 sends the request to a fetch block (not shown) which retrieves the required information from memory 82 (FIG. 4A). The retrieved information is then transmitted to the sequencer 99, through the vertex texture cache 68 (FIG. 4A) as described in greater detail below." (*Id.* at 8.) "If the next instruction to be executed is an arithmetic or logical instruction, the sequencer 99 causes the appropriate operands to be transferred from the general purpose register block 92 into the appropriate source registers (93, 95, 97) for execution, and an appropriate

	signal is sent to the processor 96 on line 101 indicating what operation or series of operations are to be executed on the several operands present in the source registers. At this point, the processor 96 executes the instructions on the operands present in the source registers and provides the result on line 85. The information present on line 85 may be transmitted back to the general purpose register block 92 for storage, or transmitted to succeeding components of the graphics processor 60." (*Id.* at 9.) "[T]he instruction store 98 maintains both vertex manipulation instructions and pixel manipulation instructions. Therefore, the unified shader 99 of the present invention is able to perform both vertex and pixel operations, as well as execute memory fetch operations. As such, the unified shader 62 of the present invention is able to perform both the vertex shading and pixel shading operations on data in the context of a graphics controller based on information passed from the multiplexer. By being adapted to perform memory fetches, the unified shader of the present invention is able to perform additional processes that conventional vertex shaders cannot perform; while at the same time, perform pixel operations." (*Id.*) "The unified shader 62 has ability to simultaneously perform vertex manipulation operations and pixel manipulation operations at various degrees of completion by being able to freely switch between such programs or instructions, maintained in the instruction store 98, very quickly. In application, vertex data to be processed is transmitted into the general purpose register block 92 from multiplexer 66. The instruction store 98 then passes the corresponding control signals to the processor 96 on line 101 to perform such vertex operations. However, if the general purpose register block 92 does not have enough available space therein to store the incoming vertex data, such information will not be transmitted as the arbitration scheme of the

	arbiter 64 is not satisfied. In this manner, any pixel calculation operations that are to be, or are currently being, performed by the processor 96 are continued, based on the instructions maintained in the instruction store 98, until enough registers within the general purpose register block 92 become available. Thus, through the sharing of resources within the unified shader 62, processing of image data is enhanced as there is no down time associated with the processor 96." (*Id.*) "[T]he graphics processor 60 of the present invention incorporates a unified shader 62 which is capable of performing both vertex manipulation operations and pixel manipulation operations based on the instructions stored in the instruction store 98. In this fashion, the graphics processor 60 of the present invention takes up less real estate than conventional graphics processors as separate vertex shaders and pixel shaders are no longer required. In addition, as the unified shader 62 is capable of alternating between performing vertex manipulation operations and pixel manipulation operations, graphics processing efficiency is enhanced as one type of data operations is not dependent upon another type of data operations. Therefore, any performance penalties experienced as a result of dependent operations in conventional graphics processors are overcome." (*Id.* at 11.)
15. A unified shader, comprising:	"A graphics processing architecture employing a single shader is disclosed." (*Id.* at 18.) "The present invention generally relates . . . to a graphics processor architecture employing a single shader." (*Id.* at 1.) "The shader of the present invention is referred to as a "unified" shader because it is configured to perform both vertex and pixel operations." (*Id.* at 4.) "Briefly stated, the present invention is directed to a graphics processor that employs a unified shader that is

capable of performing both the vertex operations and the pixel operations in a space saving and computationally efficient manner." (*Id.* at 4.)

"FIG. 4A is a schematic block diagram of a graphics processor architecture according to the present invention." (*Id.* at 12.) "Referring . . . to FIG. 4A, . . . [t]he arbiter 64 determines which of the two inputs to the multiplexer 66 is transmitted to a unified shader 62 for further processing." (*Id.* at 3.)

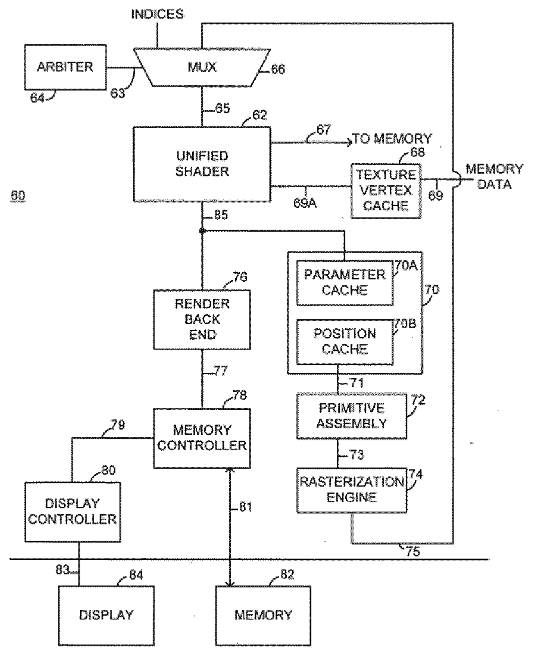


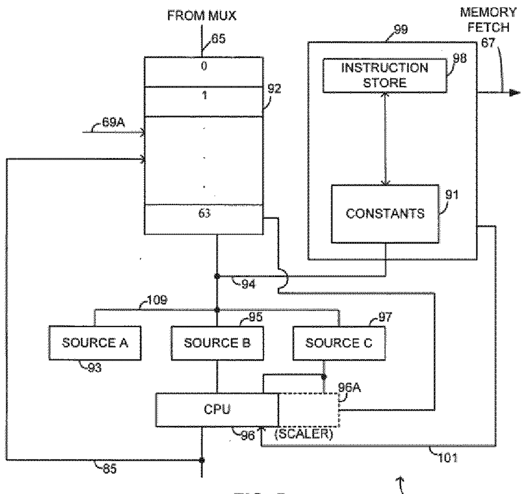
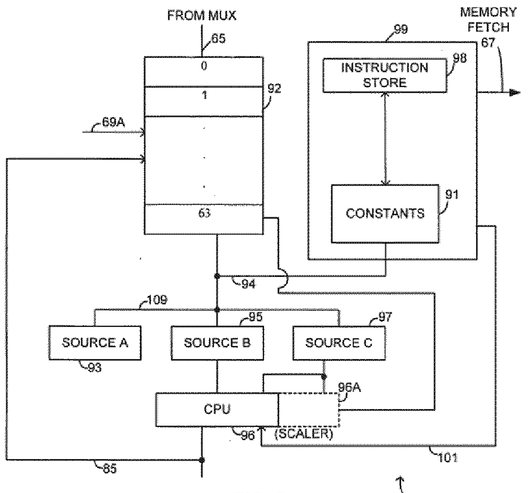
FIG. 4A

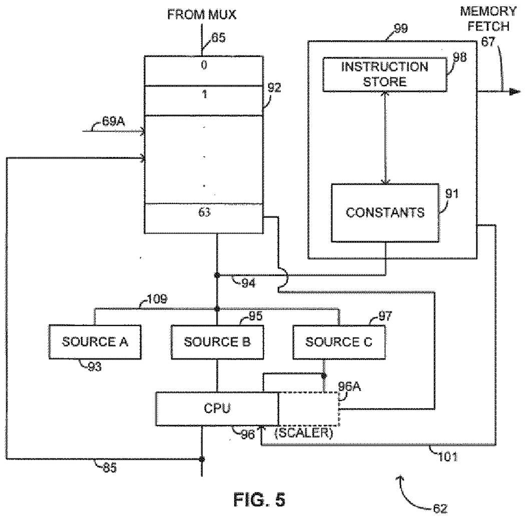
(*Id.* at FIG. 4A.)

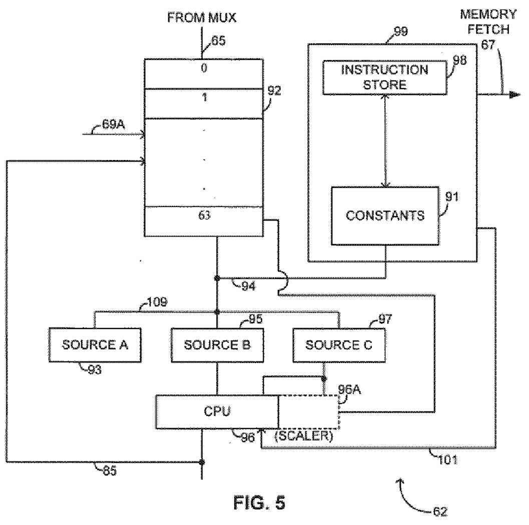
"In an exemplary embodiment, a graphics processor according to the present invention includes . . . a shader." (*Id.* at 4.)

"FIG. 5 is an exploded schematic block diagram of the unified shader employed in the graphics processor illustrated in FIG. 4A." (*Id.*)

"[T]he graphics processor 60 of the present invention incorporates a unified shader 62 which is capable of performing both vertex manipulation operations and

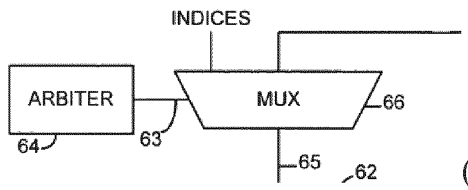
	pixel manipulation operations." (*Id.* at 11.)  FROM MUX 65 0 1 92 69A 63 MEMORY FETCH 99 67 98 INSTRUCTION STORE CONSTANTS 91 94 109 95 97 SOURCE A SOURCE B SOURCE C 93 96A CPU 96 (SCALER) 101 85 62 FIG. 5 (*Id.* at FIG. 5.)
15a. a general purpose register block for maintaining data;	"The shader includes a register block which is used to store the plurality of selected inputs." (*Id.* at 18.) "The shader includes a general purpose register block for storing at least the plurality of selected inputs." (*Id.* at 4.)  FROM MUX 65 0 1 92 69A 63 MEMORY FETCH 99 67 98 INSTRUCTION STORE CONSTANTS 91 94 109 95 97 SOURCE A SOURCE B SOURCE C 93 96A CPU 96 (SCALER) 101 85 62 FIG. 5 (*Id.* at FIG. 5.) "As illustrated, the unified shader 62 includes a general purpose register block 92 . . . . The general purpose register block 92 includes sixty four registers, or

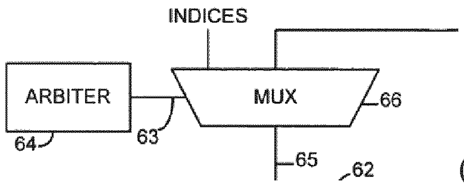
	available entries, for storing the information transmitted from the multiplexer 66 on line 65 or any other information to be maintained within the unified shader." (*Id.* at 7.) "In application, vertex data to be processed is transmitted into the general purpose register block 92 from multiplexer 66." (*Id.* at 9.)
15b. a processor unit; and	"The shader includes . . . a processor capable of executing both floating point arithmetic and logical operations on the selected inputs in response to the instructions maintained in the sequencer." (*Id.* at 18.) "The shader includes . . . a processor capable of executing both floating point arithmetic and logical operations on the selected inputs according to the instructions maintained in the sequencer." (*Id.* at 4.)  FROM MUX 65 MEMORY FETCH 67 99 INSTRUCTION STORE 98 0 1 92 69A 63 CONSTANTS 91 94 109 95 97 SOURCE A SOURCE B SOURCE C 93 96A CPU 96 (SCALER) 101 85 62 FIG. 5 (*Id.* at FIG. 5.) "As illustrated, the unified shader 62 includes . . . a processor (e.g. CPU) 96." (*Id.* at 7.) "The processor 96 may be comprised of a dedicated piece of hardware or can be configured as part of a general purpose computing device (i.e. personal computer). In an exemplary embodiment, the processor 96 is adapted to perform 32-bit floating point arithmetic

	operations as well as a complete series of logical operations on corresponding operands. As shown, the processor is logically partitioned into two sections. Section 96 is configured to execute, for example, the 32-bit floating point arithmetic operations of the unified shader. The second section, 96A, is configured to perform scaler operations (e.g. log, exponent, reciprocal square root) of the unified shader." (*Id.* at 8.) "[T]he processor 96 executes the instructions on the operands present in the source registers and provides the result on line 85." (*Id.* at 9.)
15c. a sequencer, coupled to the general purpose register block and the processor unit, the sequencer maintaining instructions operative to cause the processor unit to execute vertex calculation and pixel calculation operations on selected data maintained in the general purpose register block.	"The shader includes . . . a sequencer which maintains vertex manipulation and pixel manipulations instructions and a processor capable of executing both floating point arithmetic and logical operations on the selected inputs in response to the instructions maintained in the sequencer." (*Id.* at 18.) "The shader includes . . . a sequencer for storing logical and arithmetic instructions that are used to perform vertex and pixel manipulation operations." (*Id.* at 4.)  FROM MUX 65 0 1 92 69A 63 99 MEMORY FETCH 67 INSTRUCTION STORE 98 CONSTANTS 91 94 109 95 97 SOURCE A SOURCE B SOURCE C 93 96A CPU 96 (SCALER) 101 85 62 FIG. 5 (*Id.* at FIG. 5.) "As illustrated, the unified shader 62 includes . . . a sequencer 99." (*Id.* at 7.)

	"The sequencer 99 includes constants block 91 and an instruction store 98. The constants block 91 contains, for example, the several transformation matrices used in connection with vertex manipulation operations. The instruction store 98 contains the necessary instructions that are executed by the processor 96 in order to perform the respective arithmetic and logic operations on the data maintained in the general purpose register block 92 as provided by the source registers 93-95. The instruction store 98 further includes memory fetch instructions that, when executed, causes the unified shader 62 to fetch texture and other types of data, from memory 82 (FIG. 4A). In operation, the sequencer 99 determines whether the next instruction to be executed (from the instruction store 98) is an arithmetic or logical instruction or a memory (e.g. texture fetch) instruction. If the next instruction is a memory instruction or request, the sequencer 99 sends the request to a fetch block (not shown) which retrieves the required information from memory 82 (FIG. 4A). The retrieved information is then transmitted to the sequencer 99, through the vertex texture cache 68 (FIG. 4A)." (*Id.* at 8.) "If the next instruction to be executed is an arithmetic or logical instruction, the sequencer 99 causes the appropriate operands to be transferred from the general purpose register block 92 into the appropriate source registers (93, 95, 97) for execution, and an appropriate signal is sent to the processor 96 on line 101 indicating what operation or series of operations are to be executed on the several operands present in the source registers." (*Id.* at 9.) "[T]he instruction store 98 maintains both vertex manipulation instructions and pixel manipulation instructions. Therefore, the unified shader 99 of the present invention is able to perform both vertex and pixel operations, as well as execute memory fetch operations. As such, the unified shader 62 of the present

	invention is able to perform both the vertex shading and pixel shading operations on data in the context of a graphics controller based on information passed from the multiplexer. By being adapted to perform memory fetches, the unified shader of the present invention is able to perform additional processes that conventional vertex shaders cannot perform; while at the same time, perform pixel operations." (*Id.*) "The instruction store 98 then passes the corresponding control signals to the processor 96 on line 101 to perform such vertex operations. However, if the general purpose register block 92 does not have enough available space therein to store the incoming vertex data, such information will not be transmitted as the arbitration scheme of the arbiter 64 is not satisfied. In this manner, any pixel calculation operations that are to be, or are currently being, performed by the processor 96 are continued, based on the instructions maintained in the instruction store 98, until enough registers within the general purpose register block 92 become available." (*Id.* at 10.) "[T]he graphics processor 60 of the present invention incorporates a unified shader 62 which is capable of performing both vertex manipulation operations and pixel manipulation operations based on the instructions stored in the instruction store 98." (*Id.* at 11.)
17. The shader of claim 15, further including a selection circuit operative to provide information to the general purpose block in response to a control signal.	"The architecture includes a circuit operative to select one of a plurality of inputs in response to a control signal." (*Id.* at 18.) "In an exemplary embodiment, a graphics processor according to the present invention includes an arbiter circuit for selecting one of a plurality of inputs for processing in response to a control signal." (*Id.* at 4.) "[V]ertex information . . . is coupled to the first input of multiplexer 66." (*Id.* at 11.)

	"The resulting pixel data from the rasterization engine block 74 is the interpolated pixel parameter data that is transmitted to the second input of the multiplexer 66 on line 75." (*Id.* at 10.) "Referring now to FIG. 4A, in an exemplary embodiment, the graphics processor 60 of the present invention includes a multiplexer 66 having vertex (e.g. indices) data provided at a first input thereto and interpolated pixel parameter (e.g. position) data and attribute data from a rasterization engine 74 provided at a second input. A control signal generated by an arbiter 64 is transmitted to the multiplexer 66 on line 63. The arbiter 64 determines which of the two inputs to the multiplexer 66 is transmitted to a unified shader 62 for further processing. The arbitration scheme employed by the arbiter 64 is as follows: the vertex data on the first input of the multiplexer 66 is transmitted to the unified shader 62 on line 65 if there is enough resources available in the unified shader to operate on the vertex data; otherwise, the interpolated pixel parameter data present on the second input will be passed to the unified shader 62 for further processing." (*Id.* at 7.)  (*See id.* at FIG. 4A.)
18. The shader of claim 17, wherein the selection circuit is a multiplexer and the control signal is provided by an arbiter.	"Referring now to FIG. 4A, in an exemplary embodiment, the graphics processor 60 of the present invention includes a multiplexer 66 having vertex (e.g. indices) data provided at a first input thereto and interpolated pixel parameter (e.g. position) data and attribute data from a rasterization engine 74 provided at a second input. A control signal generated by an arbiter 64 is transmitted to the multiplexer 66 on line 63. The arbiter 64 determines which of the two inputs to the multiplexer 66 is transmitted to a unified shader 62 for

	further processing. The arbitration scheme employed by the arbiter 64 is as follows: the vertex data on the first input of the multiplexer 66 is transmitted to the unified shader 62 on line 65 if there is enough resources available in the unified shader to operate on the vertex data; otherwise, the interpolated pixel parameter data present on the second input will be passed to the unified shader 62 for further processing." (*Id.* at 7.)  (*See id.* at FIG. 4A.)
20. The shader of claim 15, wherein the processor unit executes vertex calculations while the pixel calculations are still in progress.	"In an exemplary embodiment, a graphics processor according to the present invention includes . . . a shader, . . . the shader including means for performing vertex operations and pixel operations." (*Id.* at 4.) "The unified shader 62 has ability to simultaneously perform vertex manipulation operations and pixel manipulation operations at various degrees of completion by being able to freely switch between such programs or instructions, maintained in the instruction store 98, very quickly. In application, vertex data to be processed is transmitted into the general purpose register block 92 from multiplexer 66. The instruction store 98 then passes the corresponding control signals to the processor 96 on line 101 to perform such vertex operations. However, if the general purpose register block 92 does not have enough available space therein to store the incoming vertex data, such information will not be transmitted as the arbitration scheme of the arbiter 64 is not satisfied. In this manner, any pixel calculation operations that are to be, or are currently being, performed by the processor 96 are continued, based on the instructions maintained in the instruction store 98, until enough registers within the general purpose register block 92 become available. Thus,

	through the sharing of resources within the unified shader 62, processing of image data is enhanced as there is no down time associated with the processor 96." (*Id.* at 9.) "[A]s the unified shader 62 is capable of alternating between performing vertex manipulation operations and pixel manipulation operations, graphics processing efficiency is enhanced as one type of data operations is not dependent upon another type of data operations. Therefore, any performance penalties experienced as a result of dependent operations in conventional graphics processors are overcome." (*Id.* at 11-12.)

XI. CONCEPTION

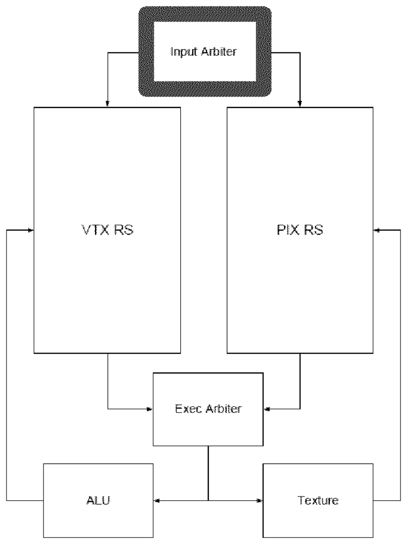
272. It is my understanding that conception is a mental formulation and disclosure by the inventor or inventors of a complete idea for a product or process. I also understand that conception turns on the inventor's ability to describe his or her invention with particularity, and conception must be sufficiently complete so as to enable the POSA to reduce the concept to practice.

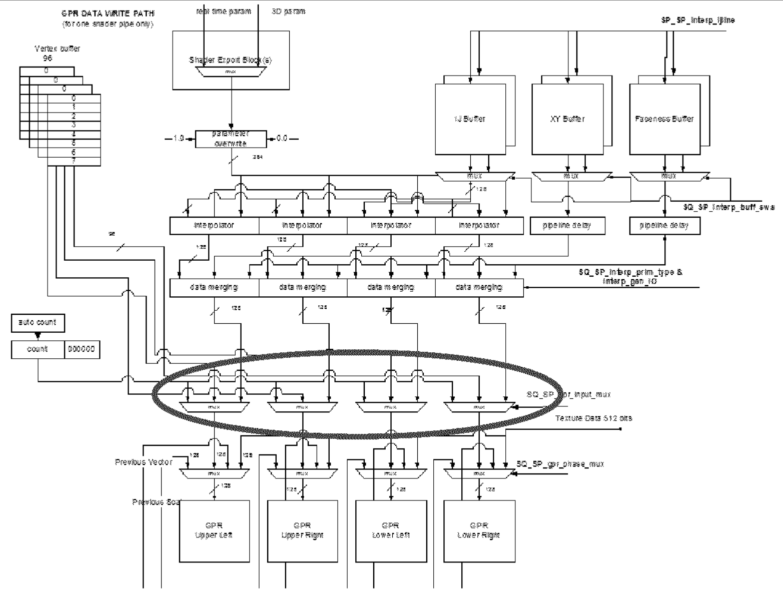
273. I have reviewed a document titled "R400 Top Level Specification" (Ex. 2041), a document titled "Shader Processor" (Ex. 2042), and two versions of a document titled "R400 Sequencer Specification" (Exs. 2010, 2028). These specification documents show that the inventors of the '871 patent—Steven Morein, Laurent Lefebvre, Andy Gruber, and Andi Skende—were collectively in possession of a complete embodiment of the claimed subject matter.

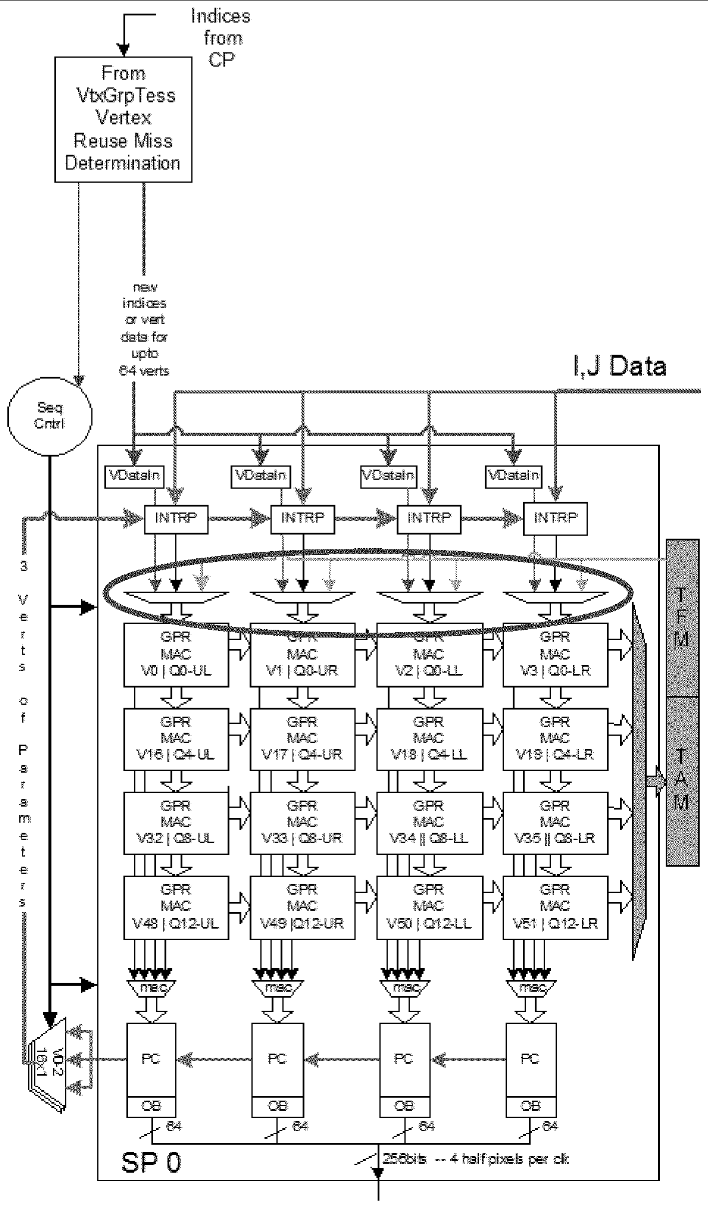
274. Each and every claim element is shown in the R400 specification documents. Further, the R400 specification documents provide sufficient detail to enable the POSA to reduce the concept to practice. Reducing the concept to practice could require substantial work, but would not require undue experimentation.

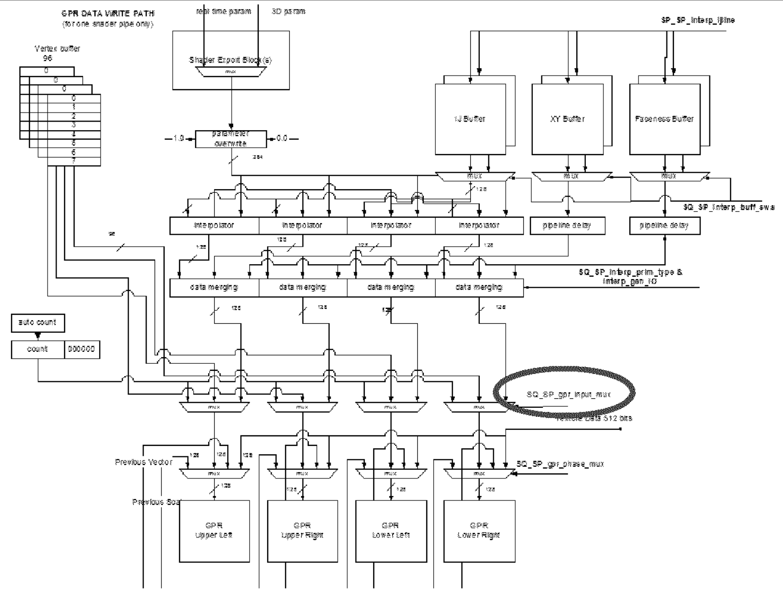
275. The following claim chart shows that the inventors conceived of the claimed subject matter no later than the date of these R400 specification documents.

Support for the '871 Patent Claims in ATI Specifications	
'871 Patent Claim	
1. A graphics processor, comprising:	The R400 Sequencer Specification, the Shader Processor specification, and the R400 Top Level Specification are architectural specifications for the R400. (Ex. 2028, p. 1 ("This is an architectural specification for the R400 Sequencer block (SEQ)"); Ex. 2042, p. 5 ("ShaderPipe (SP) [is] for the R400 Graphics Processor"); Ex. 2041.) The R400 was a graphics-chip product, which was designed to include a unified processing pipe (i.e., a single programmable pipeline for 2D video, 3D vertex, and 3D

	pixel operations). (*See* Ex. 2041, pp. 6, 7.)
1a. an arbiter circuit for selecting one of a plurality of inputs in response to a control signal; and	There is an arbiter circuit. The claimed arbiter circuit comprises the input arbiter and at least one GPR input multiplexer. The control signal is SQ_SP_gpr_input_mux. The input arbiter is outlined in red on the figure below.  Input Arbiter VTX RS PIX RS Exec Arbiter ALU Texture (Ex. 2028, p. 10.) The GPR input multiplexers are circled in red in the diagram below. "The diagram below shows all the possible data paths going into the GPR write paths, their selection and routing." (Ex. 2042, p. 28.)

	 GPR DATA WRITE PATH (for one shader pipe only) real time param 3D param SP_SP_interp_ijline Vertex buffer Shader Export Block(s) IJ Buffer XY Buffer Faceness Buffer parameter overwrite mux interpolator pipeline delay SQ_SP_interp_buff_swa data merging SQ_SP_interp_prim_type & interp_gen_IO auto count count SQ_SP_gpr_input_mux Texture Data 512 bits Previous Vector Previous Scalar SQ_SP_gpr_phase_mux GPR Upper Left GPR Upper Right GPR Lower Left GPR Lower Right (*See* Ex. 2042, p. 29.) The figure below also shows the GPR input multiplexers.

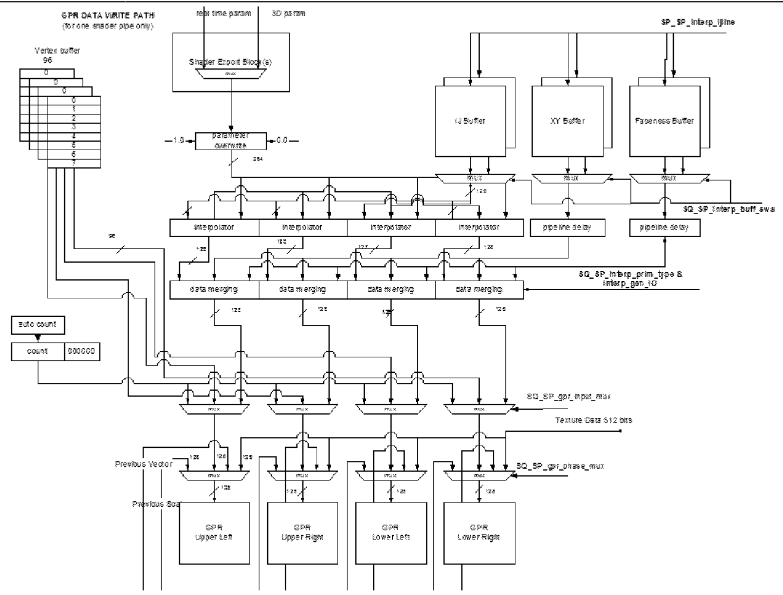
	 Indices from CP From VtxGrpTess Vertex Reuse Miss Determination new indices or vert data for upto 64 verts I,J Data Seq Cntrl VDataIn INTRP 3 Verts of Parameters GPR MAC V0 | Q0-UL GPR MAC V1 | Q0-UR GPR MAC V2 | Q0-LL GPR MAC V3 | Q0-LR GPR MAC V16 | Q4-UL GPR MAC V17 | Q4-UR GPR MAC V18 | Q4-LL GPR MAC V19 | Q4-LR GPR MAC V32 | Q8-UL GPR MAC V33 | Q8-UR GPR MAC V34 | Q8-LL GPR MAC V35 | Q8-LR GPR MAC V48 | Q12-UL GPR MAC V49 | Q12-UR GPR MAC V50 | Q12-LL GPR MAC V51 | Q12-LR TFM TAM mac PC OB V0-2 16:1 64 SP 0 256bits -- 4 half pixels per clk (*See* Ex. 2042, pp. 16, 30.) The interface between the input arbiter and the GPR input multiplexers includes the "SQ_SPx_gpr_input_mux" signal. (*Id.* at 19-20.) The signal is circled in red on the figure below.



(*See Id.* at 29.)

<u>The arbiter circuit selects one of a plurality of inputs.</u>

As shown in the diagram below, the GPR input multiplexers select from a plurality of inputs originating from: (1) vertex buffers; (2) interpolators; and (3) a count.



(*Id.*)

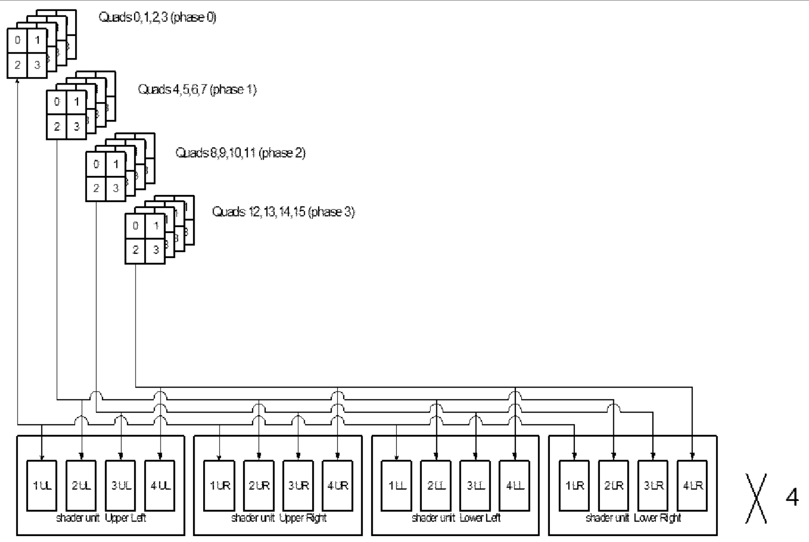
"The control of all the multiplexers present at the input . . . of the GPRs . . . and output of the parameter caches is done by the sequencer." (*Id.* at 30.)

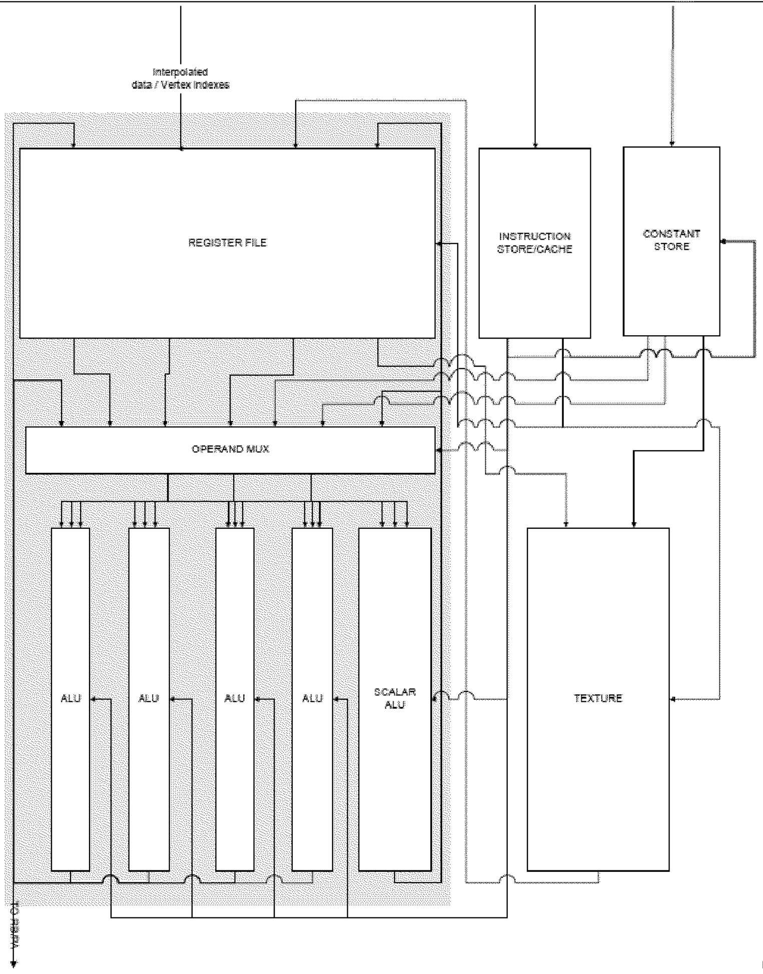
The sequencer first arbitrates between vectors of vertices that arrive from a primitive assembly and vectors of pixels that are generated in the scan converter/rasterizer. (*See* Ex. 2028, p. 6; Ex. 2041, p. 28; *see also* Ex. 2042, p. 28 (stating that the scan converter to shader pipe interface is the IJ bus).)

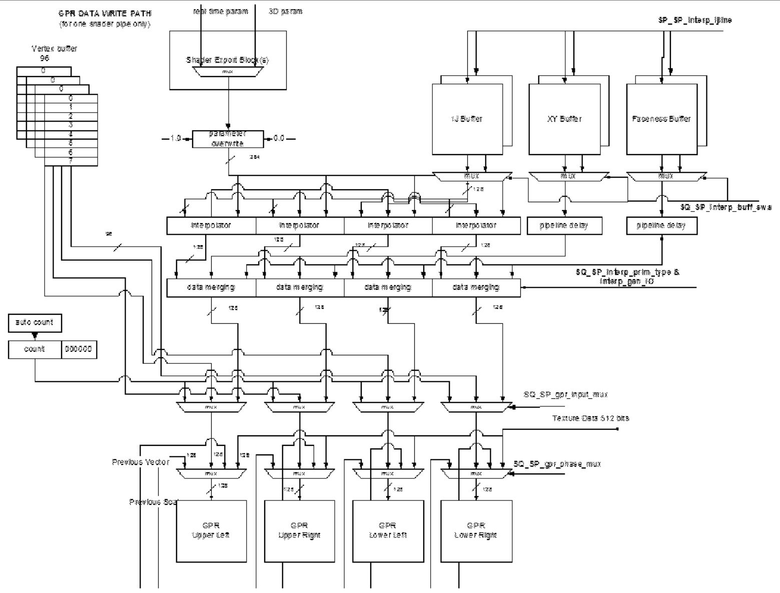
For selecting vertex vectors, "[w]hich of the four shader

	pipelines [a vertex vector] is issued to [is] determined either by some effort of load balancing or a simple round robin." (Ex. 2041, p. 10.) For pixel vectors generated by the rasterizer, "the rasterizer (which includes the sequencer and the shader pipeline) checks to make sure that there are enough free registers in the shader pipeline for the pixel shader program. If not, it stalls until there are enough. The rasterizer also needs to arbitrate between the three streams of vectors to be shaded: the vertex stream, the pixel stream, and the real time stream. I think it will be sufficient for the real time stream to have priority over the vertex stream which has priority over the pixel stream." (*Id.* at 11.) <u>The arbiter circuit selects in response to a control signal.</u> The GPR input multiplexers of the arbiter circuit selects an input in response to a control signal, which is the SQ_SP_gpr_input_mux. The GPR input multiplexers are "controlled by

	SQ_SP_gpr_input_mux part of the 'SQ_SP: Interpolation bus' interface . . . to route between vertex data/indices and interpolated pixel parameters." (Ex. 2042, p. 28.) The claimed control signal (the `SQ_SP_gpr_input_mux`) is sent along this interface.
1b. a shader, coupled to the arbiter circuit, operative to process the selected one of the plurality of inputs, the shader including means for performing vertex operations and pixel operations, and performing one of the vertex operations or pixel operations based on the selected one of the plurality of inputs,	<u>There is a shader.</u> "[T]he R400 Shader Pipe truly represents an Unified Shader Architecture. In R400, both vertex and pixel shading operations are implemented through the shader units." Ex. 2042, p. 5.) "The unified shader is a simd/vector engine that performs the same instructions on four sets of four (16 total) elements." (Ex. 2041, p. 9.) "As shown in the figure reproduced below, "four identical processing units comprise a shader unit." (Ex. 2042, p. 15.)

wherein the shader provides a appearance attribute.	 Quads 0,1,2,3 (phase 0) Quads 4,5,6,7 (phase 1) Quads 8,9,10,11 (phase 2) Quads 12,13,14,15 (phase 3) 0 1 2 3 1 UL 2 UL 3 UL 4 UL shader unit Upper Left 1 UR 2 UR 3 UR 4 UR shader unit Upper Right 1 LL 2 LL 3 LL 4 LL shader unit Lower Left 1 LR 2 LR 3 LR 4 LR shader unit Lower Right X 4 (Ex. 2042, p. 15.) A shader unit is represented by the gray area on the figure reproduced below. (Ex. 2010, p. 11.)

	 Interpolated data / Vertex indexes REGISTER FILE INSTRUCTION STORE/CACHE CONSTANT STORE OPERAND MUX ALU ALU ALU ALU SCALAR ALU TEXTURE (*Id.* at 11.) <u>The shader is coupled to the arbiter circuit.</u> The general purpose registers are part of the shader, as shown in the figure above. (*See id.* at 10.) And as shown in the figure below, the registers are coupled to the GPR input multiplexers.

	 GPR DATA WRITE PATH (for one shader pipe only) real time param 3D param SP_SP_interp_ijline Vertex buffer 96 Shader Export Block(s) mux IJ Buffer XY Buffer Faceness Buffer parameter overwrite 1.0 0.0 SQ_SP_interp_buff_swa interpolator pipeline delay SQ_SP_interp_prim_type & interp_gen_IO data merging auto count count 000000 SQ_SP_gpr_input_mux Texture Data 512 bits Previous Vector Previous Scal SQ_SP_gpr_phase_mux GPR Upper Left GPR Upper Right GPR Lower Left GPR Lower Right (Ex. 2042, p. 29.) The multiplexers are part of the arbiter circuit, which receive the SQ_SP_gpr_input_mux signal provided by the input arbiter component of the arbiter circuit and propagate the selected input to the shader. (*See supra* Claim 1a.) So, the shader is coupled to the arbiter circuit. <u>The shader includes a means for performing vertex operations and pixel operations.</u> The shader includes ALUs, which perform both vertex operations and pixel operations because the "Shader Pipe (SP) serves as the central Arithmetic and Logic Unit

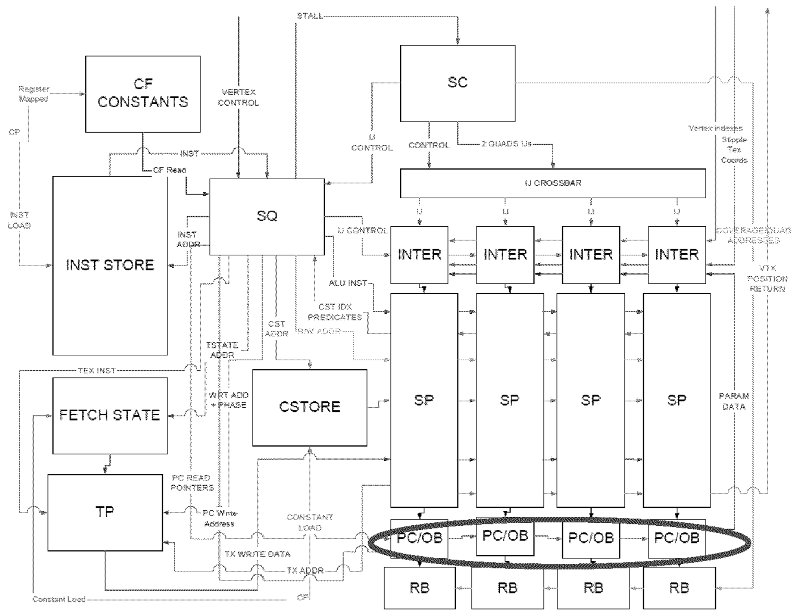
	(ALU) for the R400 Graphics Processor, and "both vertex and pixel shading operations are implemented through the shader units." (*See id.* at 5.) The shader comprises ALUs. (*See* Ex. 2010, p. 11; *see also* Ex. 2042, p. 7.) "ALU consist of two distinct units: 'Vector' ALU and the 'Scalar' ALU. The Vector ALU performs operations in parallel across a 4-component vector, while the Scalar ALU performs operations on a single component of a vector which is then replicated across all components." (Ex. 2042, p. 7.) "An ALU can do simple math, conditional moves, and permutations on the registers, and the ability to do a limited number of memory reads using the texture cache." (Ex. 2041, p. 10.) The shader can performs one of the vertex operations or pixel operations based on the selected one of the plurality of inputs

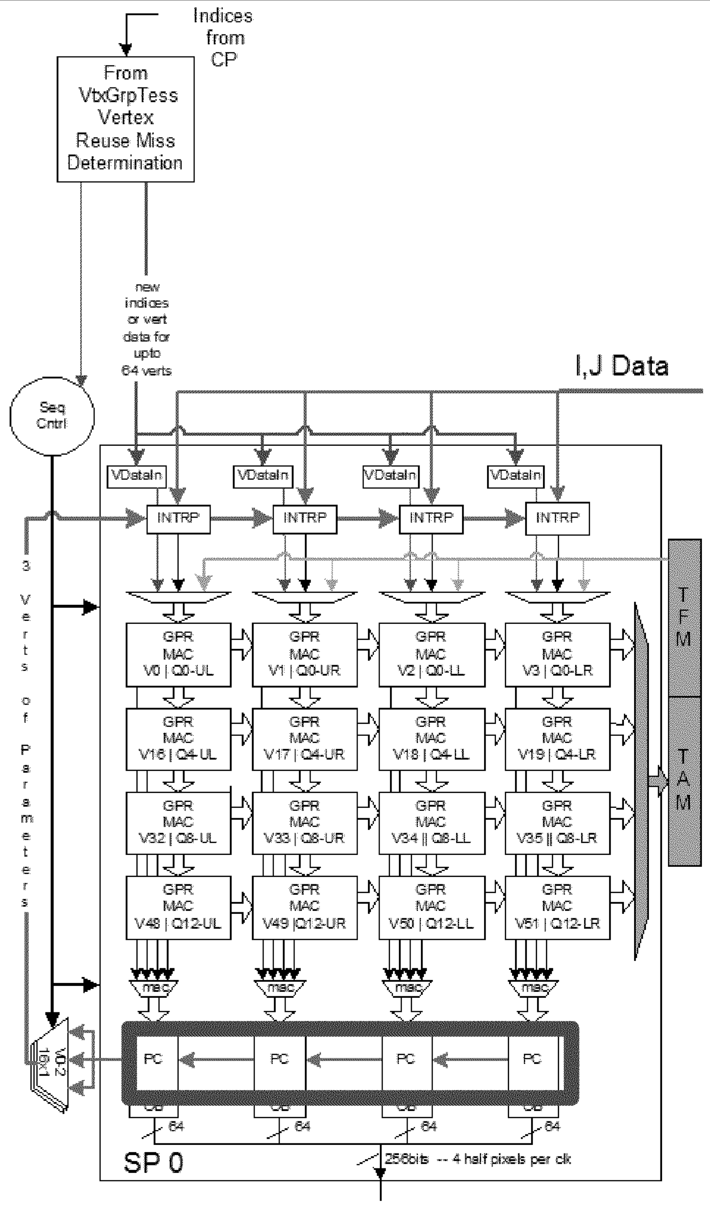
	"All the shader units of each and every pipe execute the same ALU instruction on different sets of vertex parameters/pixel values." (Ex. 2042, p. 5.) "For 3D rendering data is passed twice through the unified shader- once to transform the vertices and a second time to determine the color of the pixels." (Ex. 2041, p. 10.) (1) "The input to the 3D pipe is expected to be indexed vertex arrays." (*Id.* at 10.) When either a vector is filled with 16 entries or a state change happens . . . the vector is issued to one of the 'shader' pipelines for transformation. Which of the four shader pipelines it is issued to [is] determined either by some effort of load balancing or a simple round robin." (*Id.*) "[When t]he shader pipeline receives the vector of 16 indices from the primitive assembly block[,] [t]he shader pipeline operates, when rendering pixels, by processing a vector of four 2x2 pixel footprints, a total of 16 pixels." (*Id.*) "At the end of the vertex program, the transformed coordinates must be

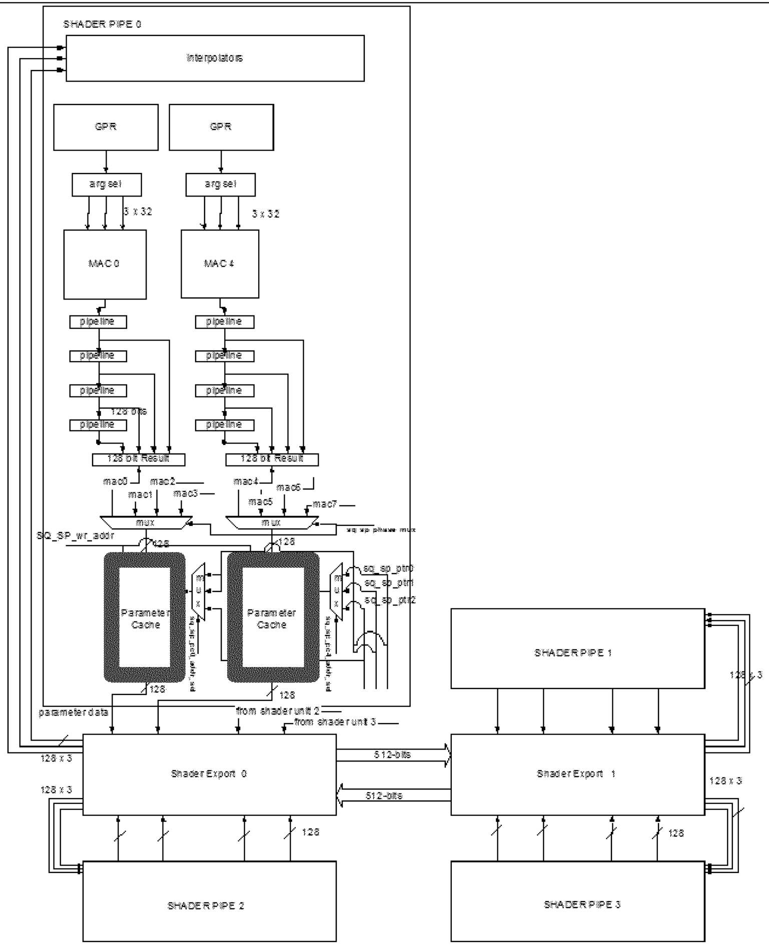
	output." (*Id.*) (2) "Before starting the processing of a vector the rasterizer (which includes the sequencer for the shader pipeline) checks to make sure that there are enough free registers in the shader pipeline for the pixel shader program." (*Id.* at 11.) "The vector is then processed by the shader pipeline. We will probably support up to eight sequentially dependent texture fetches . . . . 16 (8?) textures are supported, but each texture can be accessed multiple times by a single pixel shader." (*Id.* at 11.) <u>The shader provides an appearance attribute.</u> "The location where the data should be put in the event of an export is specified by in the destination address field of the ALU instruction word." (Ex. 2042, p. 10.) The Shader specification lists the possible types of exports and the range of addresses. (*Id.* at 10-11.) The list is divided into vertex shading and pixel shading. (*Id.*) And the list comprises of different appearance attributes such as

	position and color. (*See id.*) "One output will be the x, y, z, w position . . . . The vertex program may also output a number of parameter values (colors, texture coordinates, other interpolated inputs into the pixel shader)." (Ex. 2041, p. 10.) "The output of the pixel shader is the final color of the fragment." (*Id.* at 11.)
2. The graphics processor of claim 1, further including a vertex storage block for maintaining vertex information.	<u>There is a vertex storage block.</u> The claimed vertex storage block is a vertex cache, a parameter cache, and a position cache. The R400 specifications describe a vertex cache. (*See id.* at 10-11.) "Vertices are located in the vertex cache after the vertices were transformed. (*See id.* at 25.) "The shader pipeline will fetch the vertex array data through the cache infrastructure that is also used for texture fetches." (*Id.* at 10.) The R400 specifications also disclose a parameter cache

	and a position cache. (*See infra* Claim 3). <u>The vertex storage block maintains vertex information.</u> The vertex storage block maintains vertex information because "[t]he vertex cache stores transformed vertices." (*Id.* at 10.) The parameter cache and the position cache also maintain vertex information. (*See infra* Claim 3.)
3. The graphics processor of claim 2, wherein the vertex storage block further includes a parameter cache operative to maintain appearance attribute data for a corresponding vertex and a position cache operative to maintain position data for a	*See supra* Claim 2 (showing support for the vertex storage block). <u>The vertex storage block includes a parameter cache.</u> The R400 specifications describe a parameter cache. (*See id.* at 10-13; Ex. 2028, p. 36.) The parameter cache is included in the vertex storage block. (*See* Ex. 2041, pp. 10-11 (describing the parameter cache as the "parameter portion of the vertex cache"), 28 (describing the parameter cache as the "vertex parameter cache").

corresponding vertex.	The R400 Sequencer Specification shows four parameter caches, circled in red below. PC stands for parameter cache. (Ex. 2042, pp. 18, 30.)  STALL CF CONSTANTS Register Mapped CP VERTEX CONTROL SC IJ CONTROL CONTROL 2 QUADS IJs Vertex indexes Stipple Tex Coords INST CF Read IJ CROSSBAR INST LOAD INST STORE SQ INST ADDR IJ CONTROL IJ INTER COVERAGE/QUAD ADDRESSES VTX POSITION RETURN ALU INST CST IDX PREDICATES CST ADDR R/W ADDR TSTATE ADDR TEX INST FETCH STATE WRT ADD + PHASE CSTORE SP PARAM DATA TP PC READ POINTERS PC Write Address CONSTANT LOAD TX WRITE DATA TX ADDR PC/OB RB Constant Load CP (*See* Ex. 2028, p. 7.) The Shader Processor specification also has two figures showing the parameter cache blocks, reproduced below with the parameter cache blocks outlined in red.

	 Indices from CP From VtxGrpTess Vertex Reuse Miss Determination new indices or vert data for upto 64 verts I,J Data Seq Cntrl VDataIn INTRP 3 Verts of Parameters GPR MAC V0 | Q0-UL GPR MAC V1 | Q0-UR GPR MAC V2 | Q0-LL GPR MAC V3 | Q0-LR GPR MAC V16 | Q4-UL GPR MAC V17 | Q4-UR GPR MAC V18 | Q4-LL GPR MAC V19 | Q4-LR GPR MAC V32 | Q8-UL GPR MAC V33 | Q8-UR GPR MAC V34 | Q8-LL GPR MAC V35 | Q8-LR GPR MAC V48 | Q12-UL GPR MAC V49 | Q12-UR GPR MAC V50 | Q12-LL GPR MAC V51 | Q12-LR TFM TAM mac PC V0-2 16x1 64 SP 0 256bits -- 4 half pixels per clk (*See* Ex. 2042, pp. 16, 30.)

	 SHADER PIPE 0 Interpolators GPR GPR arg sel arg sel 3 x 32 3 x 32 MAC 0 MAC 4 pipeline 128 bits 128 bit Result 128 bit Result mac0 mac1 mac2 mac3 mac4 mac5 mac6 mac7 mux mux sq_sp_phase_mux SQ_SP_wr_addr 128 Parameter Cache Parameter Cache m u x sq_sp_ptr0 sq_sp_ptr1 sq_sp_ptr2 sq_sp_pc0_addr_sel sq_sp_pc4_addr_sel parameter data from shader unit 2 from shader unit 3 128 x 3 Shader Export 0 Shader Export 1 512-bits SHADER PIPE 1 SHADER PIPE 2 SHADER PIPE 3 (*See id.* at 32.) <u>The parameter cache is operative to maintain appearance attribute data for a corresponding vertex.</u> "The parameter cache is where the vertex shaders export their data." (Ex. 2028, p. 36; *see also* Ex. 2028, p. 40 ("a vertex shader exports its data TO THE PARAMETER CACHE").)

	"The vertex program may . . . output a number of parameter values (color, texture coordinates, other interpolated inputs." (Ex. 2041, p. 10.) "The rasterizer will request the parameter data from the parameter cache for the primitives . . . . The parameter cache is 512 bits wide," and the rasterizer can fetch parameters stored in the cache. (*Id.* at 11.). "The output of the vertex shader program, transformed parameter data is written into Parameter cache memories." (Ex. 2042, p. 32.) "The read address into parameter cache is a result of a muxing of three possible 7-bit address pointers broadcasted by the Sequencer to all shader pipes. These three pointers are part of 'Parameter Cache Read Control Bus' . . . . There are 512-bit worth of data transferred from Shader Pipe to SX blocks for every read of the parameter cache. Once read from the parameter caches, the parameter data is then routed by the SX units into the interpolation units at the top of the shader pipe." (*Id.* at 33.)

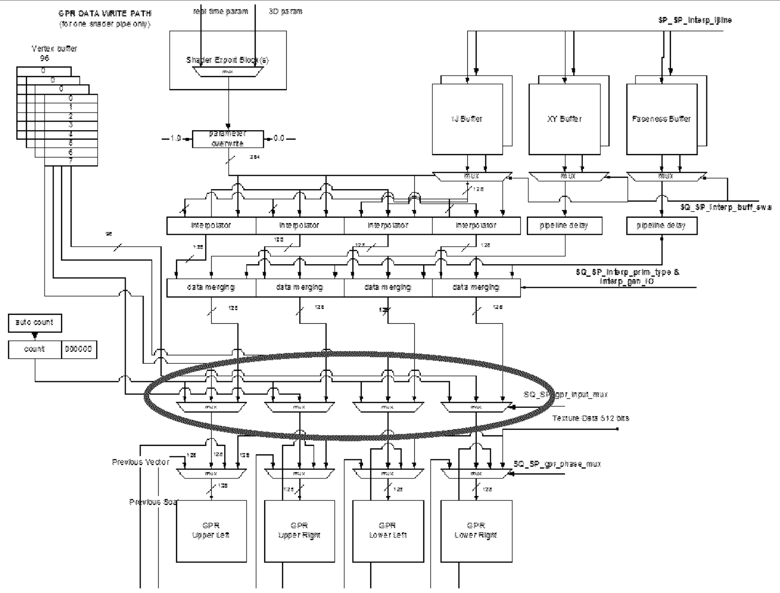
	"Parameter Cache Read control bus . . . provides three different pointers specifying the location of the parameter values in the Parameter Caches. Depending on the way the vertices get mapped into primitives, it might happen that the parameter values come from different relative offsets in the parameter caches from one parameter cache to the other across a shader pipe." (*Id.* at 18.) <u>The vertex storage block includes a position cache.</u> The R400 specifications describe a position cache. (*See* Ex. 2041, pp. 10, 26; Ex. 2028, p. 37 ("Position or parameter caches can be exported in any order in the shader program. It is always better to export posistion [sic] as soon as possible.").) The position cache is included in the vertex storage block. (*See* Ex. 2041, p. 10 (describing the position cache as the "position cache portion of the vertex cache"). <u>The position cache is operative to maintain position data for a corresponding vertex.</u>

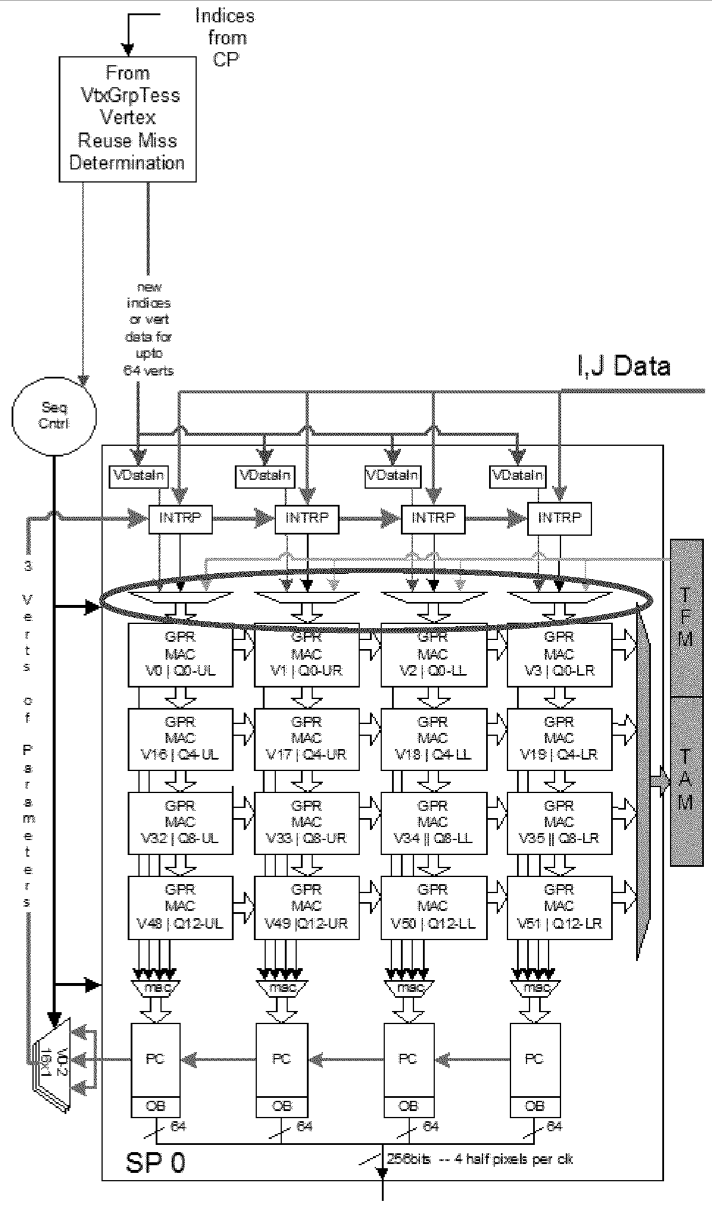
	"At the end of the vertex program, the transformed coordinates must be output. One output will be the x, y, z, w position which we be [sic] stored in the position cache." (*Id.*) "The primitive assembly block . . . accesses the position cache portion of the vertex cache." (*Id.*) The primitive assembly "receives the transformed vertex position data from the shader pipes." (*Id.* at 25.)
5. The graphics processor of claim 1, wherein the appearance attribute is position, and the position attribute is associated with a corresponding vertex when the selected one of the plurality of	*See supra* Claim 1b (showing support for the appearance attribute). <u>The appearance attribute can be a position attribute.</u> "One output will be the x, y, z, w position." (*Id.* at 10.) The R400 specifications list "the possible types of exports and the range of addresses," which includes position. (Ex. 2042, pp. 10-11.) <u>The position attribute is associated with a corresponding vertex when the selected one of the plurality of inputs is</u>

inputs is vertex data.	vertex data. "For 3D rendering data is passed twice through the unified shader- once to transform the vertices." (Ex. 2041, p. 10.) Position is associated with vertex shading. (*See* Ex. 2042, pp. 10-11.) "At the end of the vertex program, the transformed coordinates must be output. One output will be the x, y, z, w position." (Ex. 2041, p. 10.)
6. The graphics processor of claim 5, wherein the appearance attribute is color, and the color attribute is associated with a corresponding pixel when the selected one of the plurality of inputs is	*See supra* Claim 1b (showing support for the appearance attribute). The appearance attribute can be a color. "The output of the pixel shader is the final color of the fragment." (*Id.* at 11.) The R400 specifications list "the possible types of exports and the range of addresses," which includes color. (Ex. 2042, pp. 10-11.) "The vertex program may also output a number of

pixel data.	parameter values (colors, texture coordinates [etc.]).” (Ex. 2041, p. 10.) “When exporting Fog, color must be exported at the same time. Fog will be exported in the Scalar pipe and Color in the Vector pipe.” (Ex. 2042, p. 10.) <u>The color attribute is associated with a corresponding pixel when the selected one of the plurality of inputs is pixel data.</u> “For 3D rendering data is passed twice through the unified shader- once . . . to determine the color of the pixels.” (Ex. 2041, p. 10.) Color is associated with pixel shading. (*See* Ex. 2042, p. 11.) “The output of the pixel shader is the final color of the fragment.” (Ex. 2041, p. 11; *see also* Ex. 2041, p. 14 (“override the color output from the pixelshader with an ugly shade of green”).)

8. The graphics processor of claim 1, wherein the appearance value is depth.	*See supra* Claim 1b (showing support for the appearance attribute). <u>The appearance attribute can be depth.</u> "At the end of the vertex program, the transformed coordinates must be output. One output will be the x, y, z, w position." (*Id.* at 10.) A person having ordinary skill in the art would understand the z position to be depth. "Result of pixel kill in the shader pipe, which must be output for all pixel exports (depth and all color buffers)." (Ex. 2028, p. 51.)
9. The graphics processor of claim 1, further including a selection circuit, wherein the selection circuit is a multiplexer, and the control signal is provided by an	<u>There is a selection circuit.</u> The at least one GPR input multiplexer is the claimed selection circuit. The GPR input multiplexers are circled in red on the diagram below. "The diagram below shows all the possible data paths going into the GPR write paths, their selection, and routing."

arbiter, wherein the arbiter is coupled to the multiplexer.	 GPR DATA WRITE PATH (for one shader pipe only) real time param 3D param Shader Export Block(s) mux Vertex buffer 96 0 1 2 3 4 5 6 7 parameter overwrite 1.0 0.0 384 SP_SP_interp_ijline IJ Buffer XY Buffer Faceness Buffer mux 128 SQ_SP_interp_buff_swa interpolator pipeline delay 96 SQ_SP_interp_prim_type & interp_gen_IO data merging auto count count 000000 SQ_SP_gpr_input_mux Texture Data 512 bits Previous Vector Previous Scal SQ_SP_gpr_phase_mux GPR Upper Left GPR Upper Right GPR Lower Left GPR Lower Right (*See* Ex. 2042, p. 29.) The figure below also shows the GPR input multiplexers.

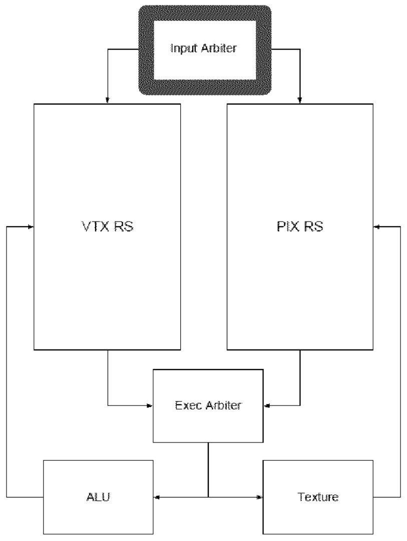
	 (*See id.* at 16, 30.) <u>The selection circuit is a multiplexer.</u> The GPR input selection circuits shown in the figures above are multiplexers. The R400 specifications call a

GPR input a "multiplexer" and a "MUX." (*See id.* at 28, 30; Ex. 2028, p. 40.)

<u>An arbiter provides the control signal.</u>

See supra Claim 1a (showing support for the control signal).

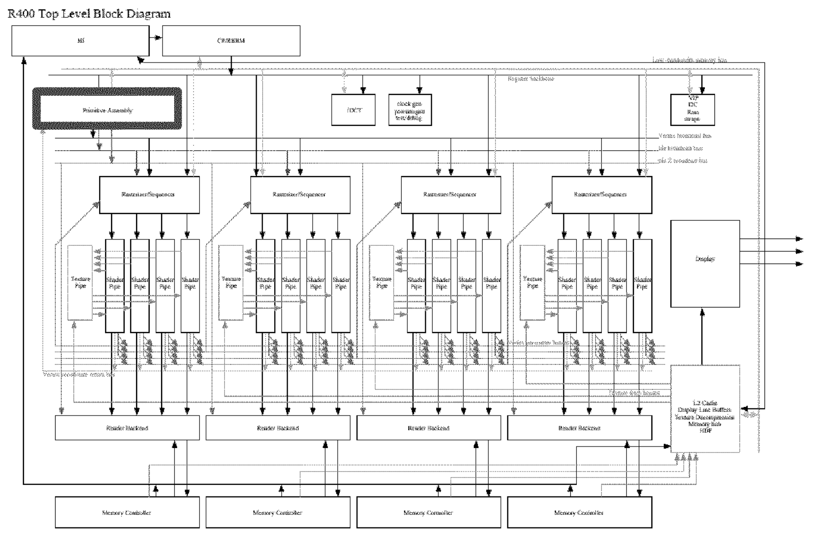
The claimed arbiter is the input arbiter outlined in red on the figure below.



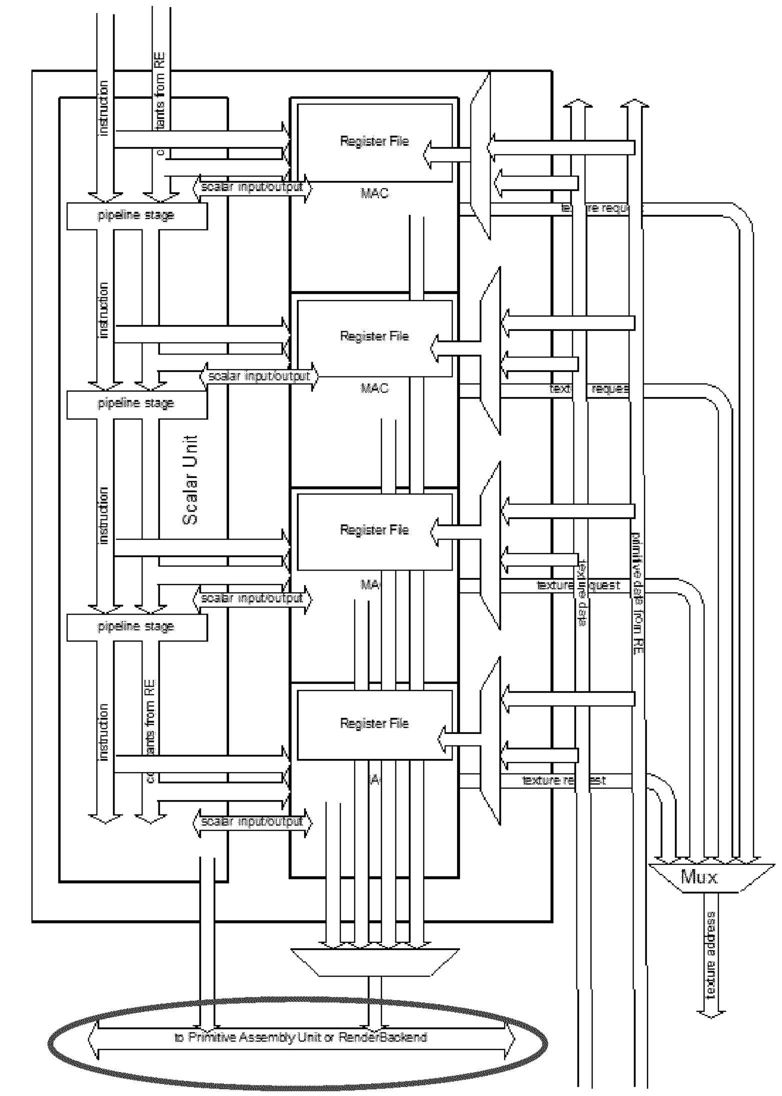
(Ex. 2028, p. 10.)

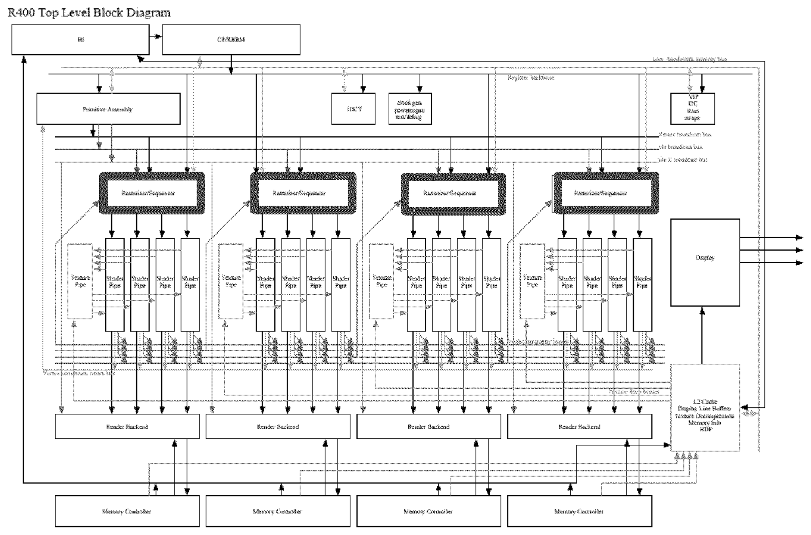
The arbiter provides the SQ_SP_gpr_input_mux control signal. "The control of all the multiplexers present at the input and output of the GPRs . . . is done by the

	sequencer." (*See* Ex. 2042, p. 30.) And the sequencer's input arbiter "first arbitrates between vectors of . . . vertices . . . and vectors of . . . pixels." (*See* Ex. 2028, pp. 6, 10.) <u>The arbiter is coupled to the multiplexer.</u> The arbiter is coupled to the multiplexers via the SQ_SP_gpr_input_mux. (*See* Ex. 2042, pp. 18-19; Ex. 2028, p. 51; *supra* Claim 1b (having both the multiplexer and the arbiter part of the arbiter circuit).)
10. The graphics processor of claim 1, wherein the shader provides vertex position data and further including a primitive assembly block, coupled to the shader, operative to generate primitives in	<u>The shader provides vertex position data.</u> "At the end of the vertex program, the transformed coordinates must be output. One output will be the x, y, z, w position . . . . The vertex program may also output a number of parameter values (colors, texture coordinates, other interpolated inputs into the pixel shader)." (Ex. 2041, p. 10.) <u>There is a primitive assembly block.</u> The claimed primitive assembly is outlined in red on the

response to the vertex position data.	top level block diagram shown below.  R400 Top Level Block Diagram CP/RBBM Primitive Assembly IDCT Rasterizer/Sequencer Texture Pipe Shader Pipe Display Render Backend Memory Controller (*See id.* at 15.) <u>The primitive assembly block is coupled to the shader.</u> The primitive assembly block is coupled to the shader via the PA → SPn bus. (*See id.* at 25.) <u>The primitive assembly block is operative to generate primitives in response to the vertex position data.</u> "[The primitive assembly] receives the transformed vertex position data from the shader pipes." (*Id.* at 25; *see also id.* at 10 ("The primitive assembly block reads the indices back out of the latency FIFO and accesses the position

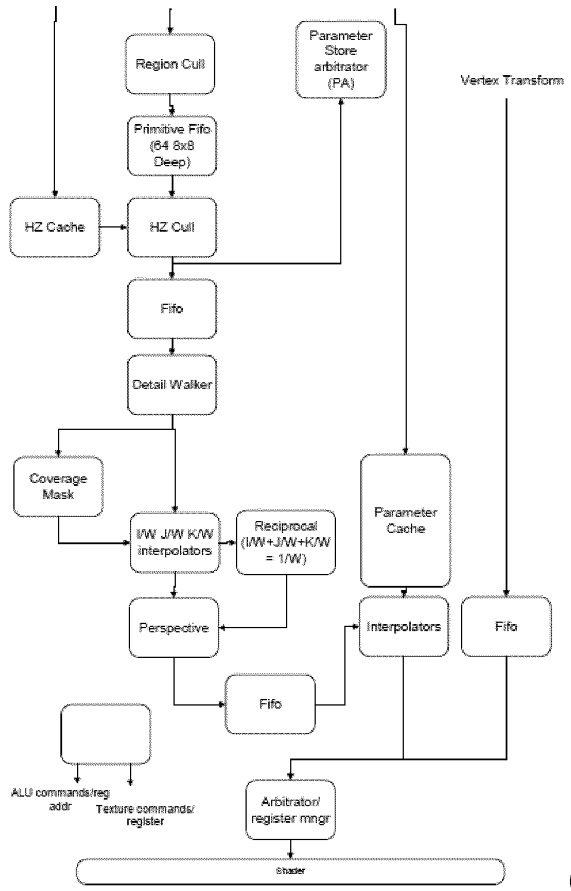
	cache portion of the vertex cache. It assembles the vertices into primitives (lines, triangles, rectangles, quads?, points, ?).”).) As circled in red on the figure of the shader pipe shown below, output data from the shader unit can go “to Primitive Assembly Unit.”

	 instruction constants from RE Register File MAC scalar input/output pipeline stage Scalar Unit texture request texture data primitive data from RE Mux texture address to Primitive Assembly Unit or RenderBackend (Ex. 2028, p. 13.) The transformed vertex position data is also shown to be sent along the vertex coordinate return bus in the top level block diagram. (*See* Ex. 2041, p. 15; *see also id.* at 25 (describing the PA → SPn bus as "8x8 tiles to be rasterized").)

	"The primitive assembly subblock then creates primitives (lines, points, rectangles, triangles) from the vertices." (*Id.* at 25.) "The resulting primitive data, including the indices back into the parameter portion of the vertex cache are broadcast to the four pipes." (*Id.* at 10.)
11. The graphics processor of claim 10, further including a raster engine, coupled to the primitive assembly block, operative to generate pixel parameter data in response to the assembled vertex data.	<u>There is a raster engine.</u> The claimed raster engine is outlined in red on the top level block diagram shown below  R400 Top Level Block Diagram Primitive Assembly Rasterizer/Sequencer Texture Pipe Shader Pipe Display Render Backend Memory Controller L2 Cache Display Line Buffers Texture Decompression Memory hub RDP (*See id.* at 15.) The R400 specifications also include a raster engine block

diagram, shown below.

8.15.3.1 RE Block diagram


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

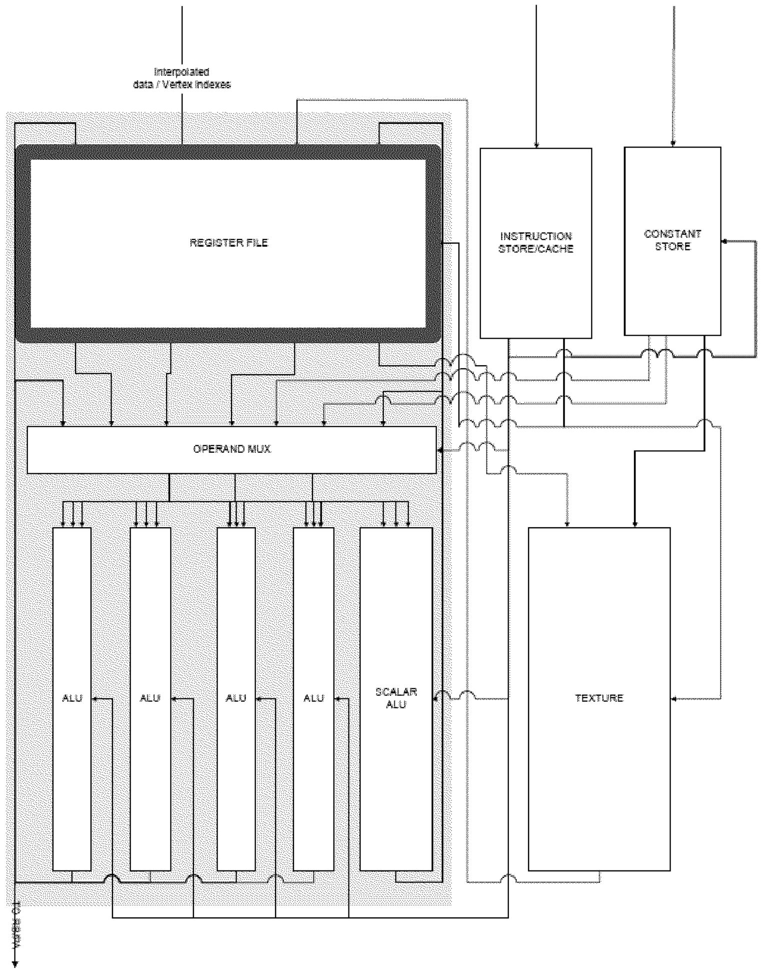
(*Id.* at 29.)

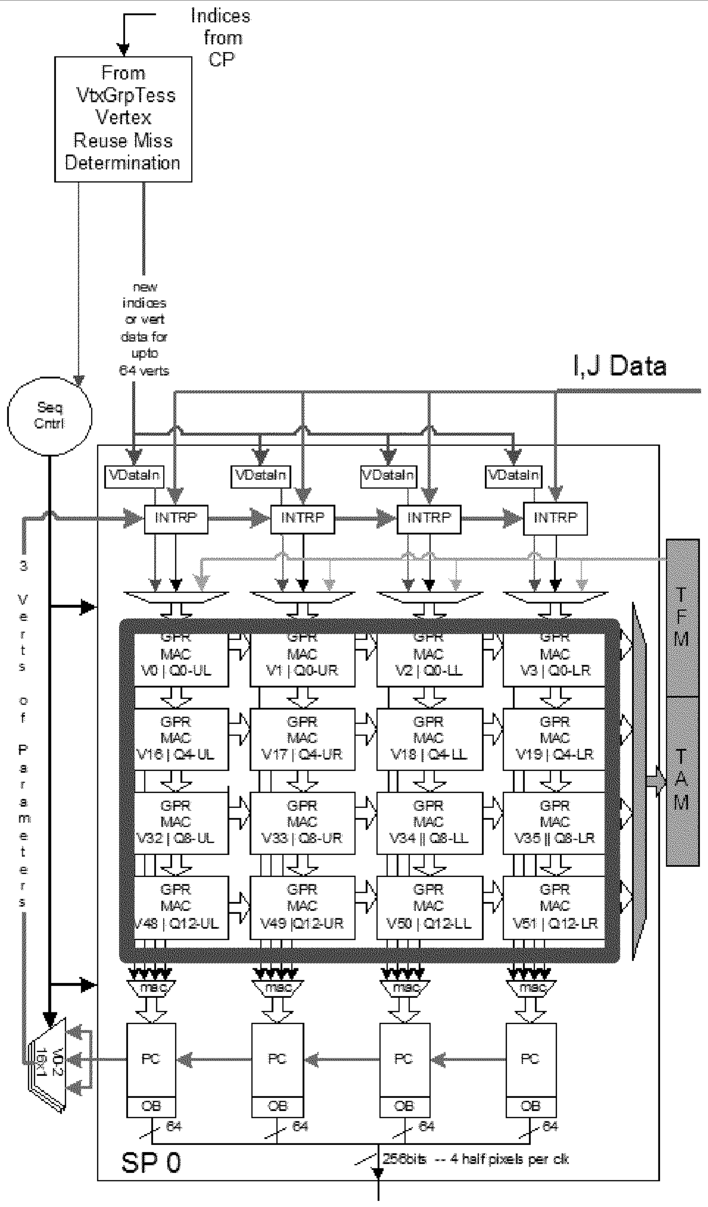
<u>The raster engine is coupled to the primitive assembly block.</u>

The raster engine is coupled to the primitive assembly via the primitive assembly → raster engine interface. (*See id.* at 15-16, 25, 28.)

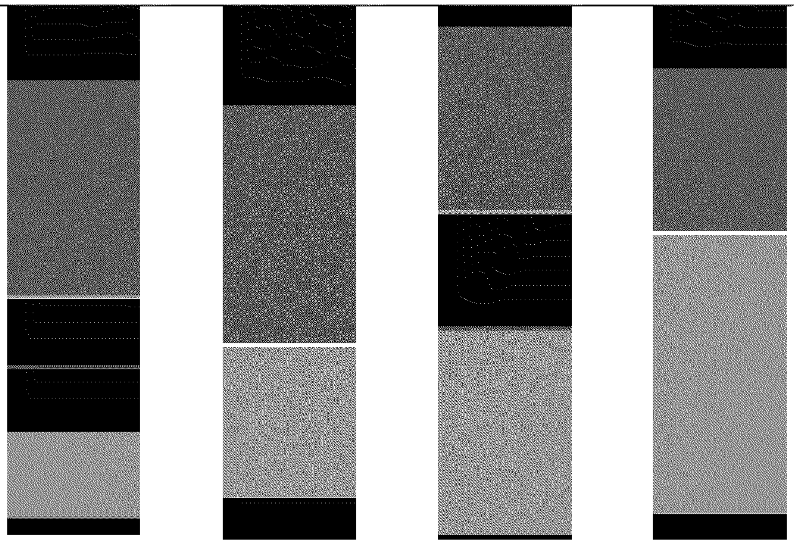
<u>The raster engine is operative to generate pixel parameter</u>

	<u>data in response to the assembled vertex data.</u> "[V]ectors of 4 quads (16 pixels) . . . are generated in the raster engine." (*Id.* at 28.) "The rasterizer will request the parameter data from the parameter cache for the primitives." (*Id.* at 11.) The rasterizer will generate four pixels per clock if there are no more than eight interpolated parameters. The rasterizer generates vectors of four 2x2 footprints (16 pixels)." (*Id.* at 11.) The primitive assembly → raster engine interface is described as "[r]equests to transform packets of vertices." (*Id.* at 28.)
13. The graphics processor of claim 1, wherein the shader includes a register block for maintaining the selected one of the plurality of inputs, a	*See supra* Claim 1b (showing support for the shader). <u>The shader includes a register block.</u> "The user model for the unified shader is composed of a variable number of general purpose registers, a subset of which are usually initialized with data." (*Id.* at10.)

computation element operative to perform arithmetic and logical operations on the data maintained in the register block, and sequencer for maintaining the instructions that are executed by the computation element.	A shader unit is represented by the gray area on the figure reproduced below. (Ex. 2010, p. 11.) The shader unit includes a register file, outlined in red. (*Id.* at11.)  Interpolated data / Vertex Indexes REGISTER FILE INSTRUCTION STORE/CACHE CONSTANT STORE OPERAND MUX ALU ALU ALU ALU SCALAR ALU TEXTURE TO RBBM (*Id.*) The claimed general purpose register blocks are outlined in red on the shader pipeline diagram reproduced below.

	 Indices from CP From VtxGrpTess Vertex Reuse Miss Determination new indices or vert data for upto 64 verts I,J Data Seq Cntrl VDataIn INTRP 3 Verts of Parameters GPR MAC V0 | Q0-UL GPR MAC V1 | Q0-UR GPR MAC V2 | Q0-LL GPR MAC V3 | Q0-LR GPR MAC V16 | Q4-UL GPR MAC V17 | Q4-UR GPR MAC V18 | Q4-LL GPR MAC V19 | Q4-LR GPR MAC V32 | Q8-UL GPR MAC V33 | Q8-UR GPR MAC V34 || Q8-LL GPR MAC V35 || Q8-LR GPR MAC V48 | Q12-UL GPR MAC V49 |Q12-UR GPR MAC V50 | Q12-LL GPR MAC V51 | Q12-LR TFM TAM mac PC OB 64 V0-2 16x1 SP 0 256bits -- 4 half pixels per clk (*See* Ex. 2042, pp. 16, 30.) (*See also id.* at 6 (describing the general purpose registers (GPRs).) The register block maintains the selected one of the

	<u>plurality of inputs.</u> "The general-purpose registers are 128 bits wide, composed of four 32-bit values. (*Id.*) General purpose registers are allocated based on the number of general purpose registers a program needs. (*See id.*; Ex. 2028, p. 39.) "The register file allocation for vertices and pixels can either be static or dynamic." (Ex. 2028, p. 31; *see also id.* at 40.) The figure shown below is an example of the general purpose registers' allocation. (*Id.* at 32.) "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." (*Id.*)

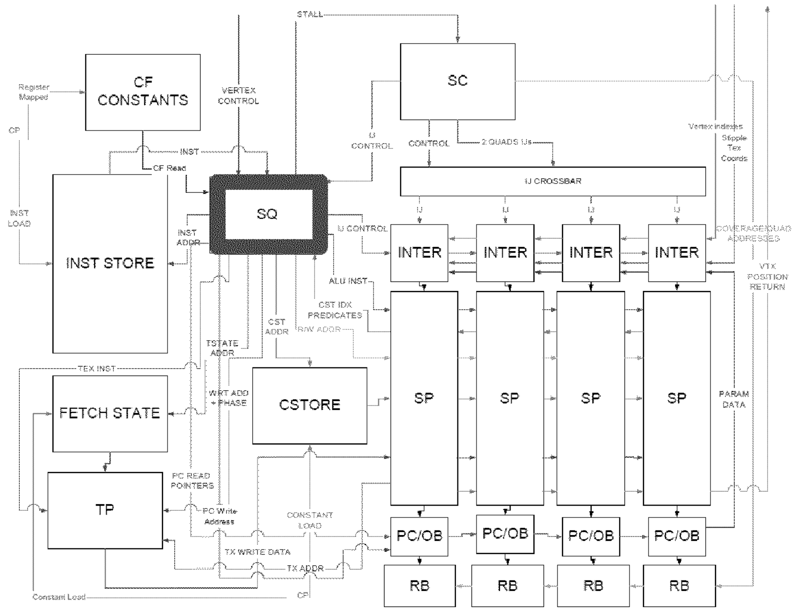
	 (*Id.*) The register block maintains the selected input because as shown above, data is written to the register files. (*See also id.* at 14.) And the data can be written to a pixel portion or vertex portion. (*See id.* at 41.) <u>The shader includes a computation element.</u> The shader includes ALUs, which are computation elements. The shader comprises ALUs. (*See* Ex. 2010, p. 11; *see also* Ex. 2042, p. 7.) "ALU consist of two distinct units: 'Vector' ALU and the

	'Scalar' ALU. The Vector ALU performs operations in parallel across a 4-component vector, while the Scalar ALU performs operations on a single component of a vector which is then replicated across all components." (Ex. 2042, p. 7.) "An ALU can do simple math, conditional moves, and permutations on the registers, and the ability to do a limited number of memory reads using the texture cache." (Ex. 2041, p. 10.) <u>The computation element is operative to perform arithmetic and logical operations on the data maintained in the register block.</u> ALU stands for "Arithmetic and Logic Unit." (Ex. 2042, p. 5.) The R400 specifications list the ALU operations. (*See id.* at 11-13 for a listing of ALU operations). The R400 specifications also list the scalar unit's operations. (*See id.* at 33-42.) The ALUs are operative to perform operations on data maintained in the register block. (*See* Ex. 2028, pp. 24-25,

51-52, 54-58.)

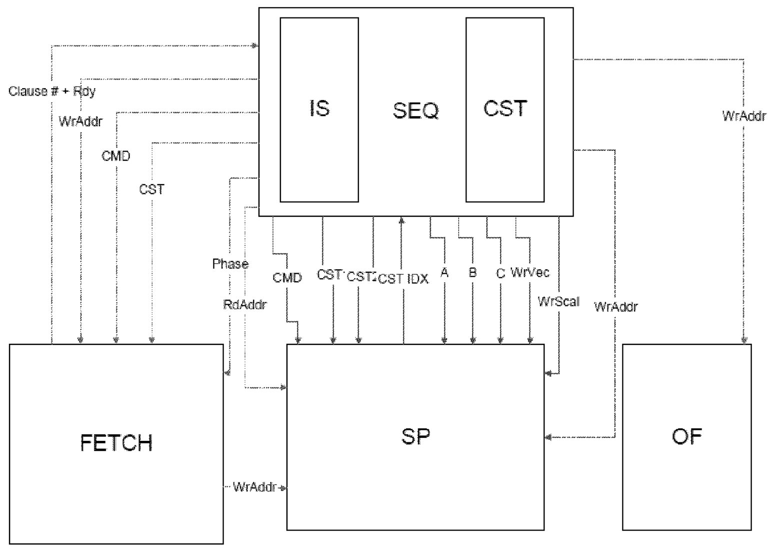
<u>The shader includes a sequencer</u>.

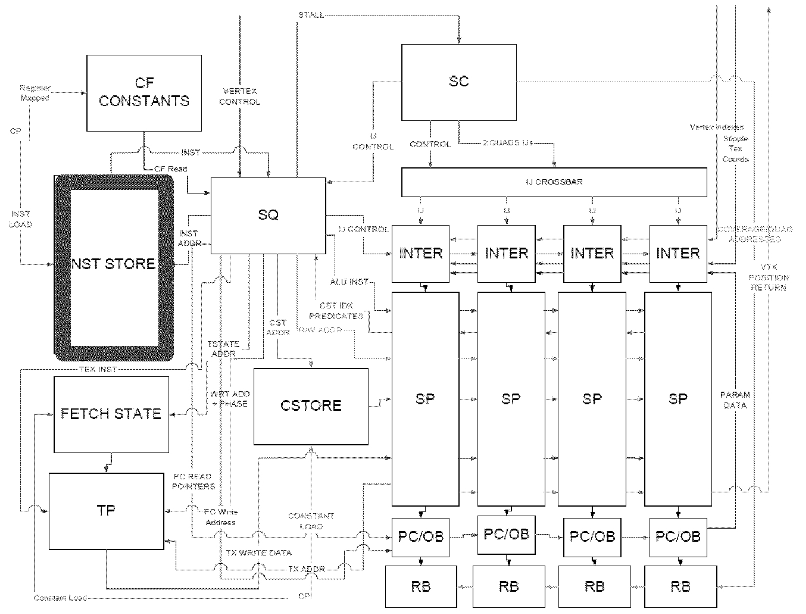
The figure below shows the R400's sequencer block. The sequencer block is outlined in red.



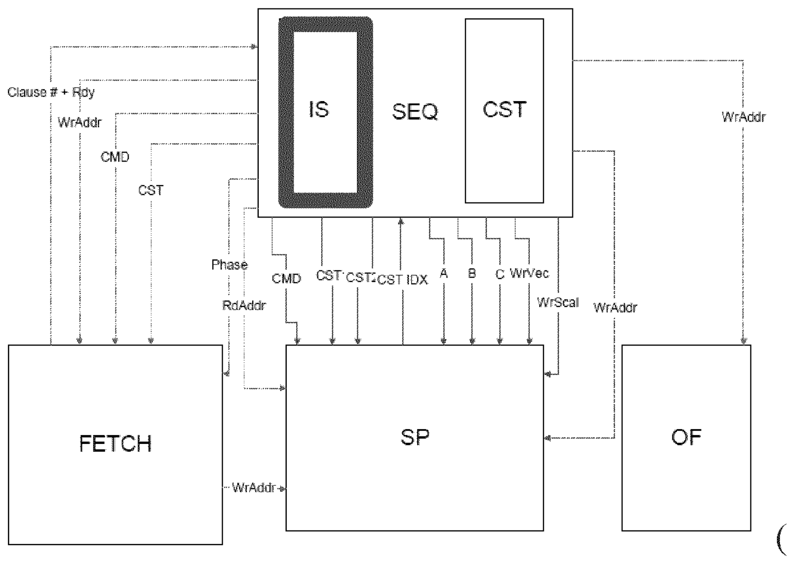
(*See id.* at 7.)

The figure reproduced below shows a sequencer (SEQ), instruction store (IS), and constant store (CST).

	 IS SEQ CST Clause # + Rdy WrAddr CMD CST Phase RdAddr CMD CST CST CST IDX A B C WrVec WrScal WrAddr WrAddr FETCH SP OF WrAddr (*Id.* at 14.) The sequencer is included in the R400 shader architecture. The sequencer maintains the instructions that are executed by the computation element. The sequencer maintains the instructions in an instruction store. The instruction store is outlined in red on the figures reproduced below.

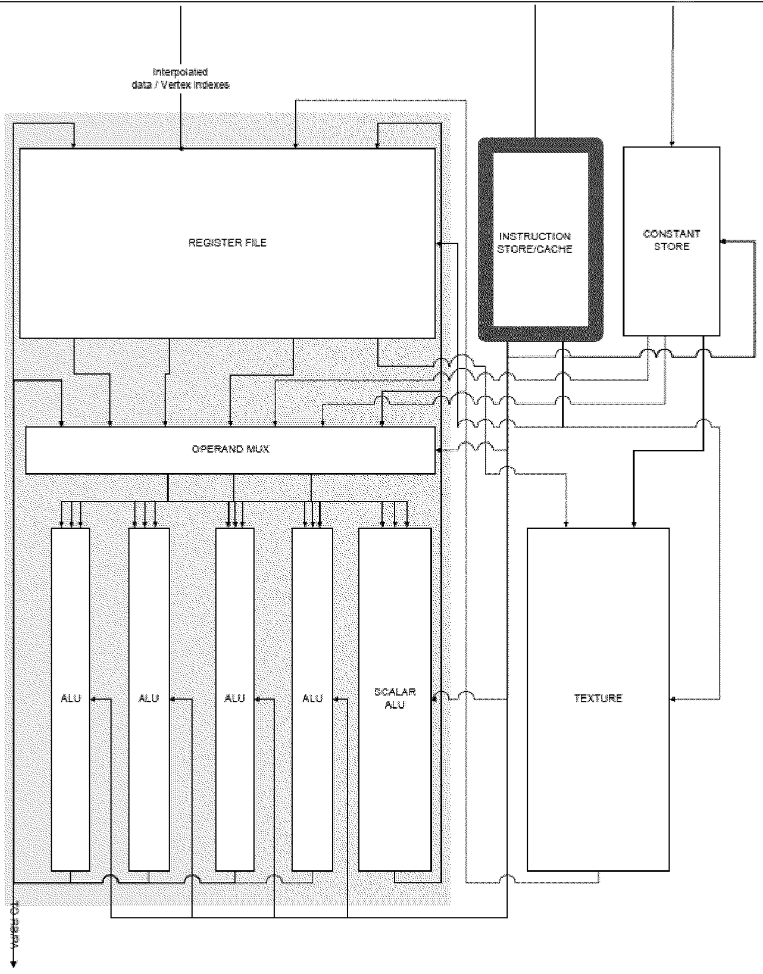


(*See id.* at 7.)

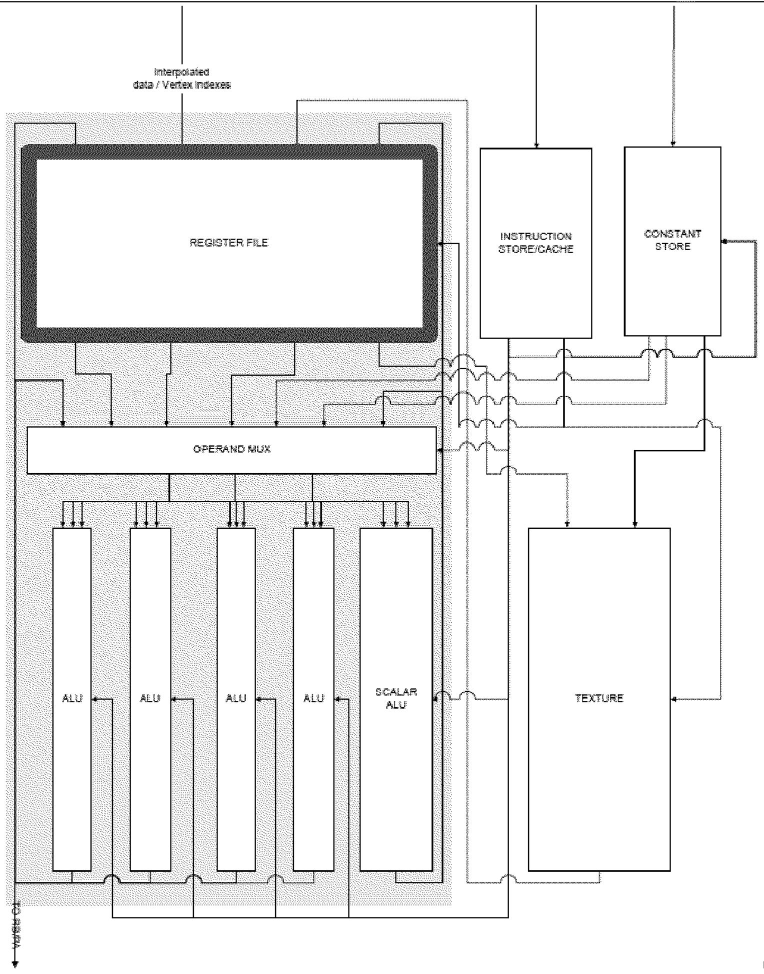


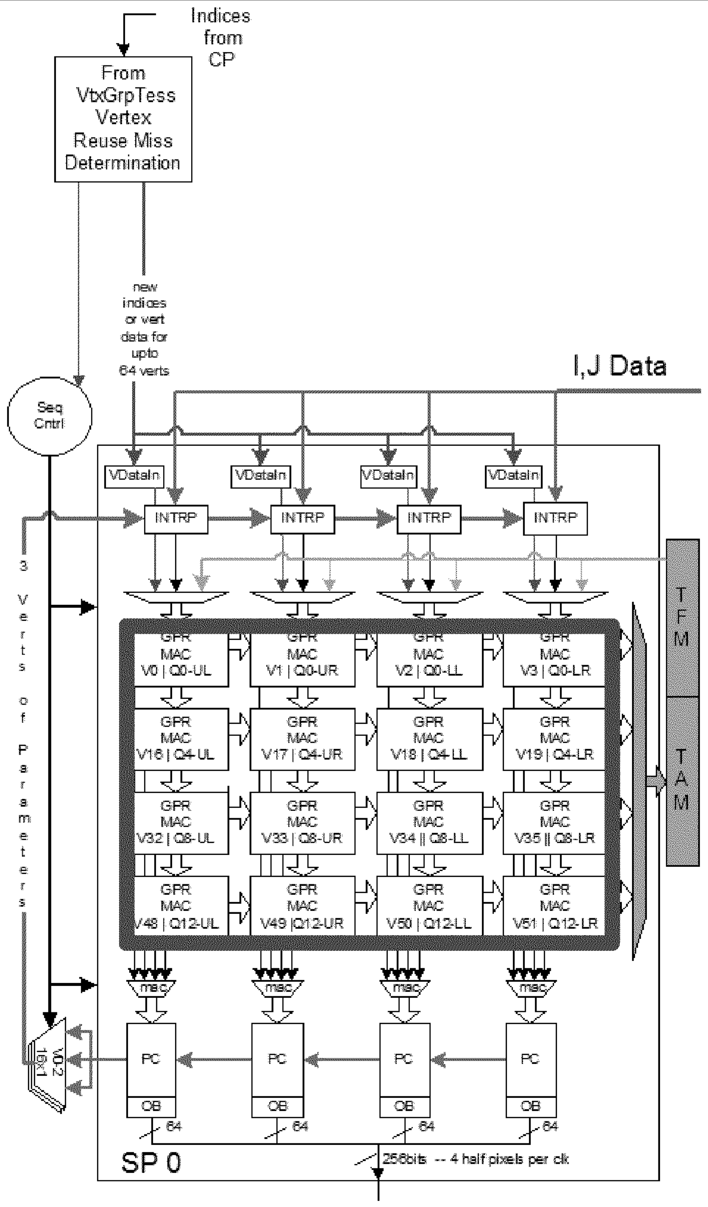
(

Id. at 14.)

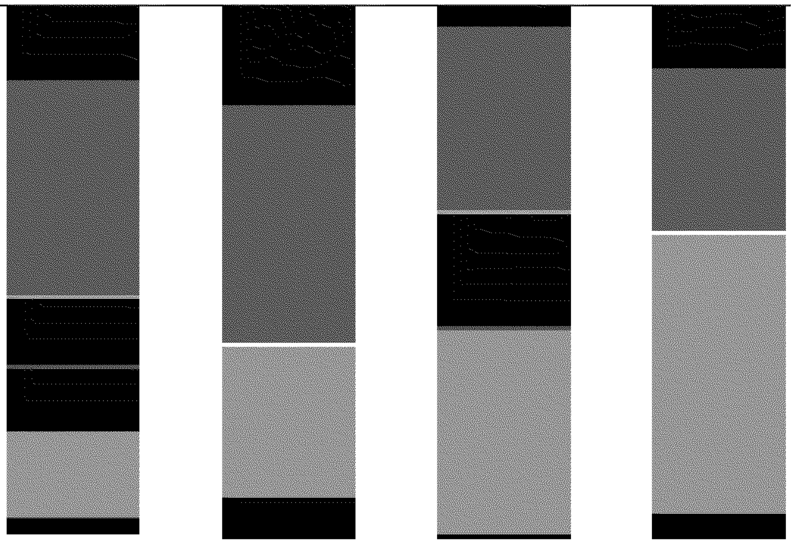
	 Interpolated data / Vertex indexes REGISTER FILE INSTRUCTION STORE/CACHE CONSTANT STORE OPERAND MUX ALU ALU ALU ALU SCALAR ALU TEXTURE TO RBPA (Ex. 2010, p. 11.) “There is going to be only one instruction store for the whole chip. It will contain 4096 instructions . . . . It is likely to be a 1 port memory.” (Ex. 2028, p. 17.)
15. A unified shader, comprising:	“The most ambitious feature in this design is the ‘truly unified pipe’ : a single programmable pipeline is used for 2D Video, 3D vertex, and 3D pixel operations. The unified

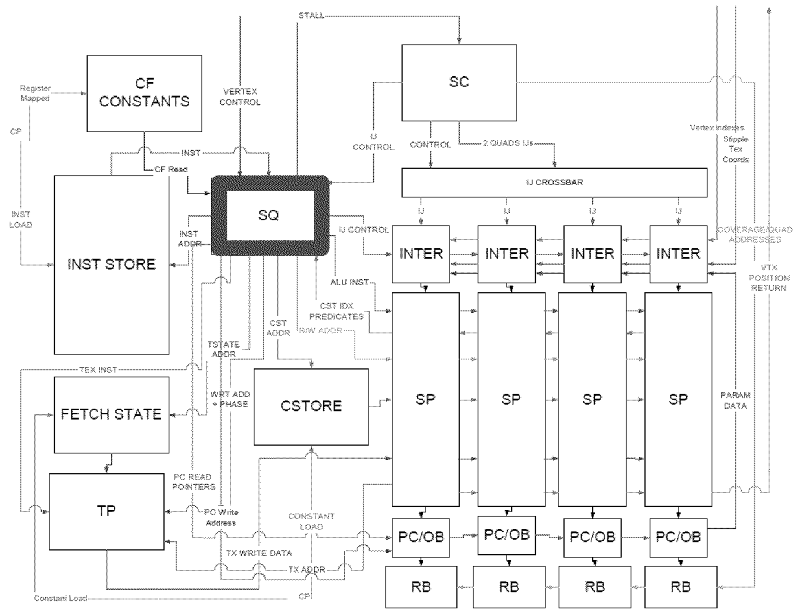
	pipeline does all of its calculations in 32 bit floating point." (Ex. 2041, p. 7.) The unified pipeline results in a single math/register structure compared to the separate structures in a more traditional design." (*Id.*) "[T]he R400 Shader Pipe truly represents an Unified Shader Architecture. In R400, both vertex and pixel shading operations are implemented through the shader units." (Ex. 2042, p. 5.)
15a. a general purpose register block for maintaining data;	<u>The shader includes a general purpose register block.</u> "The user model for the unified shader is composed of a variable number of general purpose registers, a subset of which are usually initialized with data." (Ex. 2041, p. 10.) A shader unit is represented by the gray area on the figure reproduced below. (Ex. 2010, p. 11.) The shader unit includes a register file, outlined in red. (*Id.*)

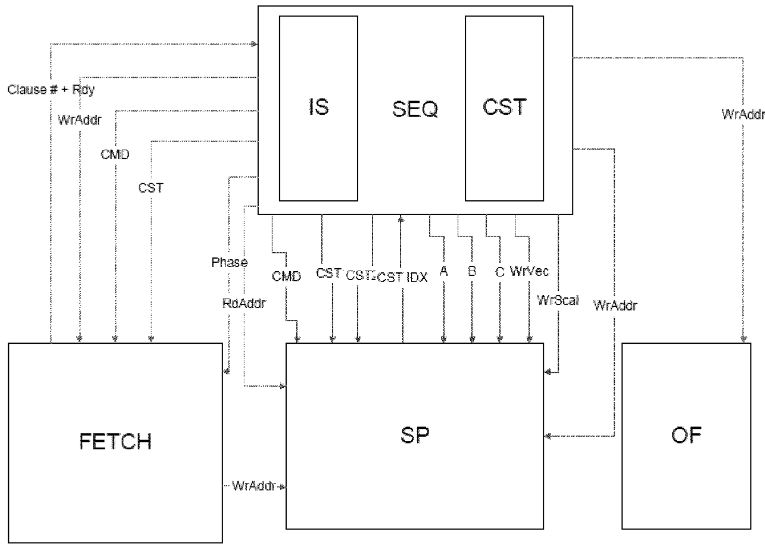
	 Interpolated data / Vertex indexes REGISTER FILE INSTRUCTION STORE/CACHE CONSTANT STORE OPERAND MUX ALU ALU ALU ALU SCALAR ALU TEXTURE TO RB/PA (*Id.*) The claimed general purpose register blocks are outlined in red on the shader pipeline diagram reproduced below.

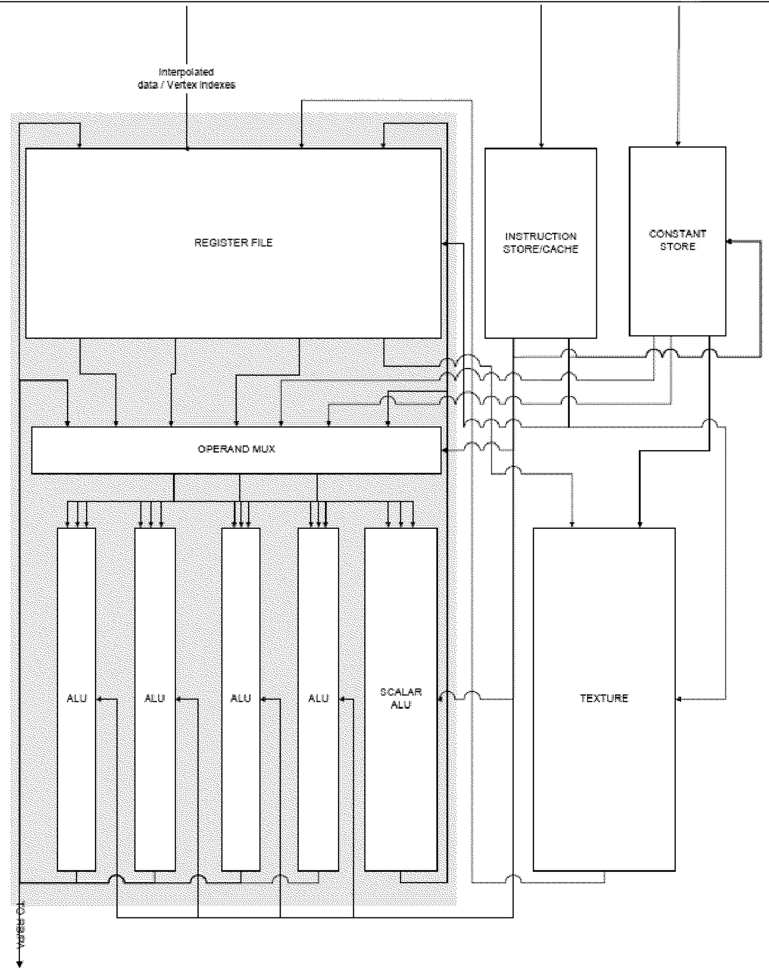
	 Indices from CP From VtxGrpTess Vertex Reuse Miss Determination new indices or vert data for upto 64 verts I,J Data Seq Cntrl VDataIn INTRP 3 Verts of Parameters GPR MAC V0 | Q0-UL GPR MAC V1 | Q0-UR GPR MAC V2 | Q0-LL GPR MAC V3 | Q0-LR GPR MAC V16 | Q4-UL GPR MAC V17 | Q4-UR GPR MAC V18 | Q4-LL GPR MAC V19 | Q4-LR GPR MAC V32 | Q8-UL GPR MAC V33 | Q8-UR GPR MAC V34 || Q8-LL GPR MAC V35 || Q8-LR GPR MAC V48 | Q12-UL GPR MAC V49 |Q12-UR GPR MAC V50 | Q12-LL GPR MAC V51 | Q12-LR TFM TAM mac PC OB 64 V0-2 16x1 SP 0 256bits -- 4 half pixels per clk (*See* Ex. 2042, pp. 16, 30.) (*See also id.* at 6 (describing the general purpose registers (GPRs).) <ins>The general purpose register block maintains data.</ins>

	"The general-purpose registers are 128 bits wide, composed of four 32-bit values. (*Id.*) General purpose registers are allocated based on the number of general purpose registers a program needs. (*See id.*; Ex. 2028, p. 39.) "The register file allocation for vertices and pixels can either be static or dynamic." (Ex. 2028, p. 31; *see also id.* at 40.) The figure shown below is an example of the general purpose registers' allocation. (*Id.* at 32.) "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." (*Id.*)

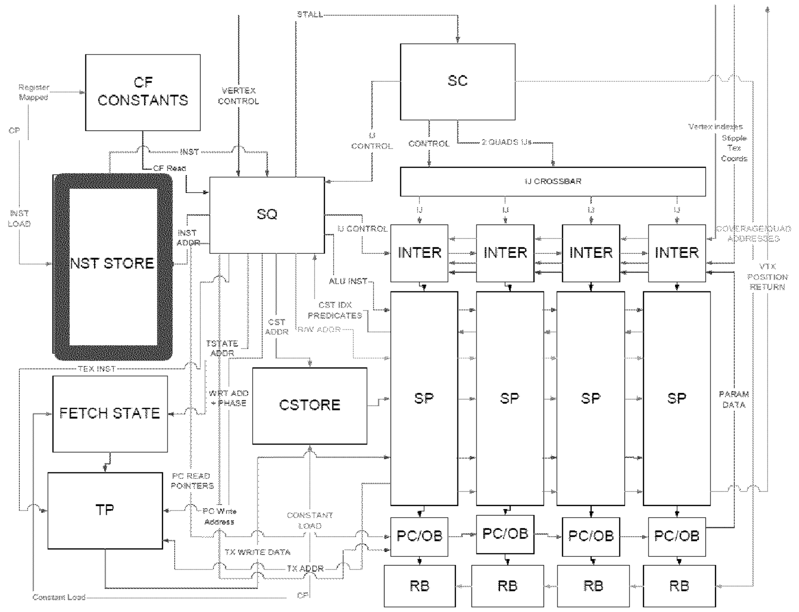
	 (*Id.*) As shown above, data is written to the register files. (*See also id.* at 14.) And the data can be written to a pixel portion or vertex portion. (*See id.* at 41.)
15b. a processor unit; and	<u>The shader includes a processor unit.</u> An ALU of the shader is the claimed processor unit. The shader comprises ALUs. (*See* Ex. 2010, p. 11; *see also* Ex. 2042, p. 7.) "ALU consist of two distinct units: 'Vector' ALU and the 'Scalar' ALU. The Vector ALU performs operations in parallel across a 4-component vector, while the Scalar

	ALU performs operations on a single component of a vector which is then replicated across all components." (Ex. 2042, p. 7.) "An ALU can do simple math, conditional moves, and permutations on the registers, and the ability to do a limited number of memory reads using the texture cache." (Ex. 2041, p. 10.)
15c. a sequencer, coupled to the general purpose register block and the processor unit, the sequencer maintaining instructions operative to cause the processor unit to execute vertex calculation and pixel calculation operations on selected data	<u>The shader includes a sequencer.</u> The figure of the R400 architecture, reproduced below, shows a sequencer highlighted in red.  STALL CF CONSTANTS VERTEX CONTROL SC Register Mapped CP INST CF Read IJ CONTROL CONTROL 2 QUADS IJs Vertex indexes Stipple Tex Coords IJ CROSSBAR INST LOAD SQ INST ADDR IJ INTER COVERAGE/QUAD ADDRESSES VTX POSITION RETURN INST STORE ALU INST CST IDX PREDICATES R/W ADDR CST ADDR TSTATE ADDR TEX INST FETCH STATE WRT ADD + PHASE CSTORE SP PARAM DATA TP PC READ POINTERS PC Write Address CONSTANT LOAD TX WRITE DATA TX ADDR Constant Load CP PC/OB RB (*See* Ex. 2028, p. 7.)

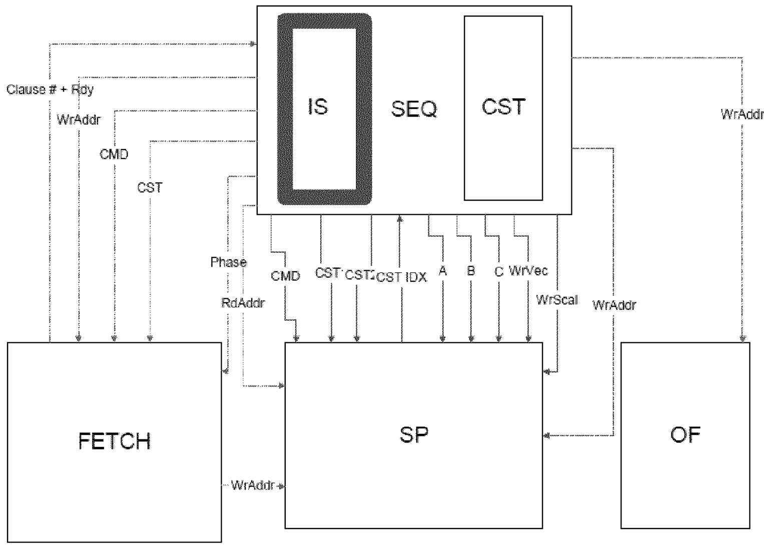
maintained in the general purpose register block.	The figure reproduced below shows a sequencer (SEQ), instruction store (IS), and constant store (CST).  IS SEQ CST Clause # + Rdy WrAddr CMD CST Phase RdAddr CMD CST CST CST IDX A B C WrVec WrScal WrAddr WrAddr FETCH SP OF WrAddr (*Id.* at 14.) The sequencer is included in the R400 shader architecture. <u>The sequencer is coupled to the general purpose register block.</u> As shown in the figures above, the sequencer is coupled to the shader pipe and as shown in the figure below, the sequencer's instructions tore and constant store are coupled to the shader's register file.

	 Interpolated data / Vertex indexes REGISTER FILE INSTRUCTION STORE/CACHE CONSTANT STORE OPERAND MUX ALU ALU ALU ALU SCALAR ALU TEXTURE TO RBBM (Ex. 2010, p. 11.) <u>The sequencer is coupled to the processor unit.</u> As shown in the figures reproduced above, the sequencer is coupled to the shader pipe and thereby also coupled to the shader's ALU processing units that receive instructions from the sequencer to process data. <u>The sequencer maintains instructions.</u>

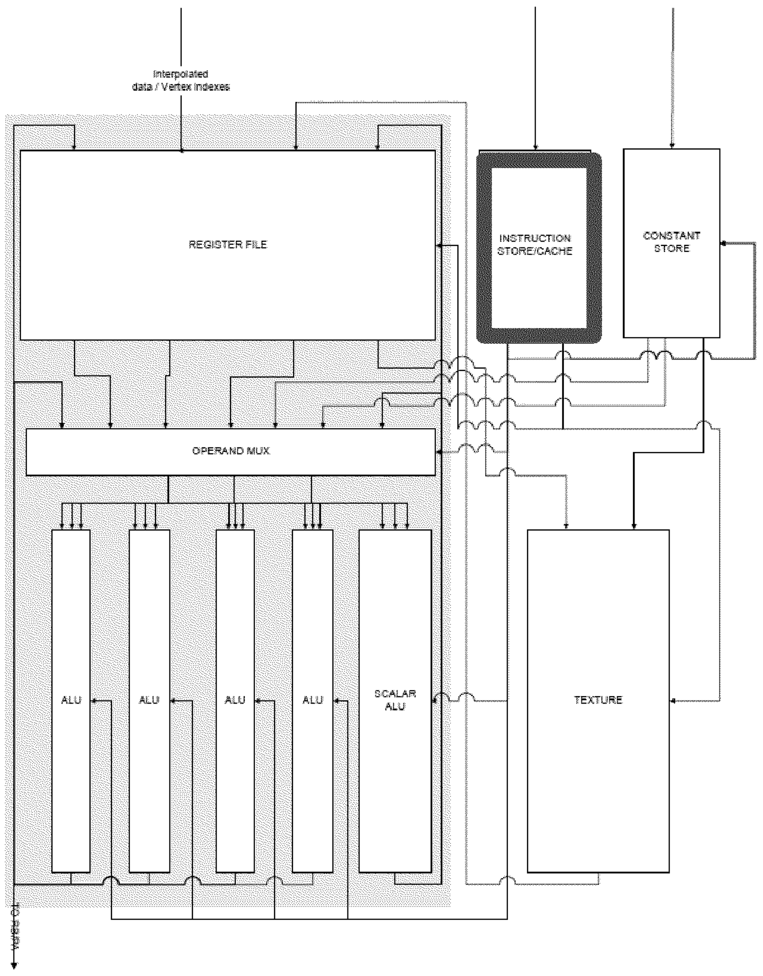
The sequencer maintains the instructions in an instruction store. The instruction store is outlined in red on the figures reproduced below.



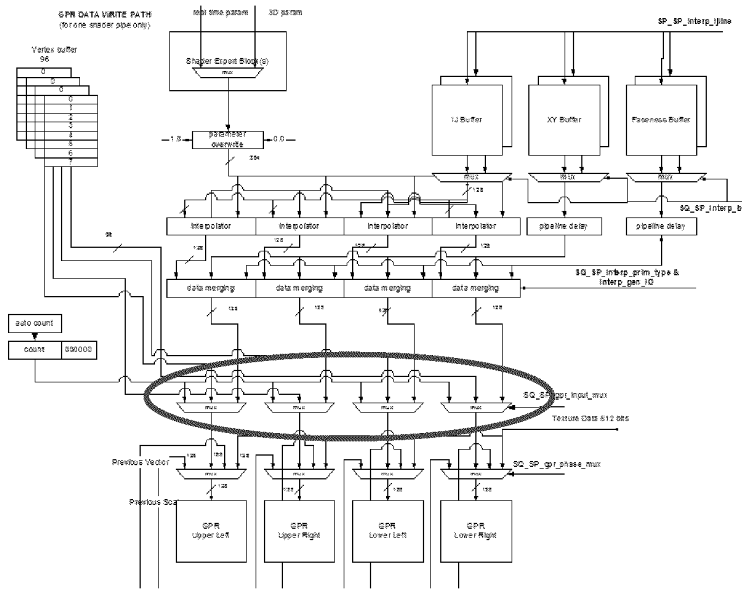
(*See* Ex. 2028, p. 7.)

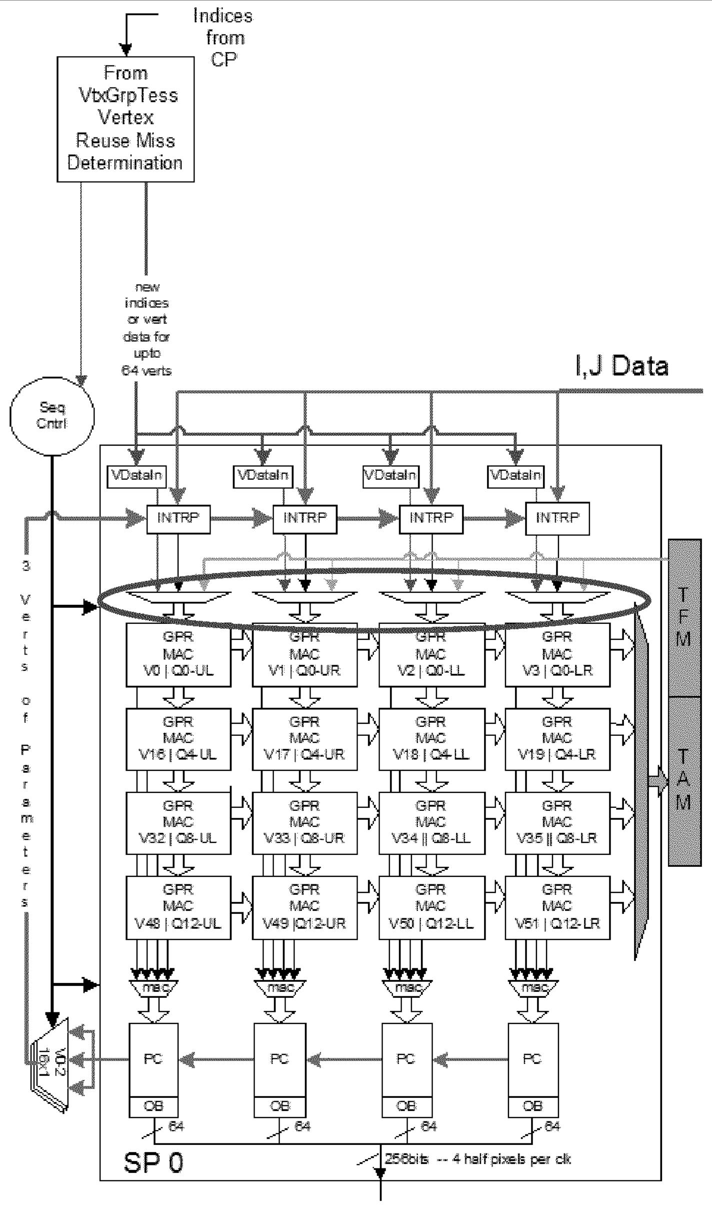


(

	Id. at 14.)  Interpolated data / Vertex indexes REGISTER FILE INSTRUCTION STORE/CACHE CONSTANT STORE OPERAND MUX ALU ALU ALU ALU SCALAR ALU TEXTURE (Ex. 2010, p. 11.) "There is going to be only one instruction store for the whole chip. It will contain 4096 instructions . . . . It is likely to be a 1 port memory." (Ex. 2028, p. 17.) <u>The instructions are operative to cause the processor unit to execute vertex calculation and pixel calculation</u>

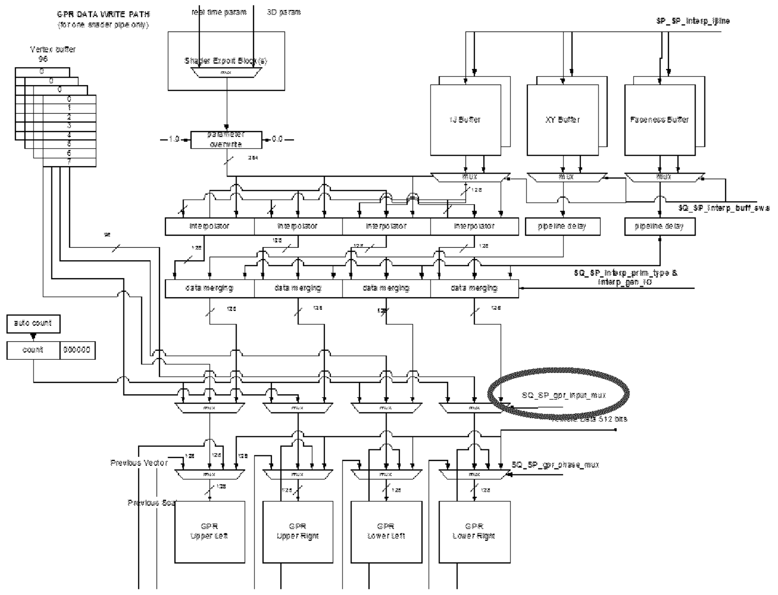
	operations on selected data maintained in the general purpose register block. *See supra* Claim 15b (showing support for the processor unit). The shader includes ALUs, which perform both vertex calculation and pixel calculation operations because the "Shader Pipe (SP) serves as the central Arithmetic and Logic Unit (ALU) for the R400 Graphics Processor, and "both vertex and pixel shading operations are implemented through the shader units." (*See* Ex. 2042, p. 5.) The ALUs are operative to perform the operations on selected data maintained in the general purpose register block. (*See* Ex. 2028, pp. 24-25, 51-52, 54-58.) "An ALU can do simple math, conditional moves, and permutations on the registers, and the ability to do a limited number of memory reads using the texture cache." (Ex. 2041, p. 10.) The instructions are operative to cause the ALUs to execute the operations. "The sequencer chooses two ALU threads and a fetch hread [sic] to execute, and executes all

	of the instructions in a block before looking for a new clause of the same type. Two ALU threads are executed interleaved to hide the ALU latency." (Ex. 2028, p. 6.)
17. The shader of claim 15, further including a selection circuit operative to provide information to the general purpose block in response to a control signal.	<u>The shader includes a selection circuit.</u> The at least one GPR input multiplexer is the claimed selection circuit. The GPR input multiplexers are circled in red on the diagram below. "The diagram below shows all the possible data paths going into the GPR write paths, their selection, and routing."  GPR DATA WRITE PATH (for one shader pipe only) real time param 3D param SP_SP_interp_ijline Vertex buffer 96 Shader Export Block(s) mux parameter overwrite 1.0 0.0 IJ Buffer XY Buffer Faceness Buffer SQ_SP_interp_buff_swap Interpolator pipeline delay data merging SQ_SP_interp_prim_type & interp_gen_IO auto count count 000000 SQ_SP_gpr_input_mux Texture Data 512 bits Previous Vector Previous Scalar SQ_SP_gpr_phase_mux GPR Upper Left GPR Upper Right GPR Lower Left GPR Lower Right (*See* Ex. 2042, p. 29.) The figure below also shows the GPR input multiplexers.

	 Indices from CP From VtxGrpTess Vertex Reuse Miss Determination new indices or vert data for upto 64 verts I,J Data Seq Cntrl VDataIn INTRP 3 Verts of Parameters TFM TAM GPR MAC V0 | Q0-UL GPR MAC V1 | Q0-UR GPR MAC V2 | Q0-LL GPR MAC V3 | Q0-LR GPR MAC V16 | Q4-UL GPR MAC V17 | Q4-UR GPR MAC V18 | Q4-LL GPR MAC V19 | Q4-LR GPR MAC V32 | Q8-UL GPR MAC V33 | Q8-UR GPR MAC V34 | Q8-LL GPR MAC V35 | Q8-LR GPR MAC V48 | Q12-UL GPR MAC V49 |Q12-UR GPR MAC V50 | Q12-LL GPR MAC V51 | Q12-LR mac PC OB V0-2 16:1 64 SP 0 256bits -- 4 half pixels per clk (*See id.* at 16, 30.) <u>The selection circuit is operative to provide information to the general purpose block in response to a control signal.</u> The GPR input multiplexers of the arbiter circuit provide

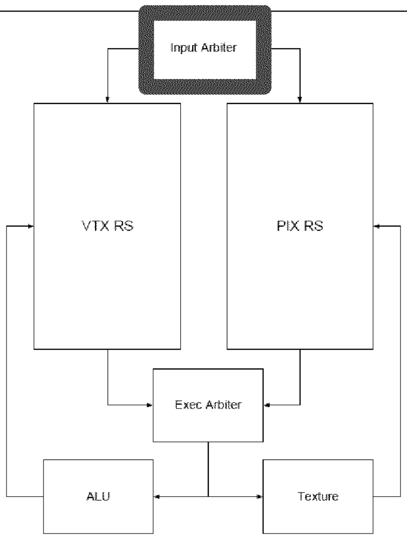
information to the GPRs in response to the SQ_SP_gpr_input_mux control signal.

The GPR input multiplexers are "controlled by SQ_SP_gpr_input_mux part of the 'SQ_SP: Interpolation bus' interface . . . to route between vertex data/indices and interpolated pixel parameters." (*Id.* at 28.) The claimed control signal is sent along this interface. This interface is circled in red on the figure below.



As shown in the figures above, the output from the selection circuit is provided to the general purpose registers.

18. The shader of claim 17, wherein the selection circuit is a multiplexer and the control signal is provided by an arbiter.	*See supra* Claim 17 (showing support for the selection circuit). <u>The selection circuit is a multiplexer.</u> The GPR input selection circuits shown in the figures above are multiplexers. The R400 specifications call a GPR input a "multiplexer" and a "MUX." (*See id.* at 28, 30; Ex. 2028, p. 40.) <u>An arbiter provides the control signal.</u> *See supra* Claim 17 (showing support for the control signal). The claimed arbiter is the input arbiter outlined in red on the figure below.

	 Input Arbiter VTX RS PIX RS Exec Arbiter ALU Texture (Ex. 2028, p. 10.) The arbiter provides the SQ_SP_gpr_input_mux control signal. "The control of all the multiplexers present at the input and output of the GPRs . . . is done by the sequencer." (*See* Ex. 2042, p. 30.) And the sequencer's input arbiter "first arbitrates between vectors of . . . vertices . . . and vectors of . . . pixels." (*See* Ex. 2028, pp. 6, 10.)
20. The shader of claim 15, wherein the processor unit executes vertex calculations while the	*See supra* Claim 15b (showing support for the processor unit). <u>The processor unit executes vertex calculations while the pixel calculations are still in progress.</u>

pixel calculations are still in progress.	While waiting for texture data, the pixel calculations can stall while the vertex calculations are in progress, and in such an instance, the vertex calculations can execute while pixel calculations are still in progress. The stall occurs because vertex calculations have priority. "[T]he vertex stream . . . has priority over the pixel stream," so, when pixel data reaches the rasterizer, a pixel calculation may wait until other vertex calculations are completed. (*See* Ex. 2041, p. 11.) The R400 specifications describe state management, which tracks the progress of pixel and vertex operations that execute at the same time. (*See id.* at 13.)

XII. OVERVIEW OF THE APPLIED REFERENCES FOR GROUNDS 1-4

276. In this section, I provide an overview of Rich and Kurihara. This overview is relevant for my comparison of these references with the '871 patent.

A. Rich

277. Rich discloses an image-generation system that has a sequential, pipelined architecture. This system performs four functions: "geometric

processing, rasterization, shading/texturing and composition." Ex. 1005, Rich, 8:61-62. Rich's system performs these functions in discrete, sequential phases.

278. The "first function" that Rich's system performs is vertex operations. *Id.* at 9:1-2; *see also id.* at 5:3-4. During this vertex-processing phase, primitives are assigned to specific processing elements 32. *Id.* at 9:5-7, 14:45-48, 15:43-47, 16:44-47. These assigned processing elements 32 may then perform one of three different vertex operations: (i) transform primitives from model coordinates to screen coordinates; (ii) determine lighting values for the primitives; or (iii) generate linear coefficients of the primitives. *Id.* at 9:18-25; *see also id.* at 14:10-18-27 (describing the "[c]onversion from model to screen coordinates").

279. After the vertex operations, Rich's "processing elements 32 write the list of transformed primitives to external memory." *Id.* at 9:27-29. According to Rich, "[t]he use of the external memory circuit may be necessitated by the fact that the processing elements 32 have only a small amount of memory 34 in their own dedicated circuitry." *Id.* at 16:52-55; *see also id.* at 17:60-64. Rich's vertex processing is "complete" when all the vertex data has been written to external memory. *Id.* at 9:36-39.

280. After the vertex data is written to external memory, Rich's system performs rasterization and pixel operations. *Id.* at 9:40-10:5. During the pixel

operations, each processing element 32 is assigned to process pixels within a unique region of a computer screen display. *Id.* at 8:32-40, 9:43-46. In particular, the pixel operations begin when each processing element 32 has been assigned pixel information to process. *Id.* at 10:2-5; *see also id.* at 9:61-64 (defining the pixel information as a "contribution").

281. Each processing element 32 in Rich is assigned a small number of "home pixels." The processing element is responsible for calculating the final value of a home pixel by combining "contribution values." This is done by repeatedly blending the colors of each contribution value with the prior values. *See id.* at 4:46-49; 8:33-40; 9:40-41; 10:35-37.

282. In other words, Rich's pixel processing occurs *only after* the completion of the vertex processing. And the processing element 32 that operates on a piece of data during the vertex-processing phase is not necessarily the same processing element 32 that operates on that data during the pixel-processing phase. So, to make the output of the vertex-processing phase accessible to the processing elements 32 during the pixel-processing phase, Rich teaches that this output (i.e., the transformed vertex data) is stored in an external, shared memory—*not* in the dedicated memory of a single processing element 32.

283. Additionally, it is important to understand that Rich discloses an extremely restrictive architecture that is quite different from a general purpose processor or a modern GPU. In the first place, Rich teaches a SIMD array with 256 or 1024 processing elements (PEs). This means that all of the PEs must run the exact same instruction every cycle. *Id.* at 7:19-23; 17:28-42; 18:19-28. Moreover, if a PE is to read or write its corresponding PE memory 34 block, the location to be accessed is specified in that same shared instruction which means that in general, every PE must access the same memory address (or corresponding addresses in quadrants of the memory) at the same time. *Id.* at 33:14-47. This provides very little programming flexibility and is configured to support addressing of pixels which naturally can be arranged in quadrants.

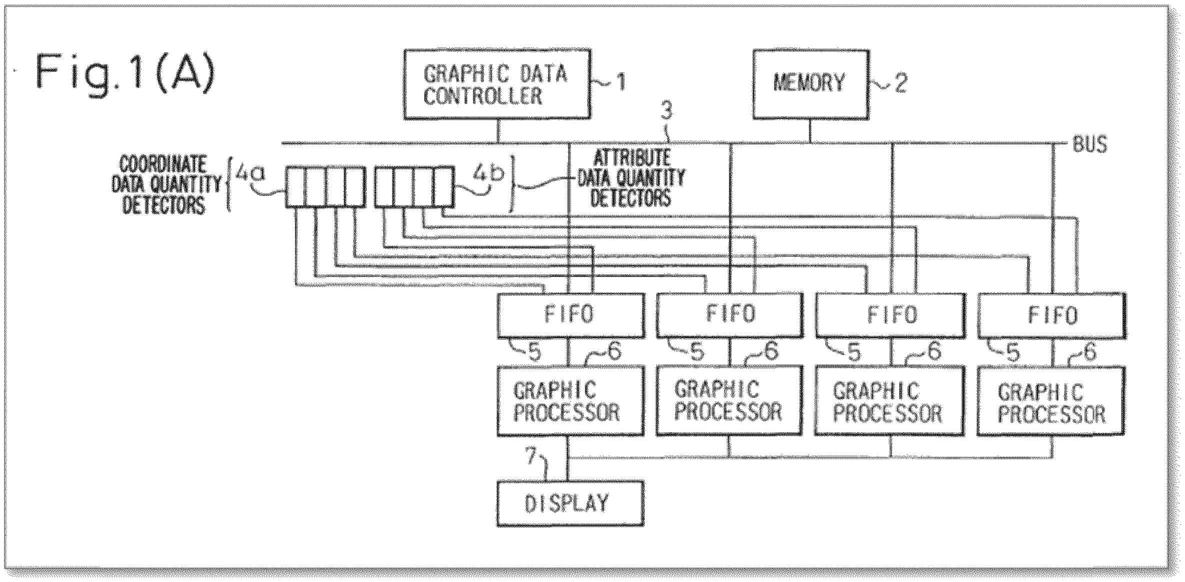
284. The PE memories are also extraordinarily small. The total memory size in each PE is only 128 bytes. This would be very ineffective at storing vertices as the vertex information for a single triangle is generally 48 bytes. Moreover, the PE itself cannot direct transfers of data between PEs or from the host memory or video memory to the PE. *See e.g. id.* 32:2-13. In order to support transfers into and out of PE memory 34, this already small memory is partitioned into a main section and an overflow section each used for different purposes. *See e.g. id.* 34:12-51.

285. Each PE also only has very limited calculating ability. This means that the floating-point calculations used for vertex operations like transform and lighting are very slow. Each PE only includes an 8-bit integer ALU (which can add and subtract) and no specific circuit for multiplication. *See e.g. id.* 30:56-31:37. In my class, I have my students measure how long it takes to perform floating-point arithmetic using commercially-available software on a more powerful 8-bit processor that includes more registers and a multiplier. I have also performed these measurements myself. On that more powerful 8-bit processing element, it takes approximately 110 cycles to perform a 32-bit floating-point add operation and 140 cycles to perform a 32-bit floating-point multiply. One would expect a multiply on Rich to take even longer. A transform calculation on one vertex requires 9-16 multiplications and 9-12 additions. This means that performing just the transform operation (and not lighting) on the Rich PE likely requires at least 2250-3560 steps. Given the long amount of time that the vertex calculation takes, the amount of time it takes to get the vertex data from memory is not significant to performance.

B. Kurihara

286. Kurihara is directed to a graphics-processing system that includes a plurality of graphics processors 6, as shown in Kurihara's Figure 1(A) (reproduced below). Each graphics processor is coupled to a corresponding FIFO 5. The FIFO

memories store graphics data to be processed in parallel by the graphics processors 6. Kurihara, 4:38-40. The graphics data in the FIFO memories can be either vertex (coordinate) data or pixel (attribute) data. *Id.* at 4:56-65. These FIFO memories then simultaneously transfer the data to the graphics processors 6. *Id.* at 4:61-62. The graphics processors 6 then process the graphics data in parallel. *Id.* at 5:37. In other words, in Kurihara's system, each graphics processor 6 performs one type of graphics-processing operation on the data type that it receives from the corresponding FIFO memory 5.



287. It's also important to note what Kurihara does not disclose. First, Kurihara has no disclosure of interleaving a pixel operation and a vertex operation. Second, Kurihara does not disclose that one type of operation can be started before

another type of operation has finished. These points are important for my analysis below.

XIII. GROUNDS 6 AND 9: OBVIOUSNESS GROUND BASED ON RICH AND KURIHARA

288. I understand that the Board instituted trial for claim 15 as allegedly obvious over Rich. I also understand that the Board instituted trial for claim 20 as allegedly obvious over the combination of Rich and Kurihara.

289. In my opinion, these claims are patentable over these references. *First*, for claim 15, a POSA would not have been motivated to modify Rich to store vertex data in an on-chip memory, as LG proposes. *Second*, in my opinion, Kurihara does not teach or suggest a single "processor unit" that "executes vertex calculations while the pixel calculations are still in progress," as in claim 20. *Third*, I discuss so-called objective indicia that, in my opinion, show that the "unified shader" of claims 15 and 20 is not obvious.

A. A POSA would not have modified Rich in the way that LG and Dr. Bagherzadeh propose.

290. In my opinion, claim 15 is not obvious in view of Rich's sequential, pipelined system. This claim is directed to "[a] unified shader," comprising:

> a general purpose register block for maintaining data;
> a processor unit; and

a sequencer, coupled to the general purpose register block and the processor unit, the sequencer maintaining instructions operative to cause the processor unit to execute *vertex calculation and pixel calculation operations on selected data maintained in the general purpose register block*.

I understand that LG acknowledges that Rich does not explicitly disclose the italicized feature. *See* Pet. at 47. And, in my opinion, a POSA would not have been motivated to make the modifications to Rich that LG proposes.

291. I understand that LG maps Rich's system to claim 15 in the following way: (i) Rich's on-processor memory 34 and registers allegedly correspond to the claimed "general purpose register block"; (ii) Rich's ALU 33 allegedly corresponds

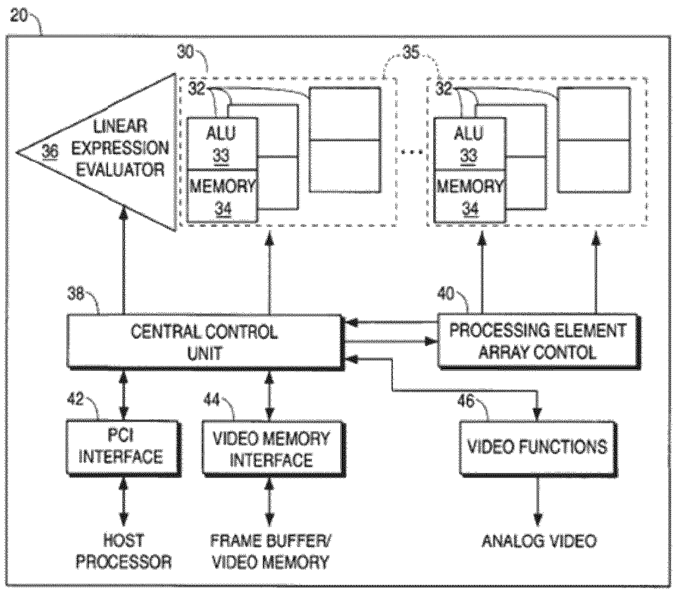


FIG. 2

to the claimed "processor unit"; and (iii) Rich's processing element array control 40 allegedly corresponds to the claimed "sequencer." *Id.* at 47. And I understand that, according to LG and Dr. Bagherzadeh, it would have been obvious to modify

Rich to store vertex data and pixel data in on-chip memory 34. *See id.* at 48. I disagree.

292. In the first place, Rich explicitly teaches that vertex data is to be stored in an external memory. It is clear to a POSA that this is not an arbitrary decision in Rich but rather that is necessitated by the Rich architecture that aims to keep each PE very simple and small with limited capabilities. In fact, when discussing primitive data, Rich explains that the use of external memory requires a relatively large memory storage circuit and that the use of external memory may be necessitated. He also teaches that the memory identified by LG as the alleged general purpose register block is quite small.

> Each processing element is assigned to one specific primitive which has associated with it a primitive specific applicability word.
>
> All of the processing elements 32 are electrically connected to a common communications bus and to ***a relatively large memory storage circuit.*** This connection may be established through the central control unit 38 and the video memory interface 44 or PCI Interface 42. ***The use of the external memory circuit may be necessitated by the fact that the processing elements 32 have only a small amount of memory 34 in their own dedicated circuitry***.

Rich, 16:45-55 (emphasis added).

> [T]he processing elements 32 write the list of transformed primitives to external memory as seen in block 53.

Id. at 9:28-30.

> Because the screen is divided into a number of regions, a list for each region is generated which lists the primitives which touch that region. This list is written to external memory as seen in block 54.

Id. at 9:33-36.

> The processing elements 32 assign starting memory locations for their working space within the external memory in accordance with the precise requirements of processing their assigned primitives.

Id. at 16:58-65.

> The plurality of processing elements 32 examine the nature of the specific primitive to which they have been assigned, and determine the amount of memory required for use in external memory in the process of calculating and storing the transformed primitive.

Id. at 17:56-60.

293. A POSA would not consider moving the large data structures of vertices (described as a list or a database in Rich) into the very small (128 byte) local memories in each PE. In the first place, this would require redesigning these data structures since only a tiny portion of the list or database could be stored in a PE. Also, since Rich is a SIMD architecture with shared instructions, the data stored in a specific location in each PE memory must be used in exactly the same way. Rich does not disclose any way to do this. In the second place, the small memories in Rich would be understood to be configured specifically to support one specific type of pixel operation which I explain below. The performance advantages of this memory are tied to this pixel operation.

294. Additionally, a POSA would not increase the size of the PE memories in Rich. Rich emphasized having many simple PEs rather than large memories. *See e.g. id.* at FIG. 12. The size of the PE memory is hardcoded into the instruction architecture in Rich to use a 7-bit address. *Id.* at 33:15-34. It also uses memory address specific bits for specific purposes during the combining phase. *Id.* at 34:39-51. Modifying the memory size would require redesigning the instructions, expanding the instruction memory, redesigning the control signals for combining, and distributing extra address bits from the instruction to 1000 or more PEs. This would not be simple to do in Rich.

295. In my opinion, the reason that Rich stores pixel data in PE memories is to facilitate a "combination" operation for home pixels assigned to each PE. This operation repeatedly applies a color-blending calculation to the same data and thus benefits substantially from storing a small amount of data in a local memory. Additionally, the local memories are used to gather pixel contributions from neighboring PEs that share a local bus. They have direct access to a shared bus 258 provided for this purpose and have been designed to allow PE to PE pixel data transfers at the same time as other calculations are taking place in the PE. *Id.* at 7:30-32, 25:18-20, 32:3-14. No similar motivations are present for storing vertex calculation data locally. The disclosed vertex calculations (transformation, lighting) are performed once for each vertex in the scene.

> The contribution values are then returned to the processing element assigned to the home pixel corresponding to the contribution and combined to provide a final pixel value.

Id. at 4:46-49; s*ee also id.* at 8:33-40.

> A final pixel value is then created by a combination of contribution values associated with a given pixel.

Id. at 9:40-41.

> The composition function begins in block 75 where contribution values are combined for the home pixels in the processing elements 32 to give the final RGB pixel value.

Id. at 10:35-37.

> Each PE has its own memory resource and a bus structure which allows for sharing of data between processing elements 32.

Id. at 12:38.

> Objects which are made up of primitives are processed by transforming each primitive separately.

Id. at 16:9-10.

> Each processing element is assigned to one specific primitive which has associated with it a primitive specific applicability word.

Id. at 16:45-47.

296. Additionally, since the PE memories are very small, they would not be very effective for storing vertices. While the 128 memories in each PE in Rich might reasonably hold up to 64 pixels using the predominant format at the time of the filing, it would only be able to hold 8-10 vertex positions. Storing so few vertices in a local memory would be unlikely to improve performance in any

situation. *See id.* at 7:27-30, 14:42-15:8. More importantly, Rich does not even hint at any calculation in his system that would benefit from storing a small number of vertices in this memory. In fact, if this memory is used at all during vertex calculations, it appears to be used in its entirety for the creation of the transformation matrix and not to store vertex data. *See id.* at 16:66-18:27.

297. Rich also disclaims any ability for the PE memory to hold both pixel and primitive data at the same time. Instead, Rich clearly explains that there is not room in each PE to store the vertex (primitive) data and thus it may be necessary to provide it to the PE multiple times during the calculations.

> Because a primitive may have an interior which requires a contribution from more than one home pixel for a particular processing element 32 it may be necessary to repeatedly provide to the processing elements 32 the primitives for a region. Thus, in the present case where there are 4 home pixels assigned to each processing element it may be necessary to provide a primitive to the processing element array 30 4 times if one processing element has a contribution for all 4 of its home pixels. In such a case the primitive is evaluated a subregion at a time.

Id. at 11:25-33.

298. In my opinion, LG has mischaracterized Rich in its petition and reached errant conclusions about the impact of the proposed modifications. In the first place, the representation that Rich suggests that databases containing vertex data may be stored in PE memory 34 is unfounded. *See* Pet. at 47. Although petitioners refer to PE memory 34 as "local processor memory 34," Rich never uses this term and does not refer to the PE memory as "local memory." Moreover, the section of the Rich specification cited for support (Rich, 9:1-12) discusses only the operation of block 50 of Figure 3. Block 50 reads "HOST UPDATES THE DATABASE OF PRIMITIVES." It is discussing the operation of the host computer and not the PEs. Therefore, a POSA would read this portion of Rich to most likely refer to memory that is local to the host processor and not to the PE. Rich always refers to memory 34 as PE memory, and not as local memory.

299. LG agrees that Rich does not disclose storing vertex data in PE memory 34, and LG and its expert, Dr. Bagherzadeh, provide only a single reason to modify Rich to allow storing of vertex data in PE memory 34. *See* Pet. at 47; Ex. 1003, ¶216. That alleged modification is that Rich could "maintain or temporarily store both primitive and vertex data from database in the processor memory 34 for the purpose of local access by the ALU unit to process and transform primitives and their vertices." *See* Pet. at 47. As noted above, PE memory 34 is very small and has no additional room for storing vertex data.

300. Moreover, the alleged reason for this modification is "for the purpose of faster and more efficient access because storing of temporary data just before processing operations will reduce the stall time required when data is directly accessed from external memory." This statement represents a misunderstanding of Rich. The PEs in Rich do not request vertices from memory. If they did, such a request might involve stall time as the PE would wait for the data.

301. But this is not how Rich operates. In Rich, vertex data is sent to the PEs by central control unit 38. *See e.g.* Rich, 7:45-55; 8:1-13; 9:1-39; 17:65-18:9. As explained above, these memory transfers occur simultaneously with PE calculations and thus there is no stall or delay waiting for the next item of vertex data. Therefore, there is no performance improvement from storing such vertex data in PE memory 34.

302. Also, each vertex calculation would take thousands of clock cycles as explained above. This provides ample time for transferring the next item of vertex data without any performance impact. In fact, there is no reason or any clear benefit to modifying Rich as suffested by petitioner.

303. Moreover, the proposed modification, as stated, would not provide an operable system. The system in Rich is very rigid and limited in capabilities. Rich explains that his motivation is to "reuse the amount of hardware required to

provide high speed image generation." *Id.* 2:64-65. Each PE has very limited capabilities and has been tailored to only include what is necessary for what Rich has disclosed. If one were to modify Rich to store vertex and primitive data in PE memory 34, one would need to redesign substantial portions of Rich. This would require experimentation and a substantial redesign of Rich's system.

304. For example, Rich only discloses how to keep a database of vertices in a central memory. If one were to move these vertices to PE memories, one would need to design an algorithm to determine which location in memory to use for each item of vertex data. One would need to design a buffering algorithm to allow one portion of the PE memory to be written with new vertex data while calculations are being performed using existing data in the same memory. The vertex calculation algorithms must then be modified to relocate that data or to use it from a different location in different phases of operation. One would need to understand what is stored in PE memory 34 in the current Rich system in detail to determine if those values need be retained. If so, one must find somewhere else to put those values and design a mechanism to manage them. One would need to design a new mechanism to return the transformed vertices to the central database. One would need to rewrite all of the vertex operation instructions to include memory addressing and determine how such addressing could be performed using Rich's pixel-oriented addressing modes.

305. In a tightly coupled system like Rich, changing the way vertex data is stored and attempting to use a PE memory designed for pixel storage for vertex data would require substantial redesign of algorithms, data structures, addressing modes, data transfer hardware, and many other parts of the system. Moreover, there is no clear benefit to storing vertices in the PE memory.

306. Furthermore, Rich's system and the systems of the '871 patent both must solve a fundamental input-routing issue. This issue arises because these systems have a single type of computational resource that performs operations on two different types of inputs—vertex inputs and pixel inputs. *See, e.g.*, '871 patent, FIG. 5 (CPU 96); Rich, FIG. 2 (processing element 32). As such, some structure and/or policy must be present to determine which calculations can access the shared computational resource at any point in time. The unified-shader architecture disclosed in the '871 patent uses an arbiter and a multiplexer to route the appropriate input data to the shared computation resource at the appropriate time and a shared instruction scheduler to schedule interleaved computational threads. This solves numerous problems related to efficiency, most importantly the ability to schedule a vertex thread to run when a pixel thread has stalled due to the need for high-latency texture data. In order to support such a high-level of efficiency, a general-purpose register block that can store vertex or pixel data is required. This provides for the ability to switch rapidly between data types.

307. Rich's system solves the input-routing issue by performing vertex and pixel operations in discrete, sequential phases. Specifically, Rich's processing elements 32 first perform vertex operations and then write the transformed vertex data to an external memory. Rich, 9:18-25, 9:27-29, 9:36-39, 17:60-64. Instead of using hardware structures such as the general-purpose register file and the sequencer claimed in '871, Rich uses a software and architecture-based policy to separate the vertex and pixel calculations into distinct phases with distinct modes of operation. Since Rich does not interleave pixel and vertex operations and does not use threads, there is no need to have a general-purpose register file that can maintain data of two types or to have a sequencer capable of maintaining instructions operative to cause the processor unit to execute vertex calculation and pixel calculation operations on selected data maintained in the general purpose register block.

308. Also, Rich specifically designed his computational phases based on storing vertex data in a list or database in a large external memory. As a result, the results of the vertex processing phase can be treated as a single global data structure and sorted, assigned, and routed to the appropriate processing elements 32 for use during the pixel-processing phase. *See e.g. id.* at 8:32-40, 9:43-46.

309. Thus, by sequentially performing vertex and pixel operations, Rich's processing elements 32 can execute the appropriate instructions on the appropriate data at the appropriate time without a need for the hardware support structures in the '871 and can devote more of his hardware to having more PEs on a single chip. In my opinion, Rich does not disclose a "general purpose register block" and a "sequencer" as in claim 15, because Rich's system has no need for these structures.

310. In my opinion, Rich's system would not work for its intended if it was modified to store vertex data in on-chip memory 34, as LG proposes. During the pixel-processing phase, Rich's processing elements 32 need access to the transformed vertex data. To give the processing elements access to this data, Rich's system stores this data in the external—shared—memory. *Id.* at 9:18-25, 9:27-29, 9:36-39, 17:60-64. If this vertex data was instead maintained or stored in on-chip memory 34, as LG proposes, then it would *not* be easily accessible to the processing elements assigned to operate on that data during the pixel-processing phase. Moreover, Rich does not disclose any algorithms or mechanisms for moving elements of his vertex database onto PE memories and then back into the vertex database. As noted, one would need to invent additional functionality and make additional unspecified modifications to Rich to permit the proposed modification to work.

311. In my opinion, a POSA would have understood that Rich teaches that the external memory is a shared memory for all the processing elements 32. *See id.* at 9:28-36, 16:45-55, 16:58-65, 17:56-6. In contrast, a POSA would have understood that the on-chip memory 34 is a dedicated memory for just one processing element. *See id.* at 7:30-32, 25:18-20, 32:3-14. A POSA would have understood, therefore, that the transformed vertex data would have to be stored in the external, shared memory to be efficiently accessible by Rich's other processing elements 32.

B. Kurihara does not teach or suggest a "processor unit" that "executes vertex calculations while the pixel calculations are still in progress," as in claim 20.

312. Claim 20 depends from independent claim 15 and recites that "the processor unit executes vertex calculations while the pixel calculations are still in progress." In my opinion, claim 20 is not obvious in view of Rich and Kurihara for at least two reasons. First, neither LG nor Dr. Bagherzadeh have explained how a POSA would modify Rich's system based on Kurihara's teachings. Second, even if these references are combined, neither of them teaches or suggests the limitations of claim 20.

313. LG concedes that Rich does not disclose this limitation. *See* Pet. at 58 ("Rich does not explicitly disclose that both vertex and pixel processing occur

simultaneously . . . ."). To fix this deficiency, LG seeks to combine Kurihara's teachings with Rich. *See id.*

314. First, in my opinion, Rich's and Kurihara's teachings are incompatible. Rich discloses a system with a plurality of processing elements that sequentially perform operations on different data types.

> Modeling transformation and the viewing operation (i.e. converting from a 3D model to a two dimensional view of that model) are next sequentially performed, followed by rasterization.

Rich, 2:7-10.

> After geometry processing, the next function carried out by the image generation system is rasterization.

Id. at 9:40-41.

315. Kurihara discloses a SIMD system that can perform multiple identical calculations at the same time using multiple graphics processors. It does not, however, disclose simultaneously operating on two different types of data or interleaving calculations of different types. In fact, Kurihara teaches almost nothing about what its graphics processors can do or how they are structured. Kurihara merely notes that its multiple graphics processors can simultaneously

process multiple sets of coordinate data at one time or multiple sets of attribute data at one time.

316. Kurihara is a patent about deciding how and when to refill FIFO memories. The only aspect of Kurihara that relates to pixel and vertex data is that the refill threshold for the FIFO depends on the data type. In fact, although Kurihara discloses that a FIFO can hold either coordinate data or attribute data, it does not disclose that this FIFO can hold both types of data at the same time. Separate detection mechanisms are provided for each data type, but Kurihara implies that all of the data in the FIFO must be of one type for the corresponding detector to operate.

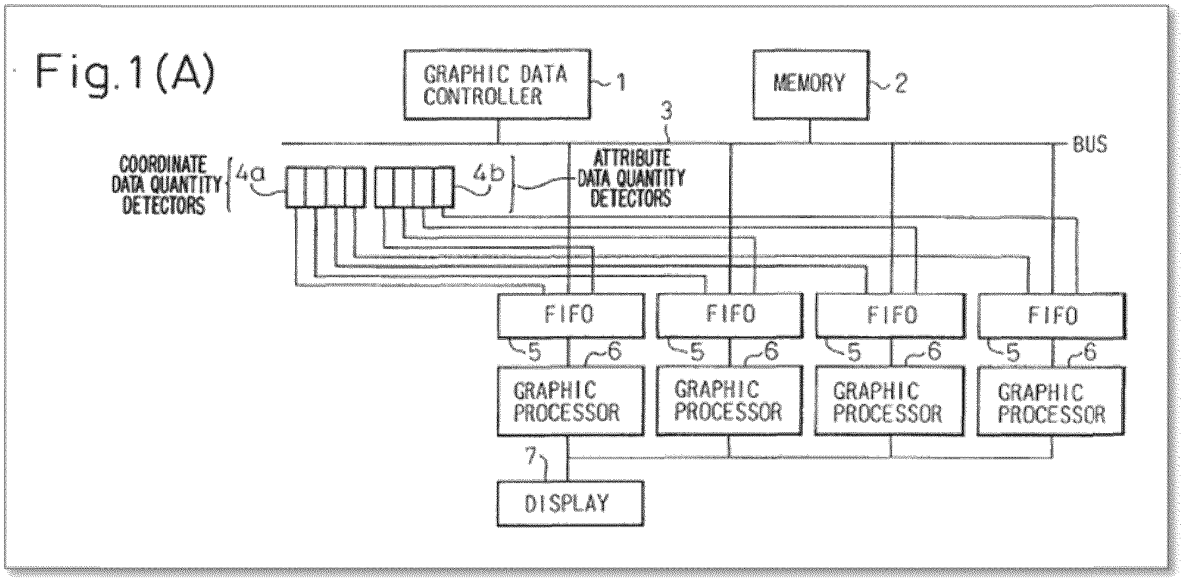
317. The FIFO memories in Kurihara are not general-purpose register files as claimed in '871 at least for the reason that they do not include addressable registers. Neither the graphics processor, nor any other instruction or sequencer in Kurihara, can select which data in the FIFO to process next. Kurihara does not teach or suggest the register file or sequencer of claim 15 nor the simultaneous operation of claim 20.

318. Because Rich's system and Kurihara's system operate in different ways, combining their teachings would be problematic for several reasons. Rich

specifically relies on addressable memories for its operands, and not FIFOs. *See e.g.* Rich, 33:2-34:11.

319. Additionally, in my opinion, Kurihara does not teach or suggest the "processor unit" of claim 20. Kurihara is directed to a graphics-processing system that includes a plurality of graphics processors 6, as shown in Kurihara's Figure 1(A) (reproduced below). Each graphics processor is coupled to a corresponding FIFO 5. The FIFO memories store graphics data to be processed in parallel by the graphics processors 6. Kurihara, 4:38-40. The graphics data in the FIFO memories can be either vertex (coordinate) data or pixel (attribute) data. *Id.* at 4:56-65. These FIFO memories then simultaneously transfer the data to the graphics processors 6. *Id.* at 4:61-62. The graphics processors 6 then process the graphics data in parallel. *Id.* at 5:37. In other words, in Kurihara's system, each graphics processor 6 performs one type of graphics-processing operation on the data type that it receives from the corresponding FIFO memory 5.

Fig.1(A)



320. Kurihara's individual graphics processors function differently than the claimed "processor unit" of claim 20. In claim 20, the single claimed "processor unit" executes a first type of graphics-processing operation (i.e., "vertex calculations") while a second type of graphics-processing operation (i.e., "pixel calculations") is still in progress. In other words, the claimed "processor unit" can stall one type of graphics-processing operation—while that operation is still in progress—in order to perform another type of graphics-processing operation. Kurihara does not disclose threads or any other mechanism for interleaving calculations of different types. Moreover, the FIFO descriptions indicate that only one type of data can be in the FIFO structures at a time.

321. Kurihara does not teach or suggest executing a first type a graphics-processing operation while a second type of graphics-processing operation is still

in progress. Instead, each of Kurihara's graphics processors 6 simply performs a graphics-processing operation on the data it receives from the corresponding FIFO memory. If one of Kurihara's graphics processors receives vertex data, it will operate on that vertex data; if it receives pixel data, it will operate on the pixel data. All of the processors receive the same type of data at the same time. Nowhere does Kurihara disclose that one of these graphics processors can stall a first type of operation (e.g., an attribute-based operation) in order to perform a second type of operation (e.g., a coordinate-based operation). In other words, unlike the claimed "processor unit," none of Kurihara's graphics processors 6 can perform one type of graphics operation while a second type of graphics operation is still in progress.

C. Objective indicia of non-obviousness

322. I have been asked to consider a number of factual questions relating to the inventions claimed in the '871 patent. I understand that these are called objective indicia of non-obviousness. I view them more as windows into the state of the art when ATI's engineers performed the work underlying the '871 patent and released their innovations to the public.

323. Two indicia that I consider are initial skepticism and later industry acceptance. These two indicia straddle the commercial release of ATI's Xenos chip, which appeared in the Microsoft® Xbox 360®.

324. In particular, ATI's Xenos chip became the first commercially available GPU with a unified shader. This is reported by one of my former students, Greg Humphreys, in an article he co-authered for *Computer*. *See* Ex. 2078, p. 4 ("Unified shaders were first realized in the ATI Xenos chip for the Xbox 360 game console . . . ."). *Computer* is an IEEE publication that is peer reviewed and is considered the flagship publication of the IEEE computer society. *See* http://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=2. In my experience, *Computer* is a reliable publication. In addition, a textbook, co-authored by a member of the LG Electronics Institute of Technology, states that "[t]he first unified shader was implemented in Xenos by ATI for X-Box 360." Ex. 2082, p. 114.

325. Based on ATI's design, Microsoft had the Xenos chip fabricated for inclusion in the Xbox 360. This is reported in Dean Takahashi's book regarding the Xbox 360. *See* Ex. 2119, Dean Takahashi, The Xbox 360 Uncloaked 187 (Spider Works LLC 2006) ("Microsoft would check progress on [ATI's] work and then set up the fabrication schedule at its chip contract manufacturer, Taiwan Semiconductor Manufacturing Co."). Dean Takahashi is a respected journalist with expertise in the 3D graphics hardware industry. In my experience, Dean Takahashi's book is a reliable publication. In fact, I know that he interviewed Microsoft staff extensively to develop the facts reported in his book.

326. In 2004, before ATI's commercial release of the Xenos chip, Nvidia's chief architect, David Kirk, questioned whether a unified-shader architecture would even work. He said:

> It's not clear to me that an architecture for a good, efficient, and fast vertex shader is the same as the architecture for a good and fast pixel shader. A pixel shader would need far, far more texture math performance and read bandwidth than an optimized vertex shader. So, if you used that pixel shader to do vertex shading, most of the hardware would be idle, most of the time.

Ex. 2080, Anton Shilov, "ATI and NVIDIA Proclaim Different Graphics Processors Architecture Goals," at p. 1 (Dec. 23, 2004).

327. Mr. Kirk also said that it would be a "challenge" and that it would be "difficult" to design a GPU with a unified shader:

> It's far harder to design a unified processor – it has to do, by design, twice as much. Another word for 'unified' is 'shared,' and another word for 'shared' is 'competing.' It's a challenge to create a chip that does load balancing and performance prediction. It's extremely important, especially in a console architecture, for the performance to be predictable. With all that balancing, it's difficult to make the performance predictable.

Ex. 2081, Anton Shilov, "NVIDIA Says It Would Make a Chip with Unified Pipes 'When it Makes Sense'" (July 11, 2005).

328. Since the Xenos chip was released in the Xbox 360, the graphics-processing industry has moved toward a unified-shader architecture. For example, Microsoft's DirectX (DX10) has adopted the unified shader model. *See* Ex. 2087, p. 14. ("Shader Model 4.0 (SM4.0) is the new instruction set architecture (ISA) for DX10 that looks at the graphics in a unified way.") One advantage of SM4.0 for DirectX is "Flexible Load Balancing." (*Id.*) "The unified shader is made up of shader blocks that can handle all vertex, pixel, and geometry instructions, so the GPU is fully utilized without concern for shader loading imbalances." *Id.* For flexible load balancing, "[t]here is also additional logic to load balance the shader units to keep all functional units fully utilized. If more pixel processing is needed, then more of the unified shader blocks can be allocated to pixel processing to increase throughput." *Id.* Dr. Bagherzadeh also agrees that both DX10 and OpenGL "require a unified shader." Ex. 2074, 103:16-20. Many companies' graphics products use the unified shader architecture, including the S3 Graphics Chrome 400 (*see* Ex. 2087, p. 14), NVIDIA GeForce 8800 GPU and GeForce GTX 200 GPU (*see* Ex. 2090, pp. 9, 21), Intel Processor Graphics (*see* Ex. 2091, p. 12), and Qualcomm Adreno GPUs (*see* Ex. 2092, p. 5).

329. The mobile environment has also been moving towards adapting the unified-shader architecture. “In the mobile environment, a fully programmable 3D graphics pipeline is required. Owing to the need for low power consumption and small area, the conventional architecture with separate vertex shader and pixel shader is hard to implement. Since a unified shader can compute vertex shading and pixel shading in a single hardware, it is a good solution for programmable 3D graphics.” Ex. 2082, p. 114. For example, a “mobile unified shader is designed to perform both programmable vertex operation and programmable pixel operation, which are fully compatible with the mobile 3-D graphics API – OPENGL|ES2.0.” Ex. 2089, p. 2049. Companies implementing the unified shader architecture in their products are thus able to remain competitive.

XIV. CONCLUSION

330. In signing this declaration, I recognize that the declaration will be filed as evidence in a contested case before the Patent Trial and Appeal Board of the United States Patent and Trademark Office. I also recognize that I may be subject to cross-examination in the case and that cross-examination will take place within the United States. If cross-examination is required of me, I will appear for cross-examination within the United States during the time allotted for cross-examination.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true. The statements in this declaration were made with the knowledge that willful false statements and the like are made punishable by fine or imprisonment under Section 1001 of Title 18 of the United States Code and that willful false statements may jeopardize the validity of the '871 patent.

Executed this 14th day of October in Los Gatos, CA

Respectfully submitted,

[illegible]

Andrew Wolfe