UNITED STATES PATENT AND TRADEMARK OFFICE

_____

BEFORE THE PATENT TRIAL AND APPEAL BOARD

_____

LG ELECTRONICS, INC.,
Petitioner

v.

ATI TECHNOLOGIES ULC,
Patent Owner

_____

Case IPR2015-00325
Patent 7,742,053

_____

DECLARATION OF ANDREW WOLFE
REGARDING ACTUAL REDUCTION TO PRACTICE OF
U.S. PATENT NO. 7,742,053

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

Table of Contents

I, Andrew Wolfe, declare as follows:

I.   INTRODUCTION

1.   I have been retained by Advanced Micro Devices ("AMD") as an expert 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, 5, 6, and 7 of the U.S. Patent No. 7,742,053 ("Lefebvre '053 patent").

2.   I have also been retained by AMD to review U.S. Patent Application No. 10/673,761 ("the '761 Application"), filed September 29, 2003, to which the '053 patent claims priority, and to provide my opinion regarding whether claims 1, 2, 5, 6, and 7 are supported by the '761 Application.

3.   And, I have been retained by AMD to review ATI Technologies ULC.'s ("ATI") R400 chip internal documents from August 24, 2001 to April 19, 2002, and to provide my opinion regarding whether the inventors of the '053 patent conceived claims 1, 2, 5, 6, and 7.

II.   BACKGROUND

4.   I have more than 30 years of experience as a computer architect, computer system designer, personal computer graphics designer, educator, and

- 1 -

executive in the electronics industry. A curriculum vitae is attached as Exhibit

2136 to this report and is summarized below.

5.   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.

6.   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.

7.   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.

8.   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.

9.   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.

10.   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.

- 3 -

11.   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.

12.   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.

- 4 -

13.	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.

14.	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.

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

16.	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.

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

18.   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.

19.   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

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

Exhibit Number	Reference
1001	United States Patent No. 7,742,053 to Lefebvre *et al.*
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 0.1)
2072	RTL Code File: sq.v
2073	RTL Code File: sq_thread_buff.v
2074	RTL Code File: sq_thread_arb.v
2075	RTL Code File: sq_ctl_flow_seq.v
2076	RTL Code File: sq_instruction_store.v
2077	RTL Code File: sq_target_instr_fetch.v
2078	RTL Code File: sq_tex_instr_queue.v
2079	RTL Code File: sq_tex_instr_seq.v
2080	RTL Code File: sq_ais_output.v
2081	RTL Code File: sq_alu_instr_queue.v
2082	RTL Code File: sq_alu_instr_seq.v
2083	RTL Code File: sp.v
2084	RTL Code File: vector.v
2085	RTL Code File: macc_gpr.v
2086	RTL Code File: macc.v
2087	RTL Code File: tp.v
2088	Emulator Code File: sq_block_model.cpp
2089	Emulator Code File: user_block_model.h
2090	Emulator Code File: arbiter.cpp
2091	Emulator Code File: arbiter.h

2092	Emulator Code File: sq_alu.cpp
2093	Emulator Code File: sq_alu.h
2094	Emulator Code File: gpr_manager.cpp
2095	Emulator Code File: gpr_manager.h
2096	Emulator Code File: instruction_store.cpp
2097	Emulator Code File: instruction_store.h
2098	Emulator Code File: reg_file.cpp
2099	Emulator Code File: reg_file.h
2100	Emulator Code File: tp.cpp
2101	Emulator Code File: tp.h
2102	Emulator Code File: sq_tp.h
2103	Emulator Code File: tp_block_model.cpp
2104	Emulator Code File: user_block_model.h (tp)
2108	RTL Code File: tp_input.v
2119	United States Patent Application No. 10/673,761 to Lefebvre *et al.*
2136	Curriculum Vitae of Dr. Andrew Wolfe

IV. REDUCTION TO PRACTICE

21. 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.

A. *Actual Reduction to Practice*

- 8 -

22.   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.

23.   Here, I have examined two types of source code: the R400 RTL code

for an early version of the R400 written in Verilog and the corresponding Emulator

Code written in C++. 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. The C++ Emulator code is

generally used in the development of 3D graphics chips to model, validate, and test

the functionality and certain structural features of a hardware design. Below, I will

identify the specific files, objects, input/output interfaces, and functions that

describe each element of claims 1, 2, 5, 6, and 7 of the '053 patent.

B.   Constructive Reduction to Practice

24.   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 September 9, 2003, with the

filing of the '761 Application. I understand that the '053 patent claims priority to

the '761 Application, because, U.S. Patent Application No. 11/764,453 from which

the '053 patent issued, is a continuation of the '761 Application. Below, I include a

claim chart where I identify support for each element of claims 1, 2, 5, 6, and 7 of

the '053 Patent in the '761 Application.

V.   U.S. PATENT NO. 7,742,053

25.   The '053 patent is directed to a graphics-processing system having a

unified shader. The unified shader can perform both pixel and vertex calculations.

To do this, the '053 patent includes at least one memory device designed to store a

plurality of pixel command threads and a plurality of vertex command threads.

('053 patent, Abstract.)

26.   The first reservation station 302 and the second reservation station

304 of the '053 patent represent the "at least one memory device" of independent

claims 1 and 5. ('053 patent, 3:63-64.) The first reservation station 302 is a pixel

reservation station and stores pixel command threads (including 312, 314, and

316), while the second reservation station is a vertex reservation station and stores

vertex command threads (including 318, 320, and 322). (*Id.*, 3:66-4:4.)

27.   The pixel command threads 312, 314, and 316 and the vertex

command threads 318, 320, and 322, exemplify the command threads of the

claimed inventions.

28.   The claims of the '053 patent also include an arbiter. The arbiter in a

preferred embodiment is operable to select a command thread from the vertex and

- 10 -

pixel reservation stations by picking the first command thread ready to execute.

(*Id.*, 3:49-51.) The arbiter's selection is based on a priority scheme, which may

depend on which commands have already been or are to be executed within a

command thread and/or the age of the command thread in the reservation station.

(*Id.*, 3:31-36.)

29.   The arbiter provides the selected command thread to a command

processing engine. (*Id.*, 3:8-11.)

30.   The '053 patent specification recites two types of exemplary

command processing engines: the ALU processing engine referred to as ALU 308

and a texture processing engine, such as a graphics-processing engine 310. (*Id.*,

4:30-33.)

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

31.   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.

32. 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."

33.   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.

34.   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 '053 patent. Only

after the components claimed in the '053 patent (along with other chip

components) worked for their intended purpose, would ATI generate the tooling

and send it for fabrication. Because the '053 patent pertains to the chip-circuit

design, the actual reduction to practice of the claims of the '053 patent would have

occurred when the RTL code or the Emulator Code performed all limitations of the

claims.

VII.   THE CODE FOR ATI'S R400 CHIP

35.   I have been asked to review the source code for ATI's R400 chip. I

will cite to the 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.

36.   There are two corresponding design databases that comprise the

source code: R400 RTL code and Emulator Code. The R400 RTL code is

implemented in a hardware-description language (HDL), called Verilog. Verilog is

used to design and verify digital circuits at register-transfer level of abstraction

which can include both structure and function. For example, in the R400 program,

Verilog was used to validate the integrated-circuit version of the graphics-

processing system recited in claims 1, 2, 5, 6, and 7.

37.   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 recited in

claims 1, 2, 5, 6, and 7 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.

38.   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

- 14 -

C++ are saved. Next the RTL 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.

39.   I have compared each element of claims 1, 2, 5, 6, and 7 to the R400 R400 RTL code and the Emulator Code. Below, I will discuss each element of claims 1, 2, 5, 6, and 7, and the corresponding files, functions, and interfaces along with the pages and line numbers in the RTL and/or Emulator Code that disclose the same element. In my opinion, both the R400 RTL code and the R400 Emulator Code each disclose all elements of claims 1, 2, 5, 6, and 7.

40.   At least one version of the R400 RTL code which discloses all elements of claims 1, 2, 5, 6, and 7 includes the files generated before or on August 5, 2002, and are attached as Exhibits 2072-2087.

41.   At least one version of the R400 Emulator Code which discloses all elements of claims 1, 2, 5, 6, and 7 includes the files generated before or on August 5, 2002, and are attached as Exhibits 2088-2104.

A.   *The R400 RTL code corresponding to claims 1, 2, 5, 6, and 7*

42.   As I mentioned above, the R400 RTL Code is written in Verilog language. Verilog is a hardware-description language used to design and specify

- 15 -

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.

43.   The R400 RTL code includes the sq.v, sp.v, tp.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, 5,

- 16 -

6, and 7. In particular, the sq.v file specifies and generates a sequencer which includes an arbiter and the at least one memory recited in the claims. The sp.v and tp.v files each specify and generate a command processing engine – the ALU processing engine (sp.v) and a texture processing engine (tp.v). I will discuss each of these components below.

1. Claim 1

a. The Preamble

44. The preamble of claim 1 recites "*A graphics processing system.*" The R400 RTL code included in the files attached as Exhibits 2072-2087 generates components of the graphics-processing system of claim 1. The file, sq.v, defines the hardware blocks of the graphics-processing system component called a sequencer. In particular, sq.v instantiates a texture thread arbiter *u_sq_tex_thread_arb* (sq.v, 43:3-44-21), an ALU thread arbiter *u_sq_alu_thread_arb* (sq.v, 47:6-48-24), a memory buffer that stores pixel command threads *u_sq_pix_thread_buff* (sq.v, 38:27-42:29), and a memory buffer that stores vertex command threads *u_sq_vtx_thread_buff* (sq.v, 34:22-38:24). The memory buffers are what the '053 patent refers to as the pixel reservation station and the vertex reservation station.

- 17 -

45.   I have generated a visual representation of these components, as I

understand them, based on the R400 RTL code, in a figure below. The figure

includes the names of the components as they are instantiated in sq.v.
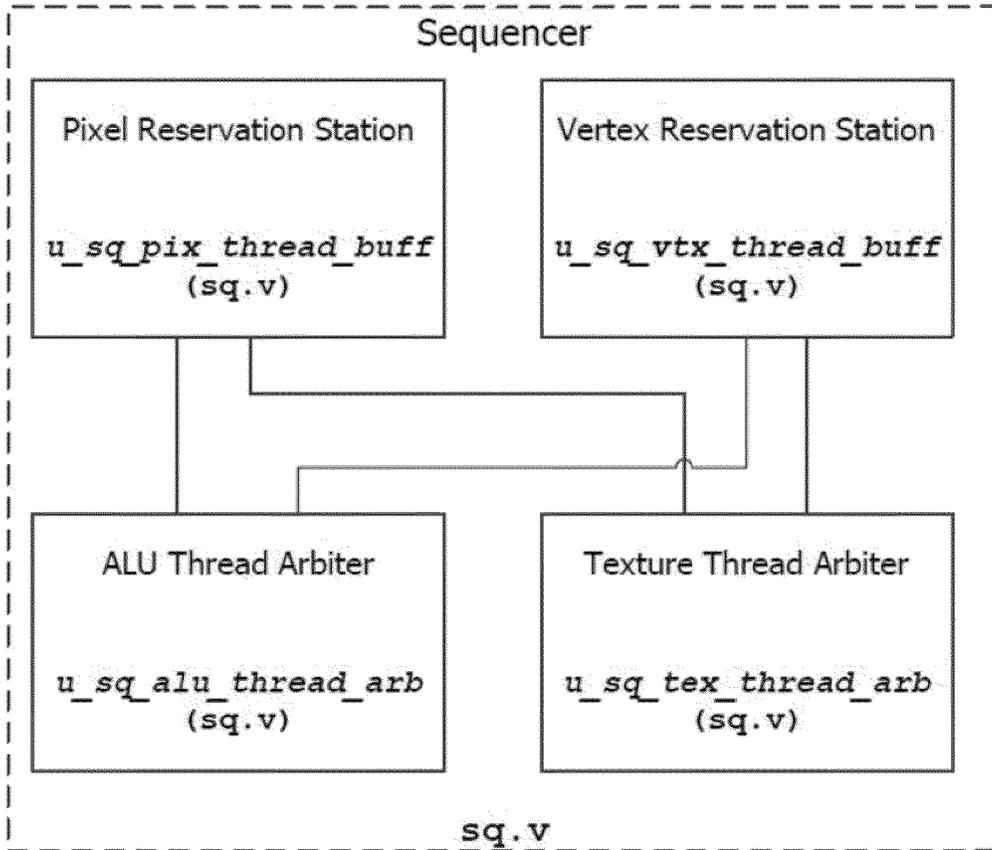
46.   The texture thread arbiter (*u_sq_tex_thread_arb*) and the ALU thread

arbiter (*u_sq_alu_thread_arb*) are arbiter of the '053 patent that I described above,

and that is recited in claim 1.

47.   Also, the *u_sq_pix_thread_buff* is a memory buffer for a pixel

reservation station, while the *u_sq_vtx_thread_buff* is a memory buffer for a vertex

reservation station, which I also described above. The *u_sq_pix_thread_buff* and

the *u_sq_vtx_thread_buff* structures are components of the at least one memory

device recited in claim 1.

b.   The at Least One Memory Device

48.   The first element of claim 1 recites *"at least one memory device*

comprising a first portion operative to store a plurality of pixel command threads

and a second portion operative to store a plurality of vertex command threads."

49.   As I discussed above, the sq.v file instantiates a memory buffer for

pixel command threads called *u_sq_pix_thread_buff* module (sq.v, 38:27- 42-29)

and a memory buffer for vertex command threads called a *u_sq_vtx_thread_buff*

module (sq.v, 34:22-38-24). The *sq_thread_buff* module defined in

sq_thread_buff.v generates *u_sq_pix_thread_buff* and *u_sq_vtx_thread_buff*.

Module *u_sq_pix_thread_buff* is the pixel reservation station and

u_sq_vtx_thread_buff is the vertex reservation station of the '053 patent. I have

generated a visual representation of the pixel reservation station and the vertex

reservation station, as I understand it, based on the R400 RTL code, in a figure

- 19 -

below. The figure includes the names of the components as they are instantiated in

files that describe the structure and behavior of the components.
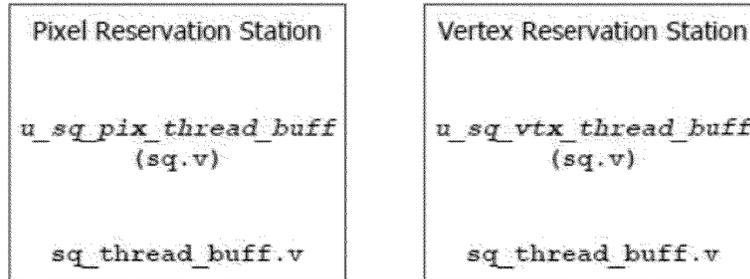
50.   With respect to the pixel command threads and vertex command

threads, each of *sq_pix_thread_buff* and *sq_vtx_thread_buff* includes 16 registers,

referred to as *u0_sq_status_reg* to *u15_sq_status_reg*. (sq_thread_buff.v, 37:16-

54:16.) Each register stores a command thread, including the command thread's

state and status information. The hardware code that generates a second register

that stores a command thread is replicated below:

```
sq_status_reg #( TID_WIDTH, STATUS_WIDTH )
   u1_sq_status_reg (
      .thread_type_strap(thread_type_strap),
      .ism_load(ism_status_sel[1]), .ism_thread_id(state_tail_ptr_q),
      .ism_resource(ism_resource),
.ism_first_thread(ism_first_thread),
      .cfs_update(cfs_update),
      .cfs_thread_id(cfs_thread_id),
      .cfs_alu_instr_pending(cfs_alu_instr_pending),
      .cfs_pulse_sx(cfs_pulse_sx),
      .cfs_last_instr(cfs_last_instr),
      .cfs_pos_allocated(cfs_pos_allocated),
      .cfs_alloc_type(cfs_alloc_type),
      .cfs_alloc_size(cfs_alloc_size),
      .cfs_tex_read_pending(cfs_tex_read_pending),
      .cfs_serial(cfs_serial),
      .cfs_resource(cfs_resource),
```

- 20 -

```
    .cfs_thread_valid(cfs_thread_valid),
    .sx_pos_avail(pos_avail_q), .sx_buf_avail(buf_avail_q),
.param_cache_wptr_q(param_cache_wptr_q),
    .winner_sel      (winner_status_sel[1]),
    .tp_done(qual_tp_done), .tp_thread_id(tp_thread_id_q),
    .ais_done(qual_ais_done), .ais_thread_id(ais_thread_id),
.pop_thread(pop_thread),
    .tex_req_q(tex_req_q[1]), .alu_req_q(alu_req_q[1]),
    .status_in_q(status_data_2), .status_out_q(status_data_1),
    .clk(clk), .reset(reset)
);
```

(sq_thread_buff.v, 40:5-41:7)

51.   There are 16 command thread registers in *sq_pix_thread_buff* and

sixteen command thread registers in *sq_vtx_thread_buff*.

52.   Additionally, each of the vertex command threads and the pixel

command threads also stores its constituent instructions in an instruction store.

These instructions are accessed using the command thread's state and status

information once an arbiter selects the command thread for processing, as will be

described below. The instruction store is instantiated as *sq_instruction_store* using

the *sq_instruction_store* module in sq.v at 87:21-88:25. The instruction store

module is defined in the sq_instruction_store.v file. It consists of 4096 instruction

words which are each 96-bits wide. I have generated a visual representation of a

reservation station (which can be either vertex or pixel reservation station)

operable to store command threads and of the instruction store operable to store

instruction(s) of the command thread. The visual representation of the figure below

is based on my understanding of the R400 RTL code.

- 21 -

Reservation Station

u0_sq_status_reg

u15_sq_status_reg

sq_thread_buff.v

Instruction Store

sq_instruction_store.v
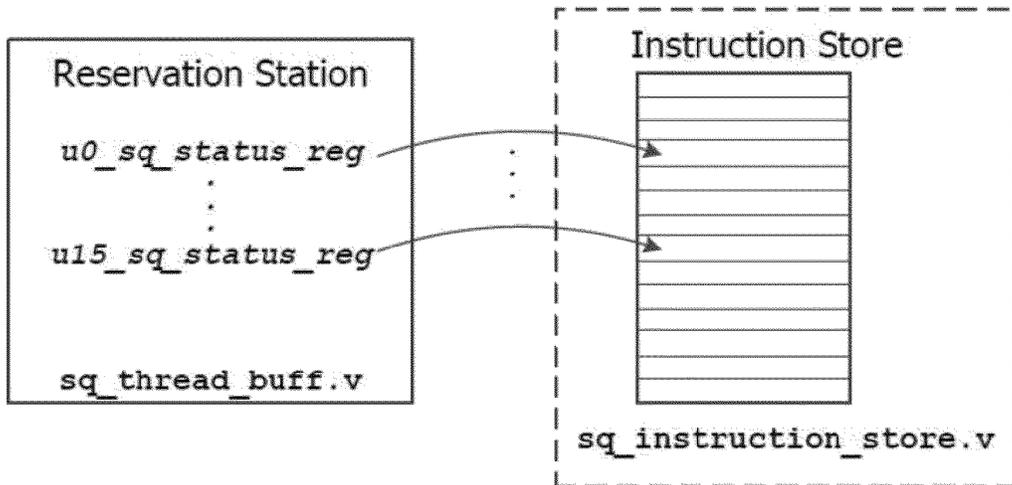
53.   The *sq_pix_thread_buff* memory buffer (the pixel reservation station)
and the *sq_vtx_thread_buff* memory buffer (the vertex reservation station)
generated using *sq_thread_buff* module along with the instruction store form the at
least one memory device recited in claim 1. The *sq_pix_thread_buff* buffer and the
instruction store form a first portion of the at least one memory operative to store a
plurality of pixel command threads. The *sq_vtx_thread_buff* and the instruction
store form the second portion of the at least one memory operative to store a
plurality of vertex command threads.

54.   In this way, the sq.v, the sq_thread_buff.v and the
sq_instruction_store.v files include the R400 RTL code that defines the at least one
memory device comprising a first portion operative to store a plurality of pixel

command threads and a second portion operative to store a plurality of vertex

command threads.

c. The Arbiter

55. The second element of claim 1 recites *"an arbiter, coupled to the at*

least one memory device." In the sq.v, the sequencer instantiates two instances of

the arbiter. The *u_sq_alu_thread_arb* arbiter and the *u_sq_tex_thread_arb* arbiter

are two instances of an arbiter that collectively correspond to the arbiter described

in the '053 patent, and that is recited in claims 1 and 5.

56. The *u_sq_alu_thread_arb* arbiter performs vertex and pixel command

thread arbitration for an ALU processing engine (sq.v, 47:6-48:24), and the

u_sq_tex_thread_arb arbiter performs vertex and pixel command thread arbitration

for a texture processing engine (sq.v, 43:3-44:21). I have generated a block

diagram representation of the two arbiters, based on my understanding of the R400

RTL code, the *u_sq_alu_thread_arb* and *u_sq_tex_thread_arb*, below. The figure

includes the names of the components as they are instantiated and files that

describe the behavior of the components.

- 23 -

```
                          Sequencer

   ┌──────────────────────────┐   ┌──────────────────────────┐
   │ Pixel Reservation Station│   │Vertex Reservation Station│
   │                          │   │                          │
   │ u_sq_pix_thread_buff     │   │ u_sq_vtx_thread_buff     │
   │        (sq.v)            │   │        (sq.v)            │
   └──────────────────────────┘   └──────────────────────────┘

   ┌──────────────────────────┐   ┌──────────────────────────┐
   │    ALU Thread Arbiter    │   │   Texture Thread Arbiter │
   │                          │   │                          │
   │ u_sq_alu_thread_arb      │   │ u_sq_tex_thread_arb      │
   │        (sq.v)            │   │        (sq.v)            │
   └──────────────────────────┘   └──────────────────────────┘

                            sq.v
```
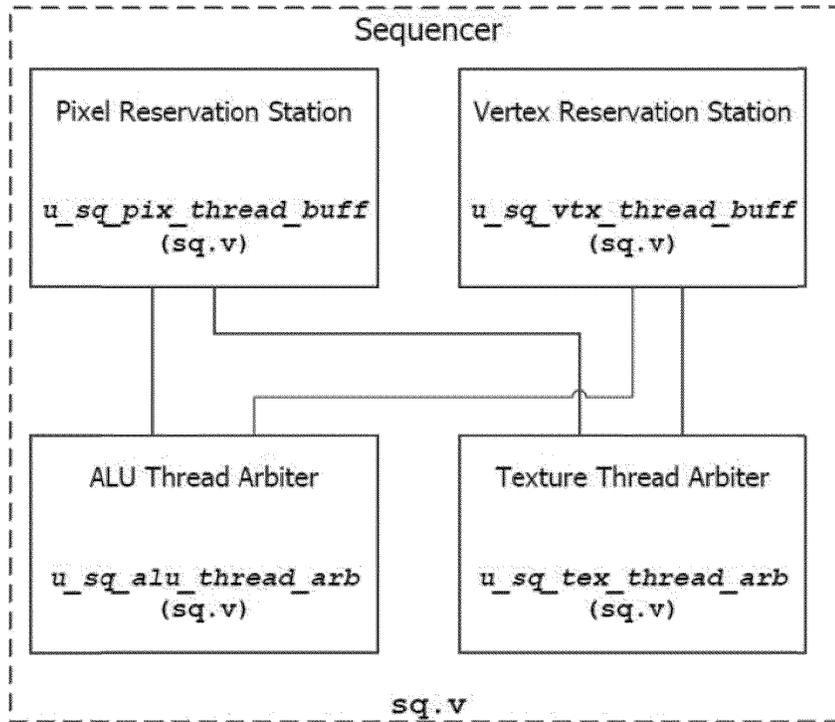
57. The R400 RTL code defining each instance of the arbiter is included in the sq_thread_arb.v file. The definition of the arbiter and the inputs and outputs associated with the arbiter are replicated below:

```
module sq_thread_arb
(
    arb_type_strap,     // tex = 1, alu = 0
    state_read_phase,   // share read access between tex and alu arbs

    // vertex and pixel thread buffer interface

    vtx_req_q,          // 16 vtx_thread_buff requests

    vtx_winner_q, // winning vertex thread_id sent back to Vertex Thread
Buffer
    vtx_winner_ack,     // request acknowledge - indicates to TB that the
winner is valid

    vtx_state,          //
    vtx_status,         //
```

- 24 -

```
    pix_req_q,           // 16 pix_thread_buff requests

    pix_winner_q, // winning pixel thread_id sent back to Pixel Thread
Buffer
    pix_winner_ack,      //

    pix_state,           //
    pix_status,          //

    // control flow sequencer interface

    arb_rts0,      // ready to send the winner to CFS0
    arb_rts1,      // ready to send the winner to CFS1
    arb_state,     // the state sent to the CFS
    arb_status,    // the status sent to the CFS
    arb_thread_type,   // vtx or pix

    cfs_rtr0,      // CFS0 can accept a thread
    cfs_rtr1,      // CFS1 can accept a thread (for alu cfs's)

    cfs1_enable,         // enable sending packets to CFS1 (this a local
register setting: SQ_FLOW_CTL.ONE_ALU)

    clk,
    reset
);
```

<p align="right">(sq_thread_arb.v, 2:8-4:1.)</p>

58.   As I show in the block diagram above, the *u_sq_alu_thread_arb* and

u_sq_tex_thread_arb arbiters are each coupled to the at least one memory which is

operable to store the plurality of the pixel command threads and the plurality of the

vertex command threads. For example, each arbiter receives the 16 pixel thread

requests and 16 vertex command thread requests including the command threads'

state and status information from the *sq_pix_thread_buff* (the pixel memory buffer)

and the *sq_vtx_thread_buff* (the vertex memory buffer), using the inputs below:

```
    input  [15:0]               pix_req_q;
    input  [`SQ_PIX_STATE_WIDTH-1:0]  pix_state;
    input  [`SQ_PIX_STATUS_WIDTH-1:0]  pix_status;
```

<p align="center">- 25 -</p>

(sq_thread_arb.v, 5:4-6; *see also* sq.v, 43:25-44:5 and 48:2-7.)

```
input   [15:0]                    vtx_req_q;
input   [`SQ_VTX_STATE_WIDTH-1:0]   vtx_state;
input   [`SQ_VTX_STATUS_WIDTH-1:0]  vtx_status;
```

(sq_thread_arb.v, 4:20-22; *see also* sq.v, 43:18-23 and 47:21-26.)

59.   Each *pix_req_q* input receives 16 pixel command thread requests, and each *pix_state* and *pix_status* input receives the state and status information for each of the 16 pixel command thread requests. Similarly, each *vtx_req_q* signal receives 16 vertex command thread requests, and each *vtx_state* and *vtx_status* input receives the state and status information for each of the 16 vertex command threads.

60.   The corresponding output from each of the pixel and vertex memory buffers that is connected to these inputs is described in sq_thread_buff.v and is replicated below:

```
output [TB_DEPTH-1:0]        tex_req_q;
output [STATE_WIDTH-1:0]     tex_state_q;
output [STATUS_WIDTH-1:0]    tex_status_q;
```

(sq_thread_buff.v, 9:12-14; *see also* sq.v, 36:16-22.)

```
output [TB_DEPTH-1:0]        alu_req_q;
output [STATE_WIDTH-1:0]     alu_state_q;
output [STATUS_WIDTH-1:0]    alu_status_q;
```

(sq_thread_buff.v, 9:23-10:2; *see also* sq.v, 37:4-9.)

- 26 -

61.   When the *u_sq_alu_thread_arb* or *u_sq_tex_thread_arb* arbiter

selects a pixel command thread or a vertex command thread, the arbiter

communicates the *thread_id* of the selected pixel command thread and the vertex

command thread to the pixel memory buffer and the vertex memory buffer, using

the interface below:

```
output [3:0]              vtx_winner_q;
output [0:0]              vtx_winner_ack;
```

(sq_thread_arb.v, 5:1-2; *see also* sq.v, 43:19-21 and 47:21-24.)

```
output [3:0]              pix_winner_q;
output [0:0]              pix_winner_ack;
```

(sq_thread_ arb.v, 5:8-9; *see also* sq.v, 44:1-3 and 48:3-5.)

and

```
input  [TB_ADDR_WIDTH-1:0]    tex_winner_q;
input  [0:0]                  tex_winner_ack;
```

(sq_thread_buff.v, 9:16-17; *see also* sq.v, 36:18-19.)

```
input  [TB_ADDR_WIDTH-1:0]    alu_winner_q;
input  [0:0]        alu_winner_ack;
```

(sq_thread_buff.v, 10:4-5; *see also* sq.v, 37:6-7.)

62.   The interfaces described above couple each of the

u_sq_alu_thread_arb arbiter and the *u_sq_tex_thread_arb* arbiter to the pixel

thread memory buffer and the vertex thread memory buffer which are the at least

one memory device.

- 27 -

d. The Arbiter is Operable to Select a Command Thread

63. The arbiter of claim 1 is "*operable to select a command thread from either of the plurality of pixel command threads and the plurality of vertex command threads based on relative priorities of the plurality of pixel command threads and the plurality of vertex command threads.*"

64. As already discussed above, the *u_sq_alu_thread_arb* arbiter and *u_sq_tex_thread_arb* arbiter retrieve the pixel thread requests (called *pix_req_q*) as inputs from the pixel thread memory buffer (*sq_pix_thread_buff*) and the vertex thread requests (called *vtx_req_q*) as inputs from the vertex thread memory buffer (*sq_vtx_thread_buff*).

65. Each arbiter then selects a winning pixel command thread from the pixel thread requests and a winning vertex command thread from the vertex thread requests. The structure and functionality that selects the winning vertex command thread and the winning pixel command thread is specified in sq_thread_arb.v. For example, these arbiters select the winning pixel command thread using a priority encoder, which prioritizes the pixel command threads as replicated below:

```
// - pixel request priority encoder

  reg            pix_winner_vld;
  reg [3:0]   pix_winner;

  always @(pix_req_q)
    begin
```

- 28 -

```
    casez (pix_req_q)
       //16'b0000_0000_0000_0000: begin pix_winner_vld = LO;
pix_winner = 4'hf; end
       16'b1000_0000_0000_0000: begin pix_winner_vld = HI;
pix_winner = 4'hf; end
       16'b?100_0000_0000_0000: begin pix_winner_vld = HI;
pix_winner = 4'he; end
       16'b??10_0000_0000_0000: begin pix_winner_vld = HI;
pix_winner = 4'hd; end
       16'b???1_0000_0000_0000: begin pix_winner_vld = HI;
pix_winner = 4'hc; end
       16'b????_1000_0000_0000: begin pix_winner_vld = HI;
pix_winner = 4'hb; end
       16'b????_?100_0000_0000: begin pix_winner_vld = HI;
pix_winner = 4'ha; end
       16'b????_??10_0000_0000: begin pix_winner_vld = HI;
pix_winner = 4'h9; end
       16'b????_???1_0000_0000: begin pix_winner_vld = HI;
pix_winner = 4'h8; end
       16'b????_????_1000_0000: begin pix_winner_vld = HI;
pix_winner = 4'h7; end
       16'b????_????_?100_0000: begin pix_winner_vld = HI;
pix_winner = 4'h6; end
       16'b????_????_??10_0000: begin pix_winner_vld = HI;
pix_winner = 4'h5; end
       16'b????_????_???1_0000: begin pix_winner_vld = HI;
pix_winner = 4'h4; end
       16'b????_????_????_1000: begin pix_winner_vld = HI;
pix_winner = 4'h3; end
       16'b????_????_????_?100: begin pix_winner_vld = HI;
pix_winner = 4'h2; end
       16'b????_????_????_??10: begin pix_winner_vld = HI;
pix_winner = 4'h1; end
       16'b????_????_????_???1: begin pix_winner_vld = HI;
pix_winner = 4'h0; end
       //default:                    begin pix_winner_vld = X;
pix_winner = 4'bxxxx; end
       default:                    begin pix_winner_vld = LO;
pix_winner = 4'bxxxx; end
    endcase
    end
```

(sq_thread_arb.v, 12:15-13:20.)

66.   Each arbiter also selects the winning vertex command thread using a

priority encoder, which prioritizes the vertex command threads as replicated

below:

- 29 -

```
    // - vertex request priority encoder

    reg              vtx_winner_vld;
    reg [3:0]   vtx_winner;

    always @(vtx_req_q)
      begin
        casez (vtx_req_q)
          16'b0000_0000_0000_0000: begin vtx_winner_vld = LO;
vtx_winner = 4'hf; end
          16'b1000_0000_0000_0000: begin vtx_winner_vld = HI;
vtx_winner = 4'hf; end
          16'b?100_0000_0000_0000: begin vtx_winner_vld = HI;
vtx_winner = 4'he; end
          16'b??10_0000_0000_0000: begin vtx_winner_vld = HI;
vtx_winner = 4'hd; end
          16'b???1_0000_0000_0000: begin vtx_winner_vld = HI;
vtx_winner = 4'hc; end
          16'b????_1000_0000_0000: begin vtx_winner_vld = HI;
vtx_winner = 4'hb; end
          16'b????_?100_0000_0000: begin vtx_winner_vld = HI;
vtx_winner = 4'ha; end
          16'b????_??10_0000_0000: begin vtx_winner_vld = HI;
vtx_winner = 4'h9; end
          16'b????_???1_0000_0000: begin vtx_winner_vld = HI;
vtx_winner = 4'h8; end
          16'b????_????_1000_0000: begin vtx_winner_vld = HI;
vtx_winner = 4'h7; end
          16'b????_????_?100_0000: begin vtx_winner_vld = HI;
vtx_winner = 4'h6; end
          16'b????_????_??10_0000: begin vtx_winner_vld = HI;
vtx_winner = 4'h5; end
          16'b????_????_???1_0000: begin vtx_winner_vld = HI;
vtx_winner = 4'h4; end
          16'b????_????_????_1000: begin vtx_winner_vld = HI;
vtx_winner = 4'h3; end
          16'b????_????_????_?100: begin vtx_winner_vld = HI;
vtx_winner = 4'h2; end
          16'b????_????_????_??10: begin vtx_winner_vld = HI;
vtx_winner = 4'h1; end
          16'b????_????_????_???1: begin vtx_winner_vld = HI;
vtx_winner = 4'h0; end
          default:                 begin vtx_winner_vld = X;
vtx_winner = 4'bxxxx; end
        endcase
      end
```

(sq_thread_arb.v, 11:8-12:12.)

- 30 -

67.   Once each arbiter selects a winning pixel command thread and a

winning vertex command thread, the arbiter chooses the command thread from the

winning pixel command thread or the winning vertex command thread. This

corresponds to the selected command thread recited in claim 1. To select the

command thread, each arbiter uses the R400 RTL code below, such that the

winning vertex command thread, if any, has priority over the winning pixel

command thread:

```
else if (ld_winner)
    begin
        vtx_winner_q <= vtx_winner;
        vtx_winner_vld_q <= vtx_winner_vld;
        pix_winner_q <= pix_winner;
        pix_winner_vld_q <= pix_winner_vld;
    end

...
...
...


case (tta_current_state)
    TTA0:
        begin
        // - ack is connected to TB State Mem read enable, so
have to
        // wait until the correct phase to ack

        if ( state_read_phase == arb_type_strap )
            if ( vtx_winner_vld_q
)                    // simply give verts the priority
            begin
                vtx_winner_ack = HI;
                tta_next_state = TTA1;
            end
            else if ( pix_winner_vld_q )
            begin
                pix_winner_ack = HI;
                tta_next_state = TTA2;
            end
        end
```

- 31 -

(sq_thread_arb.v, 16:23-17:6, 20:19-21:16.)

68.   When each arbiter selects the winning command thread, the arbiter
also outputs the attributes of the command thread using the *arb_state*, *arb_status*,
and *arb_thread_type* signals. (sq_thread_arb.v, 5:14-16 .)

69.   As such, each arbiter is operable to select the command thread, as
recited in claim 1.

2.   Claim 2

a.   The Preamble

70.   Claim 2 recites the graphics-processing system of claim 1, further
comprising "*a command processing engine, coupled to the arbiter.*" The R400
RTL code specifies two command processing engines: the ALU processing engine
and the texture processing engine. The R400 RTL code for the ALU processing
engine is included in sp.v and the corresponding sub-files and modules. The R400
RTL code for the texture processing engine is included in tp.v and the
corresponding sub-files and modules. Either the ALU processing engine or the
texture processing engine corresponds to the command processing engine recited
in claim 2.

71.   Below I generated a block diagram based on my understanding of the

R400 RTL code, describing how the ALU processing engine and a texture

processing engine are coupled to the arbiter.
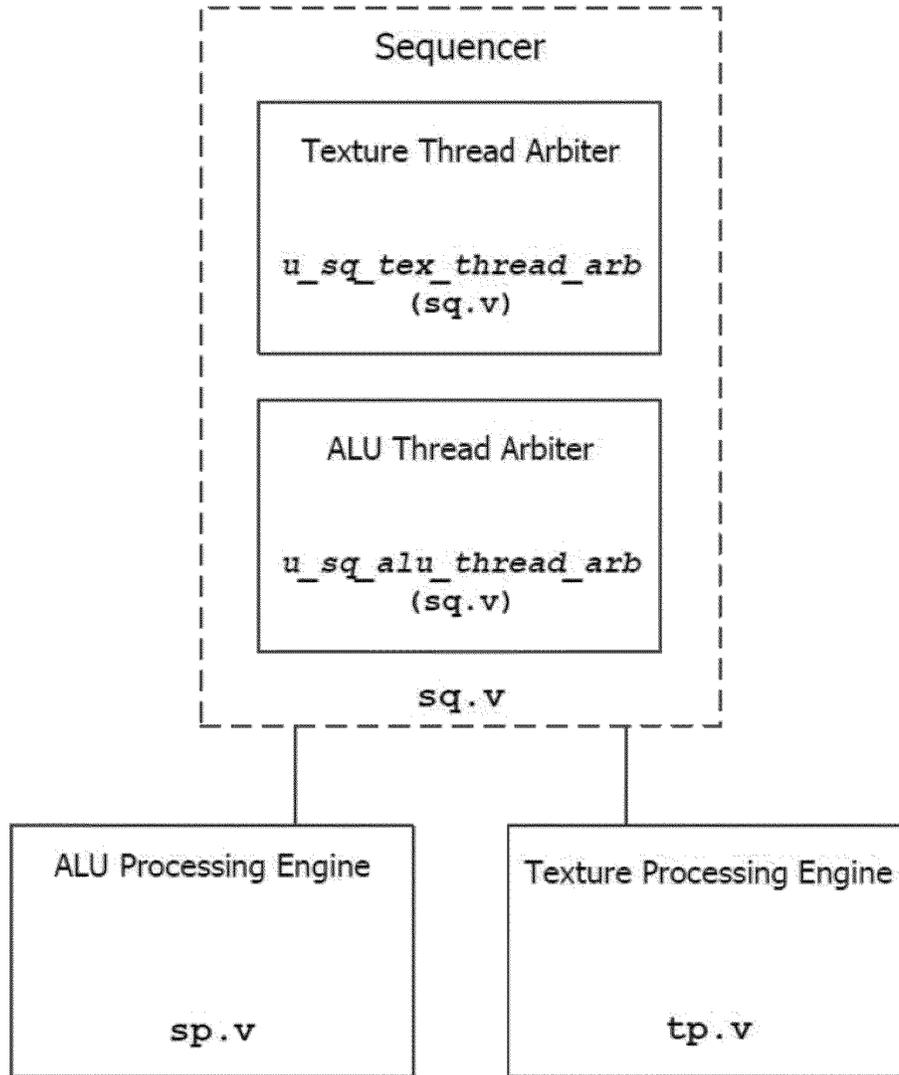
72.   The command processing engine is coupled to the arbiter through the

hardware circuitry, including an *sq_ctl_flow_seq* module (also referred to as a CFS

module and defined in sq_ctl_flow_seq.v) and an *sq_target_instr_f*etch module

(also referred to as a TIF module and defined in sq_target_instr_fetch.v).

73.   In the case of the first command processing engine, the ALU

processing engine, the TIF module is coupled to the *sq_ais_queue* module (also

referred to as an AIQ module) which is specified in sq_alu_inst_queue.v. The AIQ

module provides the command thread's instructions to the *sq_ais_output* module

(also referred to as an AIO module) which is specified in sq_ais_output.v. The AIO

module provides the command thread's instruction to the ALU command

processor. Below I also generated a detailed diagram, based on my understanding

of the R400 RTL code, describing how the ALU processing engine is coupled to

the arbiter.

Figure showing Sequencer (sq.v) containing:
- Arbiter (for ALU Processing Engine): u0_sq_alu_thread_arb, sq_thread_arb.v
- CFS Module 0: u0_sq_alu_ctl_flow_seq, sq_ctl_flow_seq.v
- CFS Module 1: u1_sq_alu_ctl_flow_seq, sq_ctl_flow_seq.v
- TIF Module 0: u0_sq_alu_instr_fetch, sq_target_instr_fetch.v
- TIF Module 1: u1_sq_alu_instr_fetch, sq_target_instr_fetch.v
- Instruction Store: sq_instruction_store.v
- AIQ Module 0: u0_sq_alu_instr_queue, sq_alu_inst_queue.v
- AIQ Module 1: u1_sq_alu_instr_queue, sq_alu_inst_queue.v
- AIO Module: u_sq_ais_output, sq_thread_arb.v
- ALU Processing Engine: sp.v

74. The details of how a command thread's instruction(s) are provided to the ALU processing engine are described below with the reference to the figure above.

75. In the case of the second command processing engine, the texture processing engine, the TIF module is coupled to the *sq_tis_queue* module (also referred to as a TIQ module) which is specified in sq_tex_inst_queue.v . The TIQ module connects to the *sq_tex_instr_seq* module (also referred to as a TIS module), which is specified in sq_tex_inst_seq.v. The TIS module provides the command thread's instructions to the texture processing engine. I generated a

- 35 -

detailed diagram below, based on my understanding of R400 RTL code,

illustrating how the texture processing engine is coupled to the arbiter.
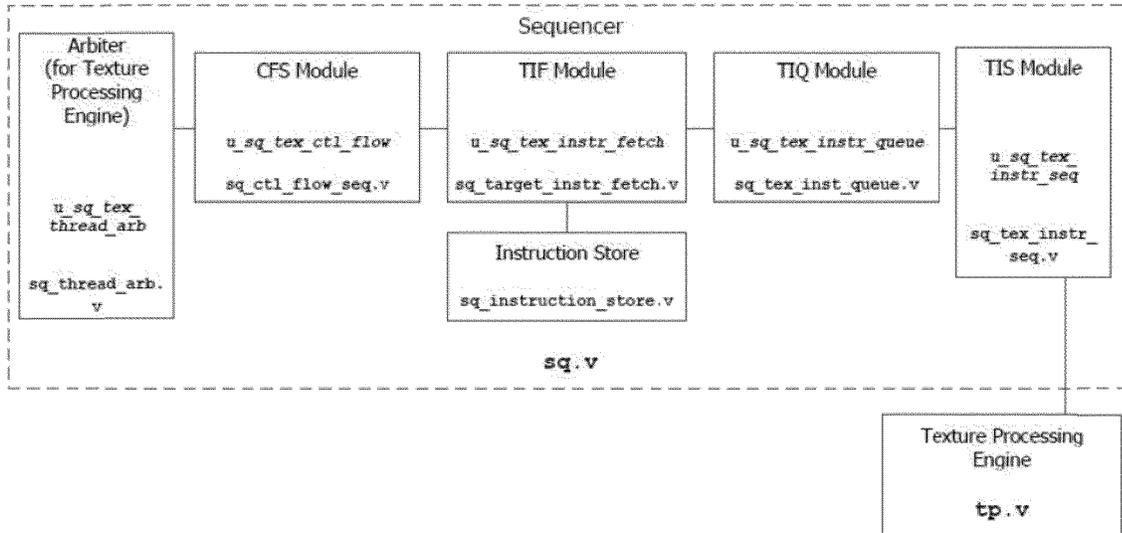
76.   Details describing how a command thread's instructions are provided

to the texture processing engine are described below with respect to this figure

above.

77.   As such, the command processing engine, whether the ALU command

processing engine or the texture processing engine, is coupled to the arbiter.

b.   The Arbiter is Operable to Provide a Command Thread to the Command Processing Engine

78.   Claim 2 also recites *"wherein the arbiter is further operable to*

provide the command thread to the command processing engine."

- 36 -

79.   The arbiter in the R400 RTL code provides the selected command

thread to the ALU processing engine and the texture processing engine. I have

included detailed block diagrams of the components that pass the selected

command thread to the ALU processing engine and the texture processing engine

described in detail below.

80.   Each of the two arbiters selects a command thread. The arbiter outputs

the selected command thread using the *arb_state*, *arb_status*, and *arb_thread_type*

signals. (sq_thread_arb.v, 5:14-16.) The arbiter then passes the selected command

thread by way of these signals to the CFS module called *sq_ctl_flow_seq*.

81.   The CFS module receives the command thread by way of the

arb_state, *arb_status*, and *arb_thread_type* signals and uses these values to

calculate a pointer to the command thread's first instruction in the instruction store

and the number of command thread instructions that require processing.

82.   The CFS module is specified in the sq_ctrl_flow_seq.v file. There are

three instances of the CFS module instantiated in sq.v. The

u0_sq_alu_ctl_flow_seq module (sq.v, 51:9-53:21) is instantiated for the first ALU

command processing engine, and the *u1_sq_alu_ctl_flow_seq* module for the

second ALU command processing engine (sq.v, 53:24-56:8). Also, a third CFS

- 37 -

module is instantiated as the *u_sq_tex_ctl_flow_seq* module for the texture

command processing engine. (sq.v, 44:24-47:23.)

       83.   The code which specifies the CFS module is replicated below.

```
module sq_ctl_flow_seq
(
    cfs_type_strap,    // 00:alu0, 01:tex, 10:alu1

    is_phase,          // 00:CF, 01:Tex, 10:ALU, 11:CP
    is_subphase,       // 00:alu0, 01:tex, 10:alu1, 11:tex
    cfs_phase,         // 00:alu0, 01:tex, 10:alu1, 11:tex

    // local registers
    // - per chip
    inst_base_vtx,     // vertex base
    inst_base_pix,     // pixel base

    // - per context
    vs_program_base_set,    // connected to SQ_VS_PROGRAM.BASE (12
bits)
    ps_program_base_set,    // connected to SQ_PS_PROGRAM.BASE (12
bits)

    // thread arbiter input
    arb_rts,           //
    arb_state,         //
    arb_status,        //
    arb_thread_type,   // vertex or pixel
    cfs_rtr_q,         // CFS can take a new packet

    pc_base_q,         // parameter cache base write pointer

    // instruction store interface
    is_read_addr_q,    // instruction store read address
    is_read_data_q,    // instruction store read data

    // output to the thread buffer (for thread updates)
    cfs_update_q, // load updated status info from CFS
    cfs_state,         //
    cfs_status,   //

    // outputs to the target instruction fetcher
    cfs_rts_q,         // ctl packet and ptr are valid
    cfs_ctl_pkt_q,     // the control packet (lod_correct, valid_bits,
gpr_base, context_id)
    //cfs_pc_base_q,   // param cache base - part of cfs_state...
```

- 38 -

```
    cfs_tgt_instr_ptr_q,     // the instr store address of the first
target instruction
    cfs_tgt_instr_cnt_q,     // the number of target instructions to be
fetched
    cfs_thread_type_q, // vertex or pixel
    tif_rtr,                 // TIF can take a new packet

    global_export_id,  // from sq_exp_alloc
    //cfs_export_id,   // to sq_exp_alloc (part of cfs_state)
    cfs_tif_xfc,       // to sq_exp_alloc

    busy,
    clk,
    reset
);
```

<div align="right">(sq_ctl_flow_seq.v, 2:6-4:5.)</div>

84.   The CFS module includes an arbiter interface, as replicated below:

```
// - thread arbiter input
arb_rts,            //
arb_state,          //
arb_status,         //
arb_thread_type,    // vertex or pixel
```

<div align="right">(sq_ctl_flow_seq.v, 2:23-3:1.)</div>

85.   The signals in the CFS-arbiter interface show the arbiter providing the selected command thread's state (*arb_state*), status (*arb_state*) and type (*arb_thread_type*) information to the CFS module. For example, *arb_rts* communicates information which indicates that the arbiter is ready to provide the selected command thread to the CFS module. (sq_ctl_flow_seq.v, 2:24), *arb_state* communicates the command thread's state information (sq_ctl_flow_seq.v, 2:25), *arb_status* communicates the command thread's status information (sq_ctl_flow_seq.v, 2:26), and *arb_thread_type* communicates information which

- 39 -

identifies the command thread as a vertex command thread or a pixel command

thread (sq_ctl_flow_seq.v, 3:1.)

86. The CFS module uses the *arb_state*, *arb_status*, and *arb_thread_type*

inputs that it receives from the arbiter to calculate the address of the command

thread's first instruction in the instruction store and the number of command thread

instructions that require processing. For example, the CFS module determines the

command thread's first instruction using the circuit described by the source code

below:

```
always @(posedge clk)
    begin
    if ( arb_xfc )
        cfs_exec_cnt_q <= arb_state[`SQ_CFS_STATE_WIDTH-
14:`SQ_CFS_STATE_WIDTH-17];
    else if ( inc_exec_cnt )
        cfs_exec_cnt_q <= cfs_exec_cnt_q + 1;
    else if ( clr_exec_cnt )
        cfs_exec_cnt_q <= 4'h0;
    else
        cfs_exec_cnt_q <= cfs_exec_cnt_q;
    end

....
....
....
// --------------------------------------
   // -- Target Instruction String --
   // --------------------------------------
   // - string of 9 {serial, resource} pairs from CF EXEC instr read
out of ppb
   //   - the CF instr count says how many of these pairs are valid
   // - the exec_cnt status says how many have already been executed
(sent to TIF)
   //   - the exec_cnt status is used to align TI string data out of
the PPB when initially loaded
always @(posedge clk)
    begin
    if (ld_tip)
```

- 40 -

```
case ( cfs_exec_cnt_q )
  4'h0: tgt_instr_str_q <= { 0'b0, ppb_read_data[33:16]};
  4'h1: tgt_instr_str_q <= { 2'b0, ppb_read_data[33:18]};
  4'h2: tgt_instr_str_q <= { 4'b0, ppb_read_data[33:20]};
  4'h3: tgt_instr_str_q <= { 6'b0, ppb_read_data[33:22]};
  4'h4: tgt_instr_str_q <= { 8'b0, ppb_read_data[33:24]};
  4'h5: tgt_instr_str_q <= {10'b0, ppb_read_data[33:26]};
  4'h6: tgt_instr_str_q <= {12'b0, ppb_read_data[33:28]};
  4'h7: tgt_instr_str_q <= {14'b0, ppb_read_data[33:30]};
  4'h8: tgt_instr_str_q <= {16'b0, ppb_read_data[33:32]};
  default: tgt_instr_str_q <= {18{X}};
endcase
else if (shift_ti_str)
  begin
    tgt_instr_str_q <= {2'b0, tgt_instr_str_q[17:2]};
  end
else
  begin
    tgt_instr_str_q <= tgt_instr_str_q;
  end
end
....
....
....


always @(posedge clk)
  begin
   if ( ld_tip )
    begin
        cfs_tgt_instr_ptr_q <= ppb_read_data[11:00] +
program_base + cfs_exec_cnt_q;
        ppb_instr_cnt_minus_one_q <= ppb_read_data[15:12] - 1;
      ppb_instr_op_q <= ppb_instr_op;
    end
  end
```

            (sq_ctl_flow_seq.v, 15:18-16:1, 24:22-26:1, and 26:17-25.)

87.   The CFS module determines the number of command thread

instruction(s) that require processing using the logic below:


```
// --------------------------------------
// -- Target Instruction Counter (TIC) --
// --------------------------------------
// - increment for every seqential target instruction being sent to
the TIF for the current thread
// - clear TIC when clearing the exec_cnt (for now will not try to
continue TIC from one exec instr
```

- 41 -

```
//     to the next)

always @(posedge clk)
  begin
   if ( reset | clr_tic )  tic_q <= 0;
   else if ( inc_tic )     tic_q <= tic_q + 1;
   else                    tic_q <= tic_q;
  end

wire [11:0] cfs_tgt_instr_cnt_q = tic_q;
```

(sq_ctl_flow_seq.v, 27:20-28:7.)

88.   The CFS module determines whether the command thread is a vertex

command thread or a pixel command thread using the logic below:

```
cfs_thread_type_q <= arb_thread_type;
```

(sq_ctl_flow_seq.v, 18:4.)

89.   The CFS module then transmits the command thread which includes

the command thread's first instruction's address, the number of instructions, and

the command thread's type to the TIF module, using the interface replicated below:

```
    // outputs to the target instruction fetcher
...
...
...

   cfs_tgt_instr_ptr_q,    // the instr store address of the first
target instruction
   cfs_tgt_instr_cnt_q,    // the number of target instructions to be
fetched
   cfs_thread_type_q, // vertex or pixel
```

(sq_ctl_flow_seq.v, 3:15-21.)

- 42 -

90.   The signal *cfs_thread_type_q* identifies the command thread type (vertex or pixel); *cfs_tgt_instr_ptr_q* identifies the starting address of the command thread's first instruction; and *cfs_tgt_instr_cnt_q* identifies the number of instructions that require processing.

91.   Additionally, the CFS module also passes the command thread's identifier as *arb_status*[21-16] to the TIF module. (*See* sq.v, 58:24, 67:14, and 73:23.)

92.   The TIF module receives a command thread from the CFS module. The TIF module uses the pointer to the command thread's first instruction (*cfs_tgt_instr_pt*r_q) and the number of instructions that require processing (c*fs_tgt_instr_cnt_q*) to fetch the command thread's instruction(s) from the instruction store. The R400 RTL code defines the TIF module as *sq_target_instr_fetch* module in sq_target_instr_fetch.v.

93.   The sq.v file instantiates three instances of the *sq_target_instr_fetch* module, one instance per command processing engine. The sq.v file instantiates an instance of the *sq_target_instr_fetch* module for each of the ALU processing engines, *u0_sq_alu_instr_fetch* for the first ALU processing engine and *u1_sq_alu_instr_fetch* for the second ALU processing engine. (sq.v, 66:13-68:23, 72:17-74-26.) Additionally, the sq.v file also instantiates an instance of

- 43 -

sq_target_instr_fetch called *u_sq_tex_instr_fetch* module for the texture

processing engine. (sq.v, 57:22-59:25.)

94.   The TIF module uses the output from the CFS module to retrieve the

command thread's instruction(s) from the instruction store. For example, the TIF

module receives the output from the CFS module, using the source code below:

```
    // cfs interface
...
...
...

    cfs_instr_ptr,      // the Instruction Store address of the first
target instruction
    cfs_instr_cnt,      // the number of instructions to be fetched
...
...
...

    cfs_thread_type,    // vertex or pixel
    cfs_thread_id,      //
```

<div align="right">(sq_target_instr_fetch.v, 2:19-3:1.)</div>

95.   The TIF module uses the inputs received from the CFS module to

fetch the command thread's instruction from the instruction memory. Each of the

u0_sq_alu_instr_fetch, *u1_sq_alu_instr_fetch*, and *u_sq_tex_instr_fetch* modules

fetches the command thread's instruction from the instruction memory for the first

and second ALU processing engine and the texture processing engine respectively.

96.   The TIF module also includes an interface with the

sq_instruction_store module defined in sq_instruction_store.v. As I discussed

- 44 -

above, the instruction store module stores the command thread's instruction(s).

The TIF module's interface with the instruction store is 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)
```

<div align="right">(sq_target_instr_fetch.v, 3:5-9.)</div>

97.   The TIF module uses the *is_read_addr* interface to send the command

thread's instruction pointer that communicates the address of the command

thread's instruction to the interface store module, using the R400 RTL code below:

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

<div align="right">(sq_target_instr_fetch.v, 5:10.)</div>

```
assign is_read_addr = tip_q;
```

<div align="right">(sq_target_instr_fetch.v, 8:2.)</div>

```
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
```

<div align="right">(sq_target_instr_fetch.v, 8:22-9:4.)</div>

98.   As I discussed above, the instruction store module defined in

sq_instruction_store.v stores the command thread's instruction(s). The instruction

store module is capable of receiving three *is_read_addr* requests, one from each of

- 45 -

the TIF modules (*u0_sq_alu_instr_fetch*, *u1_sq_alu_instr_fetch*, and

u_sq_tex_instr_fetch), using the interface below:

```
// SQ
input [11:0] i_tex_addr;
input [11:0] i_alu0_addr;
input [11:0] i_alu1_addr;
```

(sq_instruction_store.v, 2:17-20.)

99.   In response to the is_read_addr request, the instruction store module

retrieves the command thread's instruction(s) and outputs them as the *o_is_data*

signal, using the R400 RTL code below:

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

(sq_instruction_store.v, 2:26.)

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

(sq_instruction_store.v, 3:17.)

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

*(*sq_instruction_store.v, 7:19.*)*

```
// 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;
```

- 46 -

```
    q_rbi_addr_in <= d_rbi_addr_in;
     end
  end
```
<div align="right">(sq_instruction_store.v, 15:11-16:6.)</div>

100.   The TIF module receives the command thread's instruction(s) from

the instruction store module using the R400 RTL code below:

```
    input  [95:0] is_read_data;
```
<div align="right">(sq_target_instr_fetch.v, 5:11.)</div>

101.   The instructions are then loaded into the TIF 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
```
<div align="right">(sq_target_instr_fetch.v, 12:7-17.)</div>

102.   The TIF module also provides the thread type and the thread identifier

inputs from the CFS module using the TIF modules output signals, described

below:

```
         isr_thread_type_q <= cfs_thread_type;
         isr_thread_id_q <= cfs_thread_id;
...
...
...
```

- 47 -

```
tif_thread_type_q <= isr_thread_type_q;
tif_thread_id_q <= isr_thread_id_q;
```

(sq_target_instr_fetch.v, 10:10-11, 11:18-19.)

103.   Once the TIF module provides the command thread's instruction and

the CFS module's inputs using the TIF module's output signals, the TIF module

transmits the command thread to an AIQ module or TIQ module (depending on

whether the TIF module is associated with one of the ALU processing engines or a

texture processing engine) 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)
```

(sq_target_instr_fetch.v, 3:17-19.)

104.   With respect to the ALU processing engine, the TIF module passes

the command thread's instruction(s) to the AIQ module called *sq_alu_instr_queue*

module. The AIQ 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 AIQ

modules called *u0_sq_alu_instr_queue* (sq.v, 68:26-70:17) and

u1_sq_alu_instr_queue (sq.v, 75:2-76:16), one for each ALU processing engine.

- 48 -

105. The AIQ module receives the command thread's instruction from the

TIF 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)
```

<div align="right">(sq_alu_instr_queue.v, 2:14-2:22.)</div>

106. The AIQ modules pass the command thread's instruction to the AIO

module. The R400 RTL code for the AIO module is included in sq_ais_output.v.

The sq.v instantiates a single instance of the AIO module called *u_sq_ais_output*.

(sq.v, 78:16-82:2.)

107. The AIO module receives the command thread's instruction(s) from

the two AIQ modules (*u0_sq_alu_instr_queue* and *u1_sq_alu_instr_queue*) using

the interface below:

```
// inputs from the AIQs
...
...
...
   aiq0_instr,      // instruction
...
...
...
   aiq1_instr,      // instruction
```

<div align="right">(sq_ais_output.v, 2:8, 2:13, 2:20.)</div>

- 49 -

108. The AIO module acts as a multiplexer between the two instances of

AIQ modules and formats the command thread's instruction(s) from the AIQ

modules into the output signals that the AIO module transmits to the ALU

command processing engine. In particular, the R400 RTL code below shows how

the *aiq0_instr* from the *u0_sq_alu_instr_queue* and *aiq1_instr* from the

u1_sq_alu_instr_queue are provided to the SQ_SP interface. The ALU processing

engine uses the SQ_SP interface to receive and process the command thread's

instructions(s):

```
    // -----------------------------------------------------------------
-----------------------------------------
    // -- registers --
    // -----------------------------------------------------------------
-----------------------------------------

    // -----------------------------------------
    // -- Instruction Input Staging Register --
    // -----------------------------------------
    // - holds the instruction data from the AIQ for use by GPR and PC
writes (is reloaded by other thread
    //  before GPR and PC writes occur, so relavent info must be kept
here)
    // - need to save stall to know whether to assert WE to gprs or PC
also
    // - must reload after every instruction even if AIS is idle to get
the stall info saved
    // - actually need two stages here since the AIQ must be popped for
the next constant access

    always @(posedge clk)
      begin
      if (reset)
        begin
            // stall forces a NOP to the shader pipe
          // - all instruction bits are don't care when stall == 1, so
they don't need to be reset
            // - stall forces WE to GPR and PC to be deasserted
            //isr_scalar_dest_q <= 0;
```

- 50 -

```
                //isr_scalar_mask_q <= 0;
                //isr_vector_dest_q <= 0;
                //isr_vector_mask_q <= 0;
                //isr_pred_sel_q    <= 0;
                //isr_pc_base_q      <= 0;
            isr_instr_stall_q <= HI;
        end


      else if ( (gpr_phase == 2'b11) & (alu_phase == LO) )
//(ais0_ld_isr)
        begin
                isr_scalar_dest_q <= aiq0_instr[15:8];
                isr_vector_dest_q <= aiq0_instr[ 7:0];
                isr_scalar_mask_q <= aiq0_instr[23:20];
                isr_vector_mask_q <= aiq0_instr[19:16];
                isr_pred_sel_q    <= aiq0_instr[60:59];
                isr_pc_base_q      <= aiq0_pc_base;
                isr_instr_stall_q <= ais0_instr_stall;
        end


      else if ( (gpr_phase == 2'b11) & (alu_phase == HI) )
//(ais1_ld_isr)
        begin
                isr_scalar_dest_q <= aiq1_instr[15:8];
                isr_vector_dest_q <= aiq1_instr[ 7:0];
                isr_scalar_mask_q <= aiq1_instr[23:20];
                isr_vector_mask_q <= aiq1_instr[19:16];
                isr_pred_sel_q    <= aiq1_instr[60:59];
                isr_pc_base_q      <= aiq1_pc_base;
                isr_instr_stall_q <= ais1_instr_stall;
        end


      else
        begin
                isr_scalar_dest_q <= isr_scalar_dest_q;
            isr_vector_dest_q <= isr_vector_dest_q;
                isr_scalar_mask_q <= isr_scalar_mask_q;
            isr_vector_mask_q <= isr_vector_mask_q;
                isr_pred_sel_q    <= isr_pred_sel_q;
                isr_pc_base_q      <= isr_pc_base_q;
                isr_instr_stall_q <= isr_instr_stall_q;
        end
      end


    // ISR1 - need to pipe ISR0 to keep it around for the GPR/PC write

    always @(posedge clk)
      begin
      //if (reset)
        //begin
                //isr_scalar_dest_q <= 0;
```

- 51 -

```
        //isr_scalar_mask_q <= 0;
        //isr_vector_dest_q <= 0;
        //isr_vector_mask_q <= 0;
        //isr_pred_sel_q    <= 0;
        //isr_pc_base_q     <= 0;
      //isr_instr_stall_q <= HI;
    //end


  if ( (gpr_phase == 2'b11) )
    begin
        isr_scalar_dest_q1 <= isr_scalar_dest_q;
      isr_vector_dest_q1 <= isr_vector_dest_q;
        isr_scalar_mask_q1 <= isr_scalar_mask_q;
      isr_vector_mask_q1 <= isr_vector_mask_q;
        isr_pred_sel_q1    <= isr_pred_sel_q;
        isr_pc_base_q1     <= isr_pc_base_q;
        isr_instr_stall_q1 <= isr_instr_stall_q;
    end


  else
    begin
        isr_scalar_dest_q1 <= isr_scalar_dest_q1;
      isr_vector_dest_q1 <= isr_vector_dest_q1;
        isr_scalar_mask_q1 <= isr_scalar_mask_q1;
      isr_vector_mask_q1 <= isr_vector_mask_q1;
        isr_pred_sel_q1    <= isr_pred_sel_q1;
        isr_pc_base_q1     <= isr_pc_base_q1;
        isr_instr_stall_q1 <= isr_instr_stall_q1;
    end
  end



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

- 52 -

```
                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
```

- 53 -

```
            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




  // ---------------------------------------
  // -- SP gpr read address, read enable --
  // ---------------------------------------
  // - the read address comes directly from the ALU or Texture
Instruction Queue (IQ)
  // - the read address was calculated prior to being loaded into the
IQ
  // - the read enable is just the RTS out of the IQ
/*
  reg [0:0] aiq_gpr_rd_en;

  always @(alu_phase or aiq0_gpr_rd_en or aiq1_gpr_rd_en)
    begin
      case (alu_phase)
      LO: aiq_gpr_rd_en = aiq0_gpr_rd_en;
      HI: aiq_gpr_rd_en = aiq1_gpr_rd_en;
       endcase
    end
*/

  always @(posedge clk)
   begin
     case (gpr_phase)
      `SQ_SRCA_PHASE: begin
       case (~alu_phase)                   // have to invert this to
get the srcA addr in a cycle early
      LO: SQ_SP_gpr_rd_addr <= aiq0_instr[86:80];
      HI: SQ_SP_gpr_rd_addr <= aiq1_instr[86:80];
        endcase
```

- 54 -

```
         case (~alu_phase)          // have to invert this to
get the srcA addr in a cycle early
             LO: SQ_SP_gpr_rd_en <= aiq0_gpr_rd_en;
             HI: SQ_SP_gpr_rd_en <= aiq1_gpr_rd_en;
             endcase
         end
         `SQ_SRCB_PHASE: begin
           case (alu_phase)
           LO: SQ_SP_gpr_rd_addr <= aiq0_instr[78:72];
           HI: SQ_SP_gpr_rd_addr <= aiq1_instr[78:72];
             endcase
           case (alu_phase)
           LO: SQ_SP_gpr_rd_en <= aiq0_gpr_rd_en;
           HI: SQ_SP_gpr_rd_en <= aiq1_gpr_rd_en;
             endcase
         end
         `SQ_SRCC_PHASE: begin
           case (alu_phase)
           LO: SQ_SP_gpr_rd_addr <= aiq0_instr[70:64];
           HI: SQ_SP_gpr_rd_addr <= aiq1_instr[70:64];
             endcase
           case (alu_phase)
           LO: SQ_SP_gpr_rd_en <= aiq0_gpr_rd_en;
           HI: SQ_SP_gpr_rd_en <= aiq1_gpr_rd_en;
             endcase
         end
         `SQ_FA_PHASE: begin
         SQ_SP_gpr_rd_addr <= tis_gpr_rd_addr;
           SQ_SP_gpr_rd_en   <= tis_gpr_rd_en;
         end
       endcase
     end
```

<div align="right">(sq_ais_output.v, 11:21-23:2.)</div>

109.   The output interface which includes the command thread's

instruction(s) that the AIO module passes to the ALU command processing engine

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,
```

<div align="center">- 55 -</div>

```
...
    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,
```

(sq_ais_output.v, 4:1-4:21.)

110. In particular, the interface includes the *SQ_SP_instruct* signal which provides the command thread's instruction(s).

111. As such, the arbiter is operable to provide the command thread to a command processing engine which is, for example, an ALU processing engine.

112. With respect to the texture processing engine, the TIF module passes the command thread's instruction(s) to the TIQ module called *sq_tex_instr_queue*. This TIQ module calculates the gpr address (the address where the data is located that requires execution) and passes the command thread's instruction(s) to the TIS module. The R400 RTL code for the *sq_tex_instr_queue* module is in sq_tex_instr_queue.v. The sq.v file instantiates a texture instruction queue module called *u_sq_tex_instr_queue* (sq.v, 60:1-61:13).

113. The TIQ module receives the command thread's instruction(s) from the TIF module, using the interface below:

- 56 -

```
// inputs from TIF
...
...
...
    tif_thread_id_q,    //
    tir_q,         // instruction register (TIR)
...
...
...
    tif_thread_type_q, // vector type (0: pixel, 1: vertex)
```

(sq_tex_instr_queue.v, 2:14-19.)

114.   The TIQ module passes the command thread's instruction(s) to the

TIS module called *sq_tex_instr_seq* module, as replicated below:

```
// outputs to TIS
tiq_last_instr,    //
tiq_thread_type,   //
tiq_context_id,    // context_id (from ctl packet)
tiq_valid_bits,    // valid bits (from ctl packet)
tiq_lod_correct,   // lod_correct bits (from ctl packet)
tiq_thread_id,     // thread id
tiq_instr,         // instruction
```

(sq_tex_instr_queue.v, 2:25-3:6.)

115.   The TIS module receives command thread instruction(s) from the TIQ

module and formats the command thread's instruction(s) to the *SQ_TP* interface.

As I discussed above, the TIS module provides the command thread's

instruction(s) to the texture processing engine. The *SQ_TP* interface is used to

transmit the command thread instruction(s) to the texture processing engine. The

R400 RTL code for the TIS module is included in sq_tex_instr_seq.v. The sq.v file

instantiates a TIS module called *u_sq_tex_instr_seq*. (sq.v, 61:16-63:18.)

- 57 -

116. The TIS module receives command thread's instruction from the TIQ

module using the input interface below:

```
    // TIQ interface
...
...
...
    tiq_thread_id,      //
    tiq_instr,          // instruction
```

(sq_tex_instr_seq.v, 2:8-16.)

117. The TIS module propagates the command thread's instruction(s)

received from the TIQ module to the texture processing engine. The instruction(s)

are propagated using the *SQ_TP* interface as shown in the R400 RTL code below:

```
// ---------------------------------------------------------------------
---------------------------------------
    // -- registers --
    // ---------------------------------------------------------------------
---------------------------------------

    // ---------------------------------
    // -- Input Staging Register --
    // ---------------------------------
    // - holds the instruction data from the TIQ while it is sent to
the TP
    // - allows the TIQ read SM to work on the next line in the TIQ
while current data is being sent

    always @(posedge clk)
      begin
      if (ld_isr)
        begin
            isr_last_instr_q  <= tiq_last_instr;
            isr_thread_type_q <= tiq_thread_type;
            isr_context_id_q  <= tiq_context_id;
            isr_valid_bits_q  <= tiq_valid_bits;
            isr_lod_correct_q <= tiq_lod_correct;
            isr_thread_id_q   <= tiq_thread_id;
            isr_instr_q       <= tiq_instr;
        end
      else
        begin
```

- 58 -

```
            isr_last_instr_q  <= isr_last_instr_q;
            isr_thread_type_q <= isr_thread_type_q;
            isr_context_id_q  <= isr_context_id_q;
            isr_valid_bits_q  <= isr_valid_bits_q;
            isr_lod_correct_q <= isr_lod_correct_q;
            isr_thread_id_q   <= isr_thread_id_q;
            isr_instr_q       <= isr_instr_q;
       end
    end


   // TP instruction data output mux and register


   always @(posedge clk)
     begin
      case (tis_current_state)
        TIS0: begin
              SQ_TP_instr <= {isr_instr_q[41:32], isr_instr_q[26:25],
isr_instr_q[19], isr_instr_q[4:0]};
              u0_SQ_TP_pix_mask    <= isr_valid_bits_q  [3:0];
              u1_SQ_TP_pix_mask    <= isr_valid_bits_q  [7:4];
              u2_SQ_TP_pix_mask    <= isr_valid_bits_q [11:8];
              u3_SQ_TP_pix_mask    <= isr_valid_bits_q [15:12];
              u0_SQ_TP_lod_correct <= isr_lod_correct_q [5:0];
              u1_SQ_TP_lod_correct <= isr_lod_correct_q[11:6];
              u2_SQ_TP_lod_correct <= isr_lod_correct_q[17:12];
              u3_SQ_TP_lod_correct <= isr_lod_correct_q[23:18];
              SQ_TP_gpr_wr_addr    <= {isr_instr_q[12],
isr_thread_type_q};   // {gpr_wr_addr[0], type}
              SQ_TP_thread_id      <= isr_thread_id_q[1:0];
     // thread_id[1:0]
              end
         TIS1: begin
              SQ_TP_instr <= isr_instr_q[59:42];
              u0_SQ_TP_pix_mask    <= isr_valid_bits_q [19:16];
              u1_SQ_TP_pix_mask    <= isr_valid_bits_q [23:20];
              u2_SQ_TP_pix_mask    <= isr_valid_bits_q [27:24];
              u3_SQ_TP_pix_mask    <= isr_valid_bits_q [31:28];
              u0_SQ_TP_lod_correct <= isr_lod_correct_q[29:24];
              u1_SQ_TP_lod_correct <= isr_lod_correct_q[35:30];
              u2_SQ_TP_lod_correct <= isr_lod_correct_q[41:36];
              u3_SQ_TP_lod_correct <= isr_lod_correct_q[47:42];
              SQ_TP_gpr_wr_addr    <= isr_instr_q[14:13];
     // gpr_wr_addr[2:1]
              SQ_TP_thread_id      <= isr_thread_id_q[3:2];
     // thread_id[3:2]
              end
         TIS2: begin
              SQ_TP_instr <= {isr_instr_q[78:64], isr_instr_q[62:60]};
              u0_SQ_TP_pix_mask    <= isr_valid_bits_q [35:32];
              u1_SQ_TP_pix_mask    <= isr_valid_bits_q [39:36];
              u2_SQ_TP_pix_mask    <= isr_valid_bits_q [43:40];
              u3_SQ_TP_pix_mask    <= isr_valid_bits_q [47:44];
```

- 59 -

```
          u0_SQ_TP_lod_correct <= isr_lod_correct_q[53:48];
          u1_SQ_TP_lod_correct <= isr_lod_correct_q[59:54];
          u2_SQ_TP_lod_correct <= isr_lod_correct_q[65:60];
          u3_SQ_TP_lod_correct <= isr_lod_correct_q[71:66];
          SQ_TP_gpr_wr_addr     <= isr_instr_q[16:15];
    // gpr_wr_addr[4:3]
          SQ_TP_thread_id       <= isr_thread_id_q[5:4];
    // thread_id[5:4]
          end
      TIS3: begin
          SQ_TP_instr <= {2'b0, isr_instr_q[94:79]};
          u0_SQ_TP_pix_mask    <= isr_valid_bits_q [51:48];
          u1_SQ_TP_pix_mask    <= isr_valid_bits_q [55:52];
          u2_SQ_TP_pix_mask    <= isr_valid_bits_q [59:56];
          u3_SQ_TP_pix_mask    <= isr_valid_bits_q [63:60];
          u0_SQ_TP_lod_correct <= isr_lod_correct_q[77:72];
          u1_SQ_TP_lod_correct <= isr_lod_correct_q[83:78];
          u2_SQ_TP_lod_correct <= isr_lod_correct_q[89:84];
          u3_SQ_TP_lod_correct <= isr_lod_correct_q[95:90];
          SQ_TP_gpr_wr_addr     <= isr_instr_q[18:17];
    // gpr_wr_addr[6:5]
          SQ_TP_thread_id       <= {LO, isr_last_instr_q};
    // end of group
        end
    endcase

    end

  assign tis_gpr_rd_addr  = tiq_instr[11:5];    // send this right
out of the FIFO
  assign texconst_rd_addr = tiq_instr[24:20];   // send this right
out of the FIFO (should it even go in this module?)


    // ------------------------------------
    // -- Constant Data Staging Registers --
    // ------------------------------------
    // - holds the constant data from the TCS while it is sent to the
TP
    // - 1st cycle, low 48 bits from const store go right to output
reg, so const0_q is only 48 bits (to store
    //   the upper half of the first read)
    // -   16 16 16 16 16 16   16 16 16 16 16 16
    //            0  0  0
    //    1  1  1
    //                                2  2  2
    //                        3  3  3

    // - 48 bits
    always @(posedge clk)
      begin
      if (ld_c0) const0_q  <= texconst_rd_data[95:48];
```

- 60 -

```
    else       const0_q  <= const0_q;
    end


// - 96 bits
always @(posedge clk)
  begin
   if (ld_c1) const1_q  <= texconst_rd_data;
   else       const1_q  <= const1_q;
   end


// TP constant data output mux and register

always @(posedge clk)
  begin
   case (tis_current_state)
     TIS0: SQ_TP_const <= texconst_rd_data[47:0];
     TIS1: SQ_TP_const <= const0_q;
     TIS2: SQ_TP_const <= {const1_q[47:32], const1_q[62],
const1_q[30:0]};
     TIS3: SQ_TP_const <= {const1_q[95:63], const1_q[31],
const1_q[61:48]};
    endcase
    end


// misc interface registers

always @(posedge clk)
  begin
   tp_fetch_stall <= TP_SQ_fetch_stall;  // stall input
   SQ_TP_gpr_phase <= gpr_phase;          // gpr phase output
    SQ_TP_vld       <= instr_vld;         // this comes from the TIS
SM
   end
```

(sq_tex_instr_seq.v, 8:21-14:10.)

118.   The final output interface which includes the command thread's

instruction(s) that the texture instruction sequencer module passes to the texture

command processing engine is replicated below:

```
output [0:0]  SQ_TP_vld;
output [17:0] SQ_TP_instr;
output [47:0] SQ_TP_const;
output [1:0]  SQ_TP_gpr_phase;
...
...
```

- 61 -

```
...
    output [1:0]  SQ_TP_gpr_wr_addr;
    output [1:0]  SQ_TP_thread_id;

    output [5:0]  u0_SQ_TP_lod_correct;
    output [3:0]  u0_SQ_TP_pix_mask;
    output [5:0]  u1_SQ_TP_lod_correct;
    output [3:0]  u1_SQ_TP_pix_mask;
    output [5:0]  u2_SQ_TP_lod_correct;
    output [3:0]  u2_SQ_TP_pix_mask;
    output [5:0]  u3_SQ_TP_lod_correct;
    output [3:0]  u3_SQ_TP_pix_mask;
```

(sq_tex_instr_seq.v, 5:7-25.)

119.   In particular, the interface includes the *SQ_TP_instruct* signal which

provides the command thread's instruction(s).

120.   As such, the arbiter is operable to provide the command thread to the

command processing engine, which is a texture processing engine.

3.   *Claim 5*

121.   The preamble of claim 5 recites "*A graphics-processing system,*"

which I already discussed in my analysis of claim 1. This is present in the R400

RTL code for the same reasons as explained in Section VII.A.1.

122.   The first limitation of claim 5 recites "*at least one memory device*

comprising a first portion operative to store a plurality of pixel command threads

and a second portion operative to store a plurality of vertex command threads,"

which I already discussed in my analysis of claim 1. This is present in the R400

RTL code for the same reasons as explained previously in Section VII.A.1.

- 62 -

123.   The second limitation of claim 5 recites "*an arbiter, coupled to the at least one memory device*," which I already discussed in my analysis of claim 1. This is present in the R400 RTL code for the same reasons as explained previously in Section VII.A.1.

124.   The arbiter of claim 5 is "*operable to select a command thread from either of the plurality of pixel command threads and the plurality of vertex command threads.*" I have already discussed my analysis of how the arbiter selects a command thread in claim 1. This is present in the R400 RTL code for the same reasons as explained previously in Section VII.A.1.

125.   The third limitation of claim 5 recites "*a plurality of command processing engines, coupled to the arbiter, each operable to receive and process the command thread.*" As I discussed in my analysis for claims 1 and 2 in Sections VII.A.1 and VII.A.2, the R400 RTL code includes multiple command processing engines – at least an ALU processing engine and a texture processing engine.

126.   The ALU processing engine is specified in the file sp.v and its referenced modules. As I described in my analysis of claim 2 in Section VII.A.2, the AIO module formats the command thread's instruction(s) into an *SQ_SP* interface. The ALU processing engine also includes an *SQ_SP* interface, replicated below:

- 63 -

```
SQ_SP_instruct_start, SQ_SP_instruct, SQ_SP_stall,
 SQ_SP_exp_pvalid, SQ_SP_exporting, SQ_SP_exp_id, SQ_SP_const,
 SQ_SP_gpr_wr_addr, SQ_SP_gpr_rd_addr, SQ_SP_gpr_rd_en,
 SQ_SP_gpr_wr_en, SQ_SP_gpr_phase_mux, SQ_SP_channel_mask,
 SQ_SP_pix_mask, SQ_SP_gpr_input_mux, SQ_SP_auto_count, SC_SP_data,
 SC_SP_valid, SC_SP_type, SC_SP_last_quad, SQ_SP_vsr_data,
 SQ_SP_vsr_double, SQ_SP_vsr_valid, SQ_SP_vsr_read,
 SQ_SP_interp_prim_type, SQ_SP_interp_ijline, SQ_SP_interp_mode,
 SQ_SP_interp_valid, SQ_SP_interp_buff_swap, SQ_SP_interp_gen_i0,
```

(sp.v, 2:2-10.)

127.   The *SQ_SP* interface of the ALU processing engine receives

command thread's instruction(s) from the command thread selected by the arbiter.

128.   The ALU processing engine processes the instruction(s). For example,

the ALU processing engine includes four vector modules called *uvector0*,

uvector1, *uvector2*, and *uvector3*. (sp.v, 14:18-18:2.)

129.   The RTL code for the vector module is included in vector.v. Each of

the vector modules *uvector0*, *uvector1*, *uvector2*, and *uvector3* receives the

command thread's instruction(s). Below, the RTL shows how *uvector0* receives

the command thread's instruction(s):

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

(sp.v, 6:8.)

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

(sp.v, 6:24.)

```
vector uvector0(//outputs
            .sp_sx_data(osp_sx_data0),
            .sp_sx_exporting(sp_exporting),
            .sp_sx_exp_dst(sp_exp_dst),
```

- 64 -

```
        .sp_sx_exp_alu_id(sp_exp_alu_id),
   .sp_sx_exp_pvalid(sp_exp_pvalid),
   .sp_tp_data(sp_fetch_addr0),

   //inputs
   .sq_sp_instruct_start(q_sq_instruct_start),
   .sq_sp_instruct(q_sq_instruct),.sq_sp_stall(q_sq_stall),
   .sclk(sclk), .srst(srst),
   .sq_sp_wr_addr(q_sq_gpr_wr_addr),
.sq_sp_gpr_rd_addr(q_sq_gpr_rd_addr),

.sq_sp_mem_rd_ena(q_sq_gpr_rd_en),.sq_sp_mem_wr_ena(q_sq_gpr_wr_en),.s
q_sp_wr_ena(q_sq_gpr_wr_en),
           .sq_sp_gpr_phase_mux(q_sq_gpr_phase_mux),
.sq_sp_channel_mask(q_sq_channel_mask),
           .sq_sp_pixel_mask(q_sq_pix_mask),
.sq_sp_gpr_input_mux(q_sq_gpr_input_mux),
           .iInterpolated(Interpolated0),// iAutoCount,
           .iVertexIndices(VertexIndex0),
           .sq_sp_constant(q_sq_const),
           .tp_sp_data(q_tp_data0),.tp_sp_gpr_dst(q_tp_gpr_dst),

.tp_sp_gpr_cmask(q_tp_gpr_cmask),.tp_sp_data_valid(q_tp_data_valid),
           .sq_sp_exp_pvalid(q_sq_exp_pvalid),
           .sq_sp_exporting(q_sq_exporting),
           .sq_sp_exp_alu_id(q_sq_exp_alu_id)
           );
```

(sp.v,14:22-15:23.)

130. The vector module receives the command thread instruction(s) over
four cycles and passes the command thread instruction(s) to one of the four MACC
GPR units, instantiated as *macc_gpr0, macc_gpr1, macc_gpr2,* or *macc_gpr3.*
(vector.v, 13:15-16:7.) The first unit, *macc_gpr0,* is replicated in the RTL code
replicated below:

```
input [20:0] sq_sp_instruct; //four cycle transaction
```

(vector.v, 2:6.)

```
//------------------------------------------------------------------
------------------------------------------------------------------
//Instantiation of all four MACC units that create a Vector Unit
```

- 65 -

```
    //------------------------------------------------------------------
------------------------------------------------------------------
    macc_gpr
umacc_gpr0(.oVectorOutput(VectorResult0),.oScalarInput(ScalarInput0),.
oScalarOpcode(ScalarOpcode0),.oRegData(RegData0),.oexport_dst(sq_sp_ex
p_dst),.sq_sp_instruct(sq_sp_instruct),.sq_sp_instruct_start(sq_sp_ins
truct_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_mu
x),.iInterpolated(InputData0),.sq_sp_constant(sq_sp_constant),.iScalar
Data(ScalarData),.tp_sp_data(tp_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));
```

(vector.v, 13:15-14:11.)

131.   The R400 RTL code which defines the MACC GPR module is

included in macc_gpr.v. The MACC GPR module passes the command thread's

instruction(s) to the MACC module, as shown below:

```
//----------------------------------------------------------------------
----------------------------------------------------------------------/
    //Instantiation of the macc unit which does the argument selection
and input modification (swizzling ...etc)
    //1. input for the scalar unit comes as an output from this unit
and goes all the way up to vector.v module where the instance of
scalar unit
    //    can be found.
    //2. VectorResult output is only used as an input into GPRs ....
the Previous Vector Result is not exposed at this level but stays
internal
    //    to macc.v module
    //----------------------------------------------------------------
----------------------------------------------------------------------
/
    //register the output from GPRs
    reg [127:0]    q_RegData;

    macc
umacc(.oResult(VectorResult),.oScalarOpcode(oScalarOpcode),.oScalarInp
ut(oScalarInput),.oExportDst(oexport_dst),.iRegData(q_RegData),.iConst
antData(sq_sp_constant),.iScalarData(iScalarData),.iInstruction(sq_sp_
instruct), .iInstStart(sq_sp_instruct_start), .sclk(sclk),
.srst(srst));
```

(macc_gpr.v, 3:1-3:20.)

132. The R400 RTL code which defines the MACC module is included in

macc.v. The MACC module receives the command thread's instruction(s) as signal

iInstruction (macc.v, 2:24), and processes the instruction as described in macc.v,

2:24-28:6.

133. The texture processing engine is defined as a tp module in sp.v. As I

described in my analysis of claim 1 in Section VII.A.1, the TIS module provides

the command thread's instructions to the texture processing engine. For example,

the *TIS* module formats the command thread's instruction(s) into the *SQ_TP*

interface. The *SQ_TP* interface is an interface that transmits command thread's

instructions between the sequencer and the texture processing engine. The texture

processing engine also includes an *SQ_TP* interface which receives the command

thread's instruction(s), as replicated below:

```
SQ_TP_send,
SQ_TP_instr, SQ_TP_const, SQ_TP_gpr_phase, SQ_TP_gpr_wr_addr,
SQ_TP_thread_id, u0_SQ_TP_lod_correct, u0_SQ_TP_pix_mask,
u1_SQ_TP_lod_correct, u1_SQ_TP_pix_mask, u2_SQ_TP_lod_correct,
u2_SQ_TP_pix_mask, u3_SQ_TP_lod_correct, u3_SQ_TP_pix_mask,
```

(sq.v,2:7-11.)

134. In particular, the input *SQ_TP_instr* receives the command thread's

instruction(s).

135. The texture processing engine also processes the command thread's

instruction(s). For example, the texture processing engine includes a *tp_input*

- 67 -

module called *utp_input*. (tp.v, 22:16-24:4.) The *tp_input* module receives the

SQ_TP_instr, as shown below:

```
tp_input utp_input

    (
    .sclk(sclk_global),
    .srst(srst),
    .SP_TP_fetch_addr3(SP_TP_fetch_addr3),
    .SP_TP_fetch_addr2(SP_TP_fetch_addr2),
    .SP_TP_fetch_addr1(SP_TP_fetch_addr1),
    .SP_TP_fetch_addr0(SP_TP_fetch_addr0),
    .SQ_TP_lod_correct(SQ_TP_lod_correct),
    .SQ_TP_pix_mask(SQ_TP_pix_mask),
    .SQ_TP_const(SQ_TP_const),
    .SQ_TP_instr(SQ_TP_instr),
    .SQ_TP_gpr_wr_addr(SQ_TP_gpr_wr_addr),
...
...
...

    ); // utp_input
```

(tp.v, 22:16-24:4.)

136. The R400 RTL code which defines the *tp_input* module is included in

tp_input.v. The *tp_input* receives the command thread's instruction as signal

SQ_TP_instr (tp_input.v, 3:5) and processes the command thread's instruction as

described in tp_input.v, 5:13-19:6.

4.   *Claim 6*

137. Claim 6 recites the "*graphics-processing system of claim 5, wherein

the plurality of command processing engines comprises at least one arithmetic

logic unit.*" In my analysis of claim 5 in Section VII.A.3, I have already discussed

that the ALU processing engine includes at least one ALU logic unit. The ALU

- 68 -

logic unit is capable of performing vector and scalar operations. The R400 RTL

code that performs the vector and scalar operations is included in sp.v, vector.v,

and scalar_lut.mc (and their referenced modules).

138.   As such, the R400 RTL code embodies a plurality of command

processing engines that comprises at least one arithmetic logic unit.

5.   Claim 7

139.   Claim 7 recites the *"graphics processing system of claim 5, wherein*

the plurality of command processing engines comprises at least one texture

processing engine."

140.   In my analysis of claim 5 in Section VII.A.3, I have already discussed

that the command processing engine can be a texture processing engine. The R400

RTL code includes a texture processing engine as a tp module in tp.v.

141.   As such, the R400 RTL code embodies a plurality of command

processing engines that comprises at least one texture processing engine.

B.   The R400 Emulator Code Describing Claims 1, 2, 5, 6, and 7

142.   I also examined the R400 Emulator Code. Chip designers would

ordinarily use the emulator code such as this to design the integrated circuits as

part of their design process. The evidence I have reviewed indicates that this is

what ATI did at the time this R400 Emulator Code was created.

- 69 -

143.   The R400 Emulator Code is written in the C++ programming

language. The C++ language supports object-oriented programming where objects

can be user-defined data types. An object is defined in terms of a class, which

serves as a template for a type of data. A class can define variables, interfaces, and

functions (called methods in C++) for an object. These variables and functions can

be public or private. The public variables or functions of an object can be accessed

by other objects, while the private variables and functions can be accessed only

within the object itself. For example, public variables and functions may transmit

data or provide data access between objects, while private variables and functions

may manipulate data inside the object.

144.   When the C++ code is compiled the static objects that are defined by

the classes are instantiated in memory. When the C++ code is executed, the

dynamic objects are instantiated in memory and all objects operate and interact as

described in the corresponding C++ source code.

145.   The R400 Emulator Code is distributed among numerous files

including sq_block_model.cpp, arbiter.cpp, sq_alu.cpp, gpr_manager.cpp,

instruction_store.cpp and reg_file.cpp, tp.cpp and corresponding "include" files

(generally with names of the form that end with a .h). The source code in these

files, when compiled, realizes the graphics-processing system recited in claims 1,

2, 5, 6, and 7. These files define a set of objects that act as components of the

claimed graphics-processing system. In C++, the ".cpp" files generally describe

the behavior of the functions of the classes, while the ".h" files generally define the

classes, including the public and private variables of the classes, and may also

include functions that act on the public and private variables.

146.   Below, I describe relevant classes and the corresponding objects that

implement the elements of claims 1, 2, 5, 6, and 7.

1.   *Claim 1*

a.   The Preamble

147.   The preamble of claim 1 recites "*A graphics processing system.*" All

of the C++ files identified above are components of the graphics-processing

system.

148.   The sq_block_model.cpp and the user_block_mode.h define a

cUSER_BLOCK_SQ class which is the sequencer. When compiled, the sequencer

creates a *cUSER_BLOCK_SQ* object that serves as an entry point into the graphics-

processing system and also initializes other objects in the system. For example, the

constructor function "*cUSER_BLOCK_SQ::cUSER_BLOCK_SQ*" of the

cUSER_BLOCK_SQ object, when executed, creates an arbiter object with a call

- 71 -

"*arbiter = new Arbiter(this,m_dumpSQ)*." (sq_block_model.cpp, 7:11.) So, here,

the sequencer creates an instance of the arbiter for the graphics-processing system.

149.   The *cUSER_BLOCK_SQ* object also includes a main function called

cUSER_BLOCK_SQ::Main(). The *cUSER_BLOCK_SQ::Main()* function includes

three functions *Fetch()*, *Process()*, and *Output()* and causes the graphic processing

system to process command threads. (sq_block_model.cpp, 12:6-11.) The

cUSER_BLOCK_SQ::Fetch() function gets the vertex and pixel instructions from

different components in the system and stores the vertex and pixel instructions in

the registers. (sq_block_model.cpp,12:12-16:19.) The

cUSER_BLOCK_SQ::Process() function processes the vertex and pixel command

threads that manipulate the vertex and pixel data using an Arbiter

(sq_block_model.cpp, 38:6-38:16), and the *cUSER_BLOCK_SQ::Output()*

function outputs the processed vertex and pixel data. (sq_block_model.cpp, 38:18-

43:6.)

150.   The *cUSER_BLOCK_SQ::Process()* function processes vertex and

pixel command threads. The *cUSER_BLOCK_SQ::Process()* function does so by

calling three functions: *ProcessVerts()*, *ProcessPixels()*, and *arbiter-

>Execute()*.(sq_block_model.cpp, 38:6-38:16.) The

cUSER_BLOCK_SQ::ProcessVerts() and *cUSER_BLOCK_SQ::ProcessPixels()*

- 72 -

functions belong to the *cUSER_BLOCK_SQ* class and each receive respective

vertex and pixel data as inputs and store the vertex and pixel data in registers.

151.   The *cUSER_BLOCK_SQ::ProcessVerts()* function also generates a

command thread that is associated with a particular vertex data and inserts the

command thread into the portion of the memory associated with the vertex

command threads.

```
arbiter->
AddVector(0,VERTEX,eventId,valids,true,interp[0].lod_correct)
```

(sq_block_model.cpp, 34:23.)

```
arbiter-> AddVector(base_ptr,VERTEX,
vState,valids,false,interp[0].lod_correct)
```

(sq_block_model.cpp, 35:24-25.)

152.   The VERTEX variable in the code indicates that the command thread

will be added to the portion of the memory associated with the vertex command

threads. The *cUSER_BLOCK_SQ::ProcessPixels()* function also generates a

command thread that is associated with pixel data and inserts the command thread

into a portion of the memory associated with the pixel command threads.

```
arbiter->
AddVector(0,PIXEL,event,interp[buf_read].pix_mask,true,interp[buf_read
].lod_correct)
```

(sq_block_model.cpp, 20:26.)

- 73 -

```
arbiter-> AddVector(base_ptr,PIXEL,
    interp[buf_read].state_id,interp[buf_read].pix_mask,false,
    interp[buf_read].lod_correct);
```

(sq_block_model.cpp, 24:5.)

153. The PIXEL variable in the code indicates that the command thread will be added to the portion of the memory associated with the pixel command threads.

154. The portion of the memory that stores the vertex command threads is called a vertexStation and the portion of the memory that stores the pixel command threads is called a pixelStation. Additionally, an instruction store which is defined in the instruction_store.h defines the IStore object which stores the command thread's instructions and is also part of the portion of the memory. I will address each of these memory structures below.

155. The *arbiter->Execute()* function selects a command thread from among the command threads stored in either a pixelStation or the vertexStation and passes the selected command thread to an ALU processing engine or a texture processing engine. (sq_block_model.cpp, 38:14-15.) I address the *arbiter->Execute()* function below.

156. As such the R400 Emulator Code describes the graphics-processing system of claim 1.

- 74 -

b.   The at Least One Memory Device

157.   The first limitation of claim 1 recites *"at least one memory device*

comprising a first portion operative to store a plurality of pixel command threads

and a second portion operative to store a plurality of vertex command threads;"

158.   The arbiter.h file defines a structure called a *ReservationStation*. The

ReservationStation stores command threads and is replicated below:

```
struct ReservationStation
{
    ReservationStation_data data;
    ReservationStation_status status;
};
```

(arbiter.h, 3:13-17.)

159.   The *ReservationStation* structure includes two components –

ReservationStation_data and *ReservationStation_status*. Both

ReservationStation_data and *ReservationStation_status* are structures defined in

arbiter.h, 2:7-3:13. ReservationStation_data stores instruction information related

to the command thread, including pointers to the memory space where the

instructions are located as shown below:

```
struct ReservationStation_data
{
    unsigned int cfPtr;
     unsigned int execCount;
     int16        loopIter[4];
     int16        loopCount[4];
     int16        callReturn[4];
     bool         predicates[64][4];
     bool         exportId;
     unsigned int pcBasePtr;
```

- 75 -

```
    unsigned int gprBase;
    unsigned int state;
    int            LodCorrect[4][4];
    unsigned int valids[4][4];
  int            vCount;
};
```

<div align="right">(arbiter.h, 2:7-22.)</div>

160.   *ReservationStation_status* stores status information related to the

command thread, including whether the instruction is a first or last instruction in

the command thread as shown below:

```
struct ReservationStation_status
{
    bool            valid;
    Ressource_type   ressourceNeeded;
    bool            texReadsOutstanding;
    bool            serial;
    Allocation_type  allocation;
    unsigned int     allocationSize;
    bool            posAllocated;
    bool            first;
    bool            event;
    bool            last;
    bool            pulseSx;
};
```

<div align="right">(arbiter.h, 2:24-3-11.)</div>

161.   The *Arbiter* class defined in arbiter.h defines two arrays of type

ReservationStation. An array stores multiple instances of data having the same

type. Here, an array of type ReservationStation stores multiple instances of

ReservationStation structures and thus this array of type *ReservationStation* stores

a plurality of command threads, as recited in claim 1.

162.   The first such array is called *pixelStation*, and is defined as:

<div align="center">- 76 -</div>

ReservationStation pixelStation[MAX_PIX_RESERVATION_SIZE];

(arbiter.h, 4:7.)

163. The *MAX_PIX_RESERVATION_SIZE* indicates that the *pixelStation*

array stores 48 command threads, as shown by the following statement:

#define MAX_PIX_RESERVATION_SIZE 48

(arbiter.h, 2:4.)

164. The *pixelStation* array is invoked to allocate memory when the arbiter

is created using the new *Arbiter()* functions call described above. This means that

the *pixelStation* array is a memory portion that is operative to store a plurality of

pixel command threads.

165. The second array is called a vertexStation, and is defined as:

ReservationStation vertexStation[MAX_VTX_RESERVATION_SIZE];

(arbiter.h, 4:6.)

166. The *MAX_VTX_RESERVATION_SIZE* indicates that the *vertexStation*

array stores 16 command threads, as shown by the following statement in arbiter.h:

#define MAX_VTX_RESERVATION_SIZE 16

(arbiter.h, 2:5.)

- 77 -

167. The vertexStation array is also invoked to allocated memory when the *newArbiter()* function call occurs. This means that the *vertexStation* array is a memory portion that is operative to store a plurality of vertex command threads.

168. Below, I have generated a figure, based on my understanding of the R400 Emulator Code, that represents the pixel reservation station and the vertex reservation station that includes these components as they are instantiated in the R400 Emulator Code, along with the structures and/or classes that define the components:

Pixel Reservation Station	Vertex Reservation Station
pixelStation[MAX_PIX_RESERVATION_SIZE] (arbiter.h)	vertexStation[MAX_VTX_RESERVATION_SIZE] (arbiter.h)
struct ReservationStation (arbiter.h)	struct ReservationStation (arbiter.h)
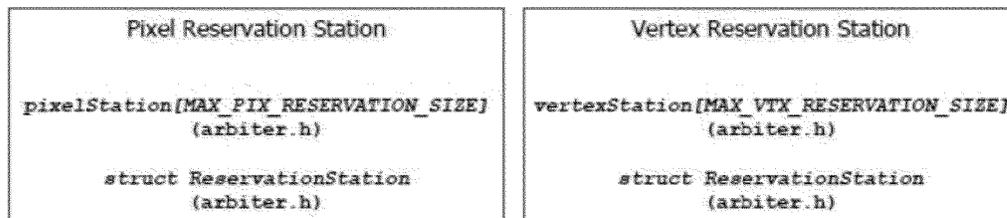
c.   The Arbiter

169. The second limitation of claim 1 recites "*an arbiter, coupled to the at least one memory device.*"

170. As explained above, the arbiter.cpp and arbiter.h files define an Arbiter class which is used to generate an arbiter object. For example, the Arbiter object is created in the constructor of the *cUSER_BLOCK_SQ* class *cUSER_BLOCK_SQ::cUSER_BLOCK_SQ* by the below code:

- 78 -

```
arbiter = new Arbiter(this,m_dumpSQ);
```

(sq_model_block.cpp, 7:11.)

171.   The keyword "new" is a C++ keyword that creates a dynamic object,

in this case an arbiter, when the R400 Emulator code executes. As part of the

object creation, the "new" call also invokes the arbiter's constructor called

Arbiter::Arbiter() which creates and initializes the arbiter recited in claim 1.

(arbiter.cpp, 2:3-9:16). The arbiter's constructor initializes the *pixelStation* array

which is a pixel portion of the memory and the *vertexStation* array which is a

vertex portion of the memory (arbiter.cpp, 3:2-4:14 and 4:16-5:21) as replicated

below:

```
// initialize all the fiels of the RS
for (i=0;i<MAX_PIX_RESERVATION_SIZE;i++)
{
        pixelStation[i].status.valid = false;
        pixelStation[i].status.event = false;
        pixelStation[i].status.first = false;
        pixelStation[i].status.last = false;
        pixelStation[i].status.texReadsOutstanding = false;
        pixelStation[i].status.pulseSx = false;
        pixelStation[i].status.allocation = SQ_NO_ALLOC;

        pixelStation[i].data.callReturn[0] = -1;
        pixelStation[i].data.callReturn[1] = -1;
        pixelStation[i].data.callReturn[2] = -1;
        pixelStation[i].data.callReturn[3] = -1;

        pixelStation[i].data.loopIter[0] = -1;
        pixelStation[i].data.loopIter[1] = -1;
        pixelStation[i].data.loopIter[2] = -1;
        pixelStation[i].data.loopIter[3] = -1;

        pixelStation[i].data.loopCount[0] = 0;
        pixelStation[i].data.loopCount[1] = 0;
        pixelStation[i].data.loopCount[2] = 0;
```

```
pixelStation[i].data.loopCount[3] = 0;

pixelStation[i].data.pcBasePtr = 0;
pixelStation[i].data.exportId =0;
pixelStation[i].data.vCount =0;

for (j=0;j<64;j++)
{
        pixelStation[i].data.predicates[j][0]= false;
        pixelStation[i].data.predicates[j][1]= false;
        pixelStation[i].data.predicates[j][2]= false;
        pixelStation[i].data.predicates[j][3]= false;
}

}

for (i=0;i<MAX_VTX_RESERVATION_SIZE;i++)
{
    vertexStation[i].status.valid = false;
    vertexStation[i].status.event = false;
    vertexStation[i].status.first = false;
    vertexStation[i].status.last = false;
    vertexStation[i].status.texReadsOutstanding = false;
    vertexStation[i].status.pulseSx = false;
    vertexStation[i].status.allocation = SQ_NO_ALLOC;

    vertexStation[i].data.callReturn[0] = -1;
    vertexStation[i].data.callReturn[1] = -1;
    vertexStation[i].data.callReturn[2] = -1;
    vertexStation[i].data.callReturn[3] = -1;

    vertexStation[i].data.loopIter[0] = -1;
    vertexStation[i].data.loopIter[1] = -1;
    vertexStation[i].data.loopIter[2] = -1;
    vertexStation[i].data.loopIter[3] = -1;

    vertexStation[i].data.pcBasePtr = 0;
    vertexStation[i].data.exportId =0;
    vertexStation[i].data.vCount =0;

    for (j=0;j<64;j++)
    {
            vertexStation[i].data.predicates[j][0]= false;
            vertexStation[i].data.predicates[j][1]= false;
            vertexStation[i].data.predicates[j][2]= false;
            vertexStation[i].data.predicates[j][3]= false;
    }
}
```

(arbiter.cpp, 3:2-5:21.)

- 80 -

172.   Another example of initialization occurs at arbiter.cpp, 3:4-6:6 where

the arbiter's *Arbiter::init()* function clears and re-initializes the *pixelStation* and

vertexStation arrays. Initialization of the memory that stores *pixelStation* and

vertexStation by the arbiter object demonstrates that claimed arbiter is coupled to

the claimed portion of the memory as recited in claim 1.

173.   I have generated below a figure, based on my understanding of the

R400 Emulator Code, that represents how the arbiter is coupled to the pixel

reservation station and the vertex reservation station. The figure includes the

components as they are instantiated in the graphics-processing system and also

includes the structures and/or classes that define the components:
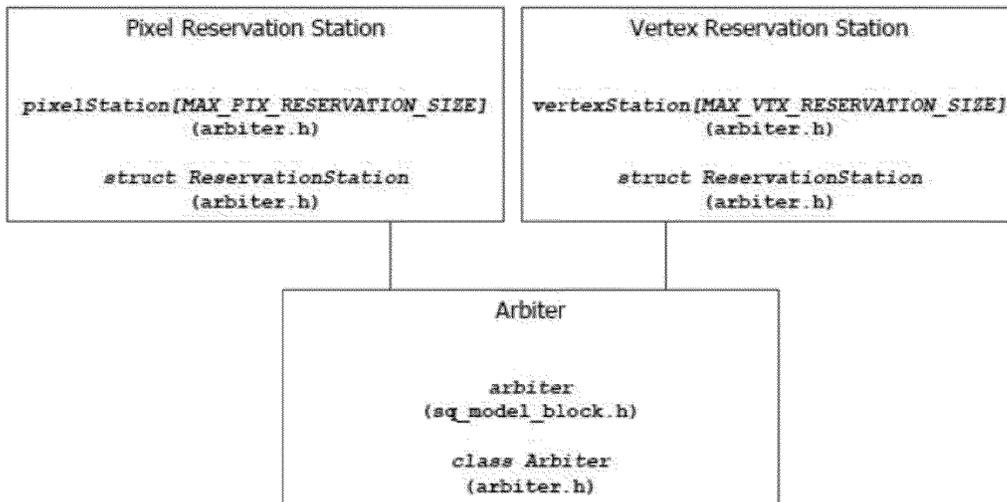
d. The Arbiter is Operable to Select a Command Thread

174. The arbiter of claim 1 is "*operable to select a command thread from either of the plurality of pixel command threads and the plurality of vertex command threads based on relative priorities of the plurality of pixel command threads and the plurality of vertex command threads.*"

175. The arbiter.cpp file describes an arbiter operable to select a thread. For example, The *Arbiter::Execute()* function allows the arbiter object to select a command thread. (arbiter.cpp, 9:18-16:18.) The *Execute()* function chooses a command thread that is processed on a texture processing engine or an ALU processing engine.

176. To choose a command thread for processing using a texture processing engine, the *Arbiter::Execute()* function calls an *Arbiter::chooseTexStation(...)* function, using the call below:

```
found=chooseTexStation(texLineNumber,texType);
```

(arbiter.cpp, 10:16.)

177. The function *Arbiter::chooseTexStation(...)* selects a command thread from either a *pixelStation* or *vertexStation*. (arbiter.cpp, 51:7-53:22.)

- 82 -

178.   For example, *Arbiter::chooseTexStation(...)* includes two "for" loops

that traverse the *pixelStation* entries and then the *vertexStation* entries. The first

"for" loop iterates through the pixelStation array, beginning at the top of the array,

and selects a candidate pixel command thread (if any such entry is present) from a

location in the *pixelStation*. The selection is based on the variety of status tests.

The code for selecting the pixel command thread is replicated below:

```
// do pixels first
lineCheck = pixelHead;
for (i=0;i<pixelRsCount;i++)
{
        if (pixelStation[lineCheck].status.valid &&
pixelStation[lineCheck].status.ressourceNeeded == TEXTURE
              && !pixelStation[lineCheck].status.event)
        {
              pixelPick=lineCheck;
        }
        // enforce restrictions based on the status
        if (pixelPick != -1)
        {
              // no texture ops while texture reads are outstanding
              if
(pixelStation[pixelPick].status.texReadsOutstanding)
                     pixelPick = -1;
              else
                     break;
        }

        lineCheck = (lineCheck+1)%MAX_PIX_RESERVATION_SIZE;
    }
}
```

(arbiter.cpp, 51:14-52:9.)

179.   By evaluating the *pixelStation* array entries and by selecting a

candidate pixel command thread based on status conditions (arbiter.cpp, 51:14-

52:9), the R400 Emulator Code describes an arbiter operable to select a pixel

- 83 -

command thread from the plurality of pixel command threads based on the relative

priorities of the plurality of pixel command threads.

180.   The second "for" loop in the *Arbiter::chooseTexStation()* function

iterates through the *vertexStation* entries, beginning at the top of the *vertexStation*

array, and selects a candidate vertex command thread (if any such entry is present)

from a location in the *vertexStation*. The selection is based on the variety of status

tests. The code for selecting the vertex command thread is replicated below:

```
lineCheck = vertexHead;
for (i=0;i<vertexRsCount;i++)
{
        if (vertexStation[lineCheck].status.valid &&
vertexStation[lineCheck].status.ressourceNeeded == TEXTURE
            && !vertexStation[lineCheck].status.event)
        {
            vertexPick=lineCheck;
        }

        // enforce restrictions based on the status
        if (vertexPick != -1)
        {
            // no texture ops while texture reads are outstanding
            if
(vertexStation[vertexPick].status.texReadsOutstanding)
                vertexPick = -1;
            else
                break;
        }

        lineCheck = (lineCheck+1)%MAX_VTX_RESERVATION_SIZE;
    }
```

(arbiter.cpp, 52:11-53:6.)

181.   By traversing the *vertexStation* after the *pixelStation*, and selecting a

candidate vertex command thread based on different status conditions, the R400

Emulator Code includes an arbiter operable to select a vertex command thread

- 84 -

from the plurality of vertex command threads based on the relative priorities of the

plurality of pixel command threads and the plurality of vertex command threads.

182.   Once the *Arbiter::chooseTexStation(...)* function selects a pixel

command thread from the *pixelStation* and a vertex command thread from the

vertexStation, the *Arbiter::chooseTexStation(...)* function selects a command

thread from the selected pixel command thread and the selected vertex command

thread, where the vertex command thread, if it exists, has priority over the pixel

command thread. That is, if a candidate vertex command thread exists, the vertex

command thread is selected and becomes the selected command thread of claim 1.

Otherwise, the pixel command thread becomes the selected command thread of

claim 1. The code that demonstrates this is replicated below:

```
if (vertexPick != -1)
{
      lineNumber = vertexPick;
      sType = VERTEX;
      return true;
}
if (pixelPick != -1)
{
      lineNumber = pixelPick;
      sType = PIXEL;
      return true;
}
```

(arbiter.cpp, 53:8-19.)

183.   Once the *Arbiter::chooseTexStation(...)* function selects a command

thread, the *Arbiter::chooseTexStation(...)* function returns a boolean (true or false)

variable which indicates whether a command thread was selected. Also,

- 85 -

Arbiter::chooseTexStation(...) returns 1) a type of the selected command thread, a vertex command thread or a pixel command thread, which is stored in variable *sType*, and 2) a location of the selected command thread in the *vertexStation* or *pixelStation* stored in the variable lineNumber. (arbiter.cpp, 51:7.)

184. If the *Arbiter::chooseTexStation(...)* function indicates that the command thread was selected, the Arbiter uses the *Arbiter::popStationVector(...)* function to remove the selected command thread from either the *vertexStation* or *pixelStation* arrays and stores the removed command thread in *texStationData*, as shown by the code below:

```
ReservationStation* texStationData;
int texLineNumber;
Shader_Type texType;

// pick a program to run on the texture pipe
found=chooseTexStation(texLineNumber,texType);
if (found)
{
    texRunning = true;
    // pop the content of the chosen clause and place the results in
the object's texture state
    // variables
    popStationVector(texStationData,texLineNumber,texType);
...
...
...
}
```

(arbiter.cpp, 10:8-11:10.)

185. The *Arbiter::popStationVector(...)* function passes a single *ReservationStation* structure called *texStationData*, the line number location of the selected command thread in the *vertexStation* or *pixelStation* array called

- 86 -

texLineNumber, and the type (vertex or pixel) of the command thread called

texType. The *texStationData* variable is of type *ReservationStation* and stores the

memory address location of the selected command thread. The code that is

operable to select a command thread is replicated below:

```
void Arbiter::popStationVector(ReservationStation*& stationData, int
lineNumber,
                                    Shader_Type sType)
{
     int i,j;
     switch (sType)
     {
     case PIXEL:
          stationData = &pixelStation[lineNumber];
          pixelStation[lineNumber].status.valid = false;
          break;
     case VERTEX:
          stationData = &vertexStation[lineNumber];
          vertexStation[lineNumber].status.valid = false;
          // refresh the vertex mask using vCount
          for (i=0;i<4;i++)
               for (j=0;j<4;j++)
               {
                    switch (vertexStation[vertexTail].data.vCount-
(i*16+j*4))
                    {
                    case 0:
                         vertexStation[vertexTail].data.valids[i][j]
= 0x00;
                         break;
                    case 1:
                         vertexStation[vertexTail].data.valids[i][j]
= 0x01;
                         break;
                    case 2:
                         vertexStation[vertexTail].data.valids[i][j]
= 0x03;
                         break;
                    case 3:
                         vertexStation[vertexTail].data.valids[i][j]
= 0x07;
                         break;
                    case 4:
                         vertexStation[vertexTail].data.valids[i][j]
= 0x0f;
```

- 87 -

```
                break;
        default:
                break;
        };
    }
    break;
    }
}
```

186. In this way, the arbiter is operable to select a command thread from either of the plurality of pixel command threads or the plurality of vertex command threads based on relative priorities of the plurality of pixel command threads and the plurality of vertex command threads.

187. In another example, *Arbiter::Execute()* function also selects a command thread from a *vertexStation* or a *pixelStation* for one of the two ALU processing engines. The two ALU processing engines run on different clock parities, one on the even clock and one on the odd clock. The code which invokes selecting a command thread for an ALU processing engine is replicated below:

```
// pick one alu clause to execute
// depending on the clock parity, run either the even alu state
machine or the odd one
if (ALU_turn)
{
    runALU(alu0Running,
        alu0CFMachine,alu1CFMachine,alu1Running,aluPhase);
}
else
{
    runALU(alu1Running,
        alu1CFMachine,alu0CFMachine,alu0Running,aluPhase);
}
```

188. The *Arbiter::runALU(...)* function (arbiter.cpp, 16:20-24-20) selects a command thread from either a *vertexStation* or a *pixelStation* using the *Arbiter::chooseAluStation(...)* function, as replicated below:

```
found=chooseAluStation(lineNumber,stype,otherAluRunning,
                        otherCFMachine,predToggle);
```

(arbiter.cpp, 17:10-11.)

189. The function *Arbiter::chooseAluStation(...)* selects a command thread from either a *pixelStation* or *vertexStation*. For example, *Arbiter::chooseAluStation(...)* includes two "for" loops. The first "for" loop iterates through the *pixelStation* entries, beginning at the top of the array, and selects a candidate pixel command thread (if any such thread is present) from a location in the *pixelStation*. The selection is based on the variety of status tests that check that the pixel command threads do not block older pixel command thread. The code for selecting the pixel command thread is replicated below:

```
// do pixels first
    lineCheck = pixelHead;
    for (i=0;i<pixelRsCount;i++)
    {
        if (pixelStation[lineCheck].status.valid != 0 &&
pixelStation[lineCheck].status.ressourceNeeded == ALU
            && !pixelStation[lineCheck].status.event)
        {
        // no allocation needed
        if (pixelStation[lineCheck].status.allocation ==
SQ_NO_ALLOC)
        {
            pixelPick = lineCheck;
        }
```

- 89 -

```
                // we need to make sure there is space in the
appropriate buffer
                else if
(pixelStation[lineCheck].status.allocationSize+1 <= sq->pSX_SQ-
>GetExportBuffer()/4)
                {
                        pixelPick = lineCheck;
                }
                // make sure the status says we can pick this vertex
                if (pixelPick != -1)
                {
                        // check for serial with texture pending
                        if (pixelStation[pixelPick].status.serial &&

        pixelStation[pixelPick].status.texReadsOutstanding)
                                pixelPick = -1;
                        // if last is set we can only pick the two
oldests threads
                        else if (pixelStation[pixelPick].status.last &&
                                !(pixelPick==pixelHead ||
pixelPick==((pixelHead-1)%MAX_PIX_RESERVATION_SIZE)))
                                pixelPick = -1;
                        // cannot pick last if texture reads are
outstanding
                        else if (pixelStation[pixelPick].status.last &&

        pixelStation[pixelPick].status.texReadsOutstanding)
                                pixelPick = -1;
                        // can only pick the second to old if the first
is already running
                        else if (pixelStation[pixelPick].status.last &&
                                !pixelStation[pixelHead].status.valid)
                        {
                                if (pixelStation[pixelPick].status.first)
                                        pixelPick = -1;
                                else
                                {
                                        predOn = false;
                                        break;
                                }
                        }
                        else
                                break;
                }
        }// endif pixels

        lineCheck = (lineCheck+1)%MAX_PIX_RESERVATION_SIZE;
    }// end for loop
```

(arbiter.cpp, 54:8-56:9.)

- 90 -

190.   By traversing the *pixelStation* and selecting a candidate pixel command thread based on status conditions, the R400 Emulator Code describes an arbiter operable to select a candidate pixel command thread from the plurality of pixel command threads based on the relative priorities of the plurality of pixel command threads and the plurality of vertex command threads.

191.   The second "for" loop in the *Arbiter::chooseTexStation()* function iterates through the *vertexStation* entries, beginning at the top of the array, and selects a candidate vertex command thread (if any such thread is present) from a location in the *vertexStation*. The selection is based on the variety of status tests. The code for selecting the vertex command thread is replicated below:

```
lineCheck = vertexHead;
for (i=0;i<vertexRsCount;i++)
{
        if (vertexStation[lineCheck].status.valid != 0 &&
vertexStation[lineCheck].status.ressourceNeeded == ALU
            &&!vertexStation[lineCheck].status.event)
        {
            // no allocation needed
            if (vertexStation[lineCheck].status.allocation ==
SQ_NO_ALLOC)
            {
                vertexPick = lineCheck;
            }
            // we need to make sure there is space in the
appropriate buffer
            else
            {
                if (vertexStation[lineCheck].status.allocation ==
SQ_MEMORY)
                {
                    if
((vertexStation[lineCheck].status.allocationSize+1 <= sq->pSX_SQ-
>GetExportBuffer()/4)
                        && sq->pSX_SQ->GetValid())
```

- 91 -

```
                        {
                                vertexPick = lineCheck;
                        }
                }
                else if
(vertexStation[lineCheck].status.allocation == SQ_PARAMETER_PIXEL)
                {
                        // determine if there is space in the PCs
for an eventual PC export
                        pcSpace =
checkPC((vertexStation[lineCheck].status.allocationSize+1)*4);
                        if (pcSpace)
                        {
                                vertexPick = lineCheck;
                        }
                }
                else if
(vertexStation[lineCheck].status.allocation == SQ_POSITION
                        && sq->pSX_SQ->GetPositionReady() && sq-
>pSX_SQ->GetValid())
                {
                        // make sure every older threads have their
position allocated
                        bool alloc_done = true;
                        int alloc_line = vertexHead;
                        while (lineCheck != alloc_line)
                        {
                                if
(vertexStation[alloc_line].status.posAllocated == false)
                                {
                                        alloc_done = false;
                                        break;
                                }
                                alloc_line =
(alloc_line+1)%MAX_VTX_RESERVATION_SIZE;
                        }
                        if (alloc_done)
                        {
                                vertexPick = lineCheck;
                        }
                }
        }
        // make sure the status says we can pick this vertex
        if (vertexPick != -1)
        {
                // check for serial with texture pending
                if (vertexStation[vertexPick].status.serial &&

    vertexStation[vertexPick].status.texReadsOutstanding)
                        vertexPick = -1;
                // if last is set we can only pick the two
oldests threads
```

- 92 -

```
        else if (vertexStation[vertexPick].status.last &&
                    !(vertexPick==vertexHead ||
vertexPick==((vertexHead-1)%MAX_VTX_RESERVATION_SIZE)))
                    vertexPick = -1;
            // cannot pick last if texture reads are
outstanding
        else if (vertexStation[vertexPick].status.last &&

    vertexStation[vertexPick].status.texReadsOutstanding)
                    vertexPick = -1;
            // can only pick the second to old if the first
is already running
        else if (vertexStation[vertexPick].status.last &&

    !vertexStation[vertexHead].status.valid)
            {
                if (vertexStation[vertexPick].status.first)
                    vertexPick = -1;
                else
                {
                    predOn = false;
                    break;
                }
            }
            else
                break;
        }
    }// endif vertex

    lineCheck = (lineCheck+1)%MAX_VTX_RESERVATION_SIZE;
    }// end for loop
```

(arbiter.cpp, 56:11-60:1.)

192. By traversing the *vertexStation* entries after the *pixelStation* entries

and selecting a candidate vertex command thread based on different status

conditions, the R400 Emulator Code describes an arbiter operable to select a vertex

command thread from the plurality of vertex command threads based on the

relative priorities of the plurality of pixel command threads and the plurality of

vertex command threads.

- 93 -

193. Once the *Arbiter::chooseAluStation(...)* function chooses the pixel

command thread from the *pixelStation* and the vertex command thread from the

vertexStation, the *Arbiter::chooseAluStation(...)* function selects a command

thread from the selected pixel command thread and the selected vertex command

thread, with the vertex command thread having priority over the pixel command

thread. That is, if a vertex command thread exists, the vertex command thread is

selected and becomes the claimed command thread. Otherwise, the pixel command

thread is selected and becomes the claimed command thread. (arbiter.cpp, 60:3-

65:25.) The excerpts of code that demonstrate this are replicated below at

arbiter.cpp, 60:3-60-8 and 63:12-15, and the code in its entirety can be found in

arbiter.cpp, 60:3-65:25:

```
// right now vertices have priority over pixels always,
// will have to change this when the registers are there.
    if (vertexPick != -1)
    {
        lineNumber = vertexPick;
        sType = VERTEX;
...
...
...
    if (pixelPick != -1)
    {
        lineNumber = pixelPick;
        sType = PIXEL;
...
...
...
    }
```

194. Once *Arbiter::chooseAluStation(...)* selects a command thread, the

Arbiter::chooseAluStation(...) function returns a boolean variable which indicates

- 94 -

whether the command thread was selected. The *Arbiter::chooseAluStation(...)*

function also returns 1) a type of the selected command thread – a vertex command

thread or a pixel command thread, which is stored in variable *sType*, and 2) a

location of the command thread in the vertexStation or pixelStation arrays stored in

the variable *lineNumber*. (arbiter.cpp, 53:24-25.)

195.   If the *Arbiter::chooseAluStation(...)* function indicates that a

command thread was found, the *Arbiter::runAlu(...)* function uses the

Arbiter::popStationVector(...) function to select the command thread from either

vertexStation or *pixelStation*, as shown below:

```
int lineNumber;
Shader_Type stype;
ReservationStation* aluStationData;
found=chooseAluStation(lineNumber,stype,otherAluRunning,otherCFMachine
,predToggle);
if (found)
{
        aluRunning = true;
        popStationVector(aluStationData,lineNumber,stype);
    ...
    ...
    ...
    }
```

(arbiter.cpp, 17:7-18-2.)

196.   As I explain above, the *Arbiter::popStationVector(...)* function passes

a single *ReservationStation* structure (called *aluStationData*), the line number

location of the selected command thread in the *vertexStation* or *pixelStation* array

called *texLineNumber*, and the type (vertex or pixel) of the command thread called

- 95 -

stype. The *aluStationData* variable is of type *ReservationStation* and stores the

memory address location of the selected command thread.

197. The *Arbiter::popStationVector()* function removes the selected

command thread from either a vertexStation or a pixelStation entry (based on the

stype variable) and stores the removed command thread in the *ReservationStation*

structure called *aluStationData*.

198. In this way, the R400 Emulator Code describes another arbiter

operable to select a command thread from either of the plurality of pixel command

threads and the plurality of vertex command threads based on relative priorities of

the plurality of pixel command threads and the plurality of vertex command

threads.

2. Claim 2

a. The Preamble

199. Claim 2 recites the graphics-processing system of claim 1, further

comprising "*a command processing engine, coupled to the arbiter.*"

200. The sq_alu.h and sq_alu.cpp files define an ALU processing engine

and create an object of type *SQ_ALU*. The ALU processing engine is a command

processing engine.

- 96 -

201. This ALU processing engine object is a component of the

cUSER_BLOCK_SQ class identified above as the sequencer class. For example, in

the user_block_mode.h file, the *cUSER_BLOCK_SQ* class defines an ALU

processing engine using a static *sqAlu* object definition, as shown in the code

below:

```
// the ALU
SQ_ALU sqAlu;
```
                                    (user_block_model.h, Exhibit 2089,11:6-7.)

202. This means that when an object in the *cUSER_BLOCK_SQ* class is

created, the *SQ_ALU* object, such as, *sqAlu* is also created.

203. The *sqAlu* object defined in the *cUSER_BLOCK_SQ* class is a public

variable, which means that other objects, such as an arbiter, can access the ALU

processing engine. The arbiter in the *Arbiter::runAlu(…)* function, for example,

accesses the ALU processing engine defined in the *cUSER_BLOCK_SQ* object

with a call to the *sqAlu.Execute(…)* function, as illustrated below:

```
sq->sqAlu.Execute(sq->regFile[aluPhase],sq->outBuffer,  sq-
>constantStore[currentCFMachine.stationData->data.state],
srcAAddr,srcBAddr,  srcCAddr,srcCRegPtr,dstAddr,scalarDstAddr,  inst,
currentCFMachine.stationData->data.valids[aluPhase], aluPhase,sq-
>pSQ_SP, currentCFMachine.sType,&(currentCFMachine.stationData-
>data.predicates[aluPhase*16]), sq,AluID);
```
                                              (arbiter.cpp, 21:26-23:9.)

- 97 -

204. Since the ALU processing engine may be invoked by a function defined in the Arbiter class, the R400 Emulator Code shows that the ALU processing engine (which is a command processing engine) is coupled to the arbiter.

205. Below, I have provided a figure that represents how the command processing engine is coupled to the arbiter. The figure includes components as they are instantiated in the C++ source code and with the structures and/or classes that define the components:

```
                    ┌─────────────────────────────┐
                    │          Sequencer          │
                    │                             │
                    │                             │
                    │      class cUSER_BLOCK_SQ    │
                    │      (user_block_model.h)    │
                    └─────────────────────────────┘
                       /                        \
                      /                          \
    ┌────────────────────────┐      ┌────────────────────────────┐
    │        Arbiter         │      │    ALU Processing Engine    │
    │                        │      │                            │
    │        arbiter         │      │           sqALU            │
    │   (user_block_model.h) │      │    (user_block_model.h)    │
    │                        │      │                            │
    │     class Arbiter      │      │       class SQ_ALU         │
    │      (arbiter.h)       │      │        (sq_alu.h)          │
    └────────────────────────┘      └────────────────────────────┘
```

- 98 -
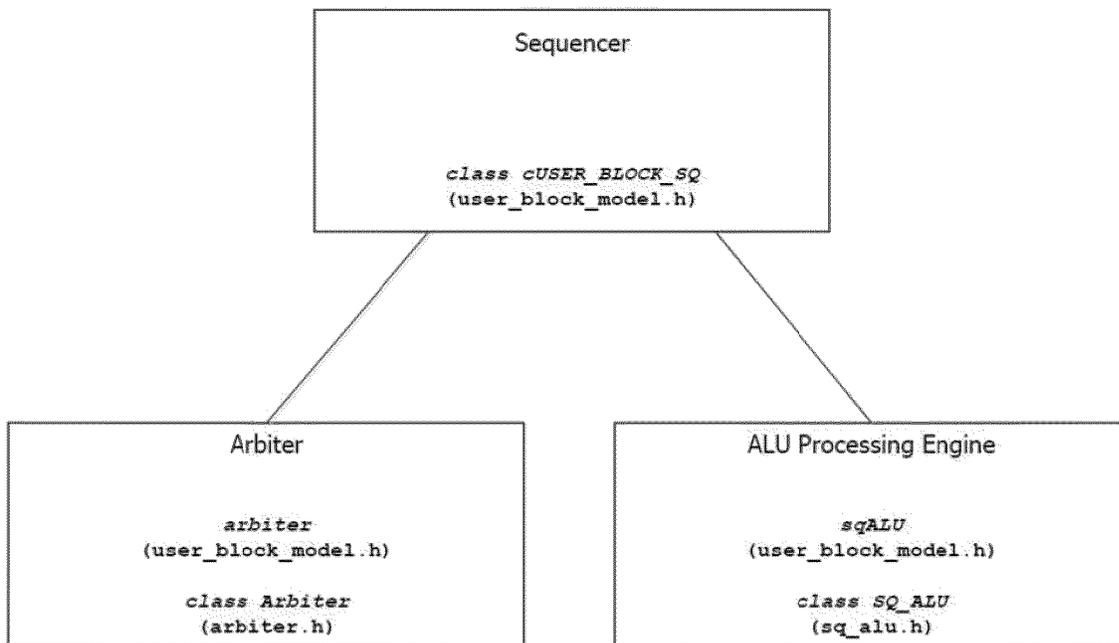
b. **The Arbiter is Operable to Provide a Command Thread to the Command Processing Engine**

206. Claim 2 also recites *"wherein the arbiter is further operable to provide the command thread to the command processing engine."*

207. When the arbiter uses the *Arbiter::popStationVector()* function to select a command thread that is accessed via *aluStationData,* the arbiter stores the command thread in a control flow machine object called currentCFMachine, as shown using the R400 Emulator Code below:

```
currentCFMachine.init(sq,aluStationData,stype,lineNumber,position,NULL
);
```

(arbiter.cpp, 17:18.)

208. The arbiter then sets the instruction address of the command thread from the command thread's context in *aluStationData* using the statement below:

```
stop = currentCFMachine.getNextInstruction(aluInstruction,nop,last);
```

(arbiter.cpp, 18:19.)

209. When the command processing engine is ready to execute the command thread's instruction, the arbiter retrieves the instruction from the instruction memory, using the statement below:

```
sq->instructionStore.GetInst(inst,aluInstruction);
```

(arbiter.cpp, 18:19.)

- 99 -

210. Next, the arbiter uses the GPRAddressCompute function to calculate the addresses in memory required for the command processing engine to process the command thread, using the statement below:

```
GPRAddressCompute(currentCFMachine.stationData-
>data.gprBase,inst,currentCFMachine,srcAAddr,
srcBAddr,srcCAddr,dstAddr,scalarDstAddr)
```

(arbiter.cpp, 21:22-24.)

211. Next, the arbiter invokes the ALU processing engine to process the command thread's instruction, using the statement below:

```
sq->sqAlu.Execute(sq->regFile[aluPhase],sq->outBuffer,   sq-
>constantStore[currentCFMachine.stationData->data.state],
srcAAddr,srcBAddr, srcCAddr,dstAddr, scalarDstAddr, inst,
currentCFMachine.stationData->data.valids[aluPhase], aluPhase,sq-
>pSQ_SP, currentCFMachine.sType,&(currentCFMachine.stationData-
>data.predicates[aluPhase*16][0]), sq);
```

(arbiter.cpp, 21:26-22:9.)

212. The *SQ_ALU::Execute(...)* function is included in the *SQ_ALU* class and, when invoked by the *sqAlu* object by way of the arbiter, processes the command thread's instruction. (sq_alu.cpp, 2:9-7:24.) In particular, the *SQ_ALU::Execute(...)* invokes the *SQ_ALU::ExecuteAluInstruction(...)* function at page 7, line 20 (7:20) which executes the command thread's instruction.

213. As such, the R400 Emulator Code recites an arbiter operable to provide the command thread to the command processing engine.

- 100 -

3. Claim 5

214. The preamble of claim 5 recites "*A graphics processing system,*" discussed above in my analysis of claim 1 in Section VII.B. 1. This is present in the R400 Emulation Code for the same reasons as explained previously.

215. The first limitation of claim 5 recites "*at least one memory device comprising a first portion operative to store a plurality of pixel command threads and a second portion operative to store a plurality of vertex command threads,*" discussed above in my analysis of claim 1 in Section VII.B.1. This is present in the R400 Emulation Code for the same reasons as explained previously.

216. The second limitation of claim 5 recites "*an arbiter, coupled to the at least one memory device,*" which I discussed above in my analysis of claim 1 in Section VII.B.1. This is present in the R400 Emulation Code for the same reasons as explained previously.

217. The arbiter of claim 5 is "*operable to select a command thread from either of the plurality of pixel command threads and the plurality of vertex command threads.*" The prior discussion explains my analysis of how the arbiter selects a command thread in claim 1 in Section VII.B.1. This is present in the R400 Emulation Code for the same reasons as explained previously.

218.   The third limitation of claim 5 recites "*a plurality of command processing engines, coupled to the arbiter, each operable to receive and process the command thread*." As discussed in my analysis for claim 2 in Section VII.B.2, the R400 Emulator Code describes the ALU processing engine which is a command processing engine. The ALU processing engine exists in the *sqAlu* object which receives the command thread's instruction(s). Also, the R400 Emulator Code describes a texture processing engine which is a command processing engine.

219.   I have also provided a figure that represents the coupling between the arbiter and the ALU processing engine and the texture processing engine. The figure includes components as they are instantiated in the R400 Emulator Code and also the structures and/or classes that define the components:

- 102 -
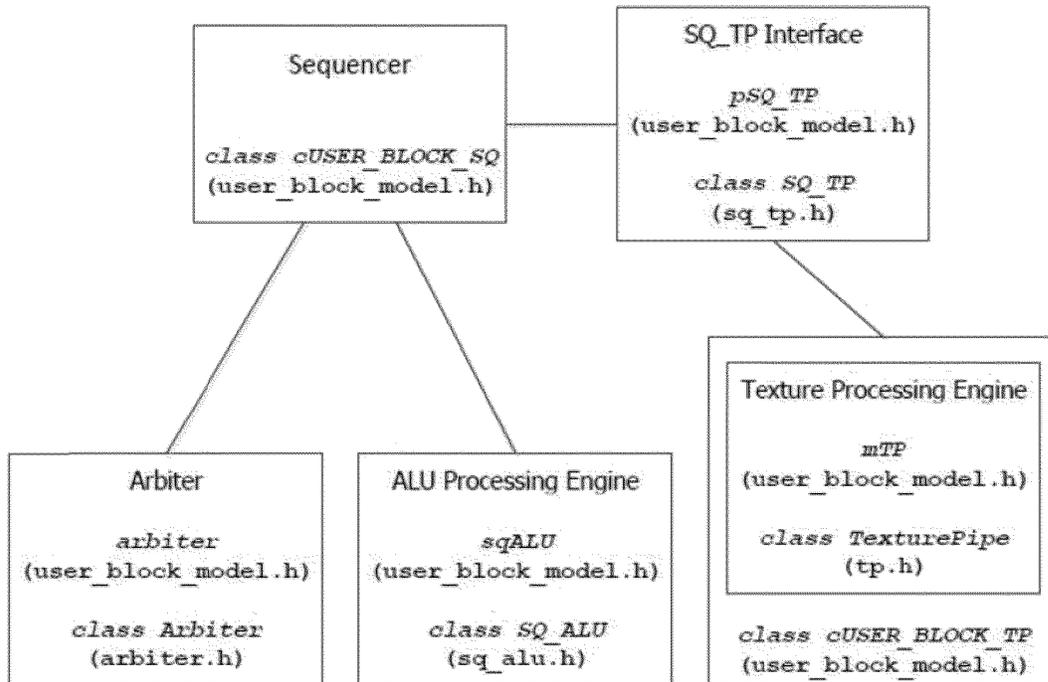
220. With respect to the ALU processing engine, the *sqAlu* object is of type *SQ_ALU*. The *sqAlu* object receives and processes the command thread's instruction(s) using two functions, the *SQ_ALU::Execute(...)* function (sq_alu.cpp, 2:9-7:24) and the *SQ_ALU::ExecuteAluInstruction(...)* function (sq_alu.cpp, 8:1-36:1.) Each of the functions process the command thread's instruction. The *SQ_ALU::Execute(...)* calls the *SQ_ALU::ExecuteAluInstruction(...)* function as replicated below:

```
switch(VectorIndex)
     {
     case 0:
          SPData.Instruction = Instruction.SrcASel +
(Instruction.SourceANegate << 2) +
                              (Instruction.SourceASwizzle <<
4) +
```

- 103 -

```
((Instruction.VectorResultPointer&0x3F)<<12);
        break;
    case 1:
        SPData.Instruction = Instruction.SrcBSel +
(Instruction.SourceBNegate << 2) +
                                    (Instruction.SourceBSwizzle <<
4) +

((Instruction.ScalarResultPointer&0x3F)<<12);
        break;
    case 2:
        SPData.Instruction = Instruction.SrcCSel +
(Instruction.SourceCNegate << 2) +
                                    (Instruction.SourceCSwizzle <<
4);
        break;
    case 3:
        SPData.Instruction = Instruction.VectorOpcode +
(Instruction.ScalarOpcode << 5)+
                                    (Instruction.VectorClamp << 11)
+ (Instruction.ScalarClamp << 12)+
                                    (Instruction.VectorWriteMask <<
13) + (Instruction.ScalarWriteMask << 17);
        break;
    }

    // do all the static stuff for next turn
    if (Instruction.SrcASel)
        Constants.GetConstValue(constant[VectorIndex],SrcAAddr);
    else if (Instruction.SrcBSel)
        Constants.GetConstValue(constant[VectorIndex],SrcBAddr);
    else if (Instruction.SrcCSel)
        Constants.GetConstValue(constant[VectorIndex],SrcCAddr);

    for (i=0;i<4;i++)
        PMasks[VectorIndex][i] = valids[i];

    switch(VectorIndex)
    {
    case 0: // interpolator and SRC A
        CMask[VectorIndex] = 127-SrcAAddr;
        RAddr[VectorIndex] = SrcAAddr;
        WAddr[VectorIndex] = 126-SrcAAddr;
        REn[VectorIndex] = true;
        WEn[VectorIndex] = false;
        break;
    case 1: //TX and SRC B
      CMask[VectorIndex] = 125-SrcBAddr;
        RAddr[VectorIndex] = SrcBAddr;
        WAddr[VectorIndex] = 124-SrcBAddr;
        REn[VectorIndex] = true;
```

- 104 -

```
            WEn[VectorIndex] = false;
            break;
    case 2: // Vector and SRC C
            CMask[VectorIndex] = Instruction.VectorWriteMask;
            RAddr[VectorIndex] = SrcCAddr;
            REn[VectorIndex] = false; // no tree operands for now
            // if exporting
        if (((Instruction.VectorResultPointer & 0x80) != 0) &&
(Instruction.PredicateSelect < 2)) {
            WAddr[VectorIndex] = Instruction.VectorResultPointer & 0x3F;
            WEn[VectorIndex] = false;
        }
        else {
            WAddr[VectorIndex] = DestAddr;
            WEn[VectorIndex] = true;
        }
            break;
    case 3: // Scalar and TX
            CMask[VectorIndex] = Instruction.ScalarWriteMask;
            RAddr[VectorIndex] = 123-ScalarDestAddr;
            REn[VectorIndex] = false;
            // if exporting
        if (((Instruction.ScalarResultPointer & 0x80) != 0) &&
(Instruction.PredicateSelect < 2)) {
            WAddr[VectorIndex] = Instruction.ScalarResultPointer & 0x3F;
            WEn[VectorIndex] = false;
        }
            /*
        else {
            WAddr[VectorIndex] = ScalarDestAddr;
            WEn[VectorIndex] = true;
        }*/ // No scalar ops for now...
            break;
    }

    pSQ_SP->SetAll(&SPData);
    pSQ_SP->SetValid(true);


    // Real Emulator code
    CurrentRegFile = Reg;
    OutputBuffer = &ExportBuffer;
    CurrentAluInstruction = Instruction;
    AluPhase = VectorIndex;
    AluType = currentAluType;
    Predicates = &(pred[0]);
    validBits= &(valids[0]);
    ExecuteAluInstruction(SrcAAddr,SrcBAddr,SrcCAddr,DestAddr,ScalarD
estAddr,VectorIndex,Constants);
```

(sq_alu.cpp,4:2-6:21.)

221. The Emulator Code also includes a texture processing engine defined in the class called *TexturePipe*. (tp.h, tp.cpp.) The *TexturePipe* generates a TexturePipe object called *mTP*, which is a texture processing engine. (user_block_model.h, Exhibit 2104, 3:12.) The class which instantiates an *mTP* object is called *cUSER_BLOCK_TP*, and is defined in user_block_model.h. The *cUSER_BLOCK_TP* also instantiates an interface between the arbiter (described above) and the *mTP*, the texture processing engine, called *mSQ_TP* of type *SQ_TP*. (user_block_model.h, Exhibit 2104, 3:11.) The *SQ_TP* class is defined in sq_tp.h.

222. As described above for claims 1 and 2 in Sections VII.B.1 and VII.B.2, when the arbiter selects a command thread for the texture processing engine in the *Arbiter::Execute()* function, the arbiter stores the address of the selected command thread in *texStationData*. (arbiter.cpp, 11:15.) The arbiter then stores the command thread in a control flow machine object called *textureCFMachine*, as shown below:

```
textureCFMachine.init(sq,texStationData,texType,texLineNumber,NULL);
```

(arbiter.cpp, 11:18.)

223. When the command processing engine is ready to execute the command thread's instruction(s), the arbiter retrieves the instruction(s) from the instruction memory and stores the location of the texture instruction in textureInstruction, using the statement below:

- 106 -

```
stop =
textureCFMachine.getNextInstruction(textureInstruction,nop,last);
```

(arbiter.cpp, 11:16.)

224.   Next, the arbiter invokes an *Arbiter::fillTextureInterface(...)* function using the statement below:

```
fillTextureInterface(textureInstruction,texturePhase,stop);
```

(arbiter.cpp, 13:22.)

225.   The *Arbiter::fillTextureInterface(...)* function retrieves the command thread's instruction(s) from the instruction store using textureInstruction and stores the command thread's instruction(s) in the *TInstrPacked* object called instr, in the statement below:

```
sq->instructionStore.GetInst(inst,textureInstAddr);
```

(arbiter.cpp, 41:5.)

226.   Next, *Arbiter::fillTextureInterface(...)* provides the instruction to the arbiter-texture pipeline interface object *pSQ_TP*, using the statement below:

```
sq->pSQ_TP->SetSQ_TP_instr(inst);
```

(arbiter.cpp, 45:1.)

- 107 -

227. Here, *pSQ_TP* is an interface that transfers the instruction(s) between the sequencer and the texture processing engine. The object *pSQ_TP* is defined in sq_tp.h

228. On the texture processing engine side, when the *cUSER_BLOCK_TP* object executes the *cUSER_BLOCK_TP::Main()* function, the *cUSER_BLOCK_TP::Main()* fetches, processes and outputs the command thread's instruction set with the *Arbiter::fillTextureInterface()* function. For example, *cUSER_BLOCK_TP::Main()* includes three functions: *Fetch()*, *Process()*, and *Output()*. The *cUSER_BLOCK_TP::Fetch()* function retrieves the instruction, using the statement below:

```
void cUSER_BLOCK_TP::Fetch(void)
{
    ...
    //Copy the interface data
    mSQ_TP->GetAll( &mSQ_TP_data );
}
```

(tp_block_model.cpp, 2:23-3:6.)

229. Once the *cUSER_BLOCK_TP* object retrieves the command thread's instruction(s) as *mSQ_TP_data*, the *cUSER_BLOCK_TP::Process()* function causes the texture processing engine, called *mTP*, to process the command thread's instruction(s) using the statement below:

```
void cUSER_BLOCK_TP:: Process(void)
{
...
mTP->process(mSQ_TP_data, mTP_SQ_data);
```

- 108 -

...
}

(tp_block_model.cpp, 3:8-3:26.)

230. The texture processing engine then processes the command thread's

instruction(s) using the *TexturePipe::process(...)* function. (tp.cpp, 6:3-8:11.) In

particular, the *TexturePipe::process(...)* function stores the command thread's

instruction using *TexturePipe::init(...)* function (tp.cpp, 6:7) and invokes the

TexturePipe:Run() function which processes the command thread's instruction(s),

as replicated below:

```
void
TexturePipe::run
( void )
{
    switch( mInstrPacked->getOPCODE() )
    {
    case TInstr::Opcode_FetchVertex:
        VF_DoSubVector();
        break;
    case TInstr::Opcode_FetchTextureMap:
        TF_DoSubVector();
        break;
    default:
        cerr << "Unsupported OPCODE: " << mInstrPacked->getOPCODE() <<
endl;
        break;
    }
}
```

(tp.cpp, 11:2-11:18.)

231. The ALU processing engine and the texture processing engine

included in the R400 Emulator Code are a plurality of command processing

engines coupled to the arbiter. As described above, each of the ALU processing

- 109 -

engine and the texture processing engine is operable to receive and process the

command thread.

4. Claim 6

232.   Claim 6 recites the *"graphics processing system of claim 5, wherein the plurality of command processing engines comprises at least one arithmetic logic unit."* In my analysis of claim 5 in Section VII.B.3, I have explained that the processing engine comprises at least one ALU unit.

5. Claim 7

233.   Claim 7 recites the *"graphics processing system of claim 5, wherein the plurality of command processing engines comprises at least one texture processing engine."* In my analysis of claim 5 in Section VII.B.3, I have explained that the processing engine comprises at least one texture processing engine.

VIII. The Claims of the '053 Patent Are Supported by the Priority Document

234.   I understand that a specification must contain a 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.

235.   I understand that a priority document to which a patent claims

priority, can be used to satisfy the written description requirement.

236.   I have examined the specification and figures of the '761 Application

– the priority document. The '761 Application was filed on September 29, 2003. I

understand the '053 patent claims priority to the '761 Application, because, U.S.

Patent Application No. 11/764,453 from which the '053 patent issued, is a

continuation of the '761 Application.

237.   Based on my examination of the '761 Application, I have generated a

claim chart which demonstrates that claims 1, 2, 5, 6, and 7 are supported by the

'761 Application.

Support for the '053 Patent Claims in U.S. Patent Application No. 10/673,761	
'053 Patent Claim	
1. A graphics processing system comprising	"The present invention relates generally to graphics processing." (Ex. 2008, ¶ 1.) "Generally, the present invention includes a multi-thread graphics processing system." (*Id.* at ¶ 14.) "[T]he present invention allows for multi-thread command processing effectively using designated reservation station,

- 111 -

	in conjunction with the arbiter, for the improved processing of multiple command threads. The present invention further provides for the effective utilization of the ALU and the graphics processing engine, such as the texture engine, for performing operations for both pixel command threads and vertex command threads, thereby improving graphics rendering and improving command thread processing flexibility." (*Id.* at ¶ 38.) "The present invention includes a multi-thread graphics processing system." (*Id.* at Abstract.)
1a. at least one memory device comprising a first portion operative to store a plurality of pixel command threads and a second portion operative to store a plurality of	"Generally, the present invention includes a multi-thread graphics processing system and method thereof including a reservation station having a plurality of command threads stored therein. A reservation station may be any type of memory device capable of reserving and storing command threads. Furthermore, a command thread is a sequence of commands applicable to the corresponding element, such as pixel command thread relative to processing of pixel elements and a vertex command thread relative to vertex

- 112 -

| vertex command threads; and | processing commands." (*Id.* at ¶ 14.)

"FIG. 2 illustrates a schematic block diagram of a multi-thread processing system, in accordance with one embodiment of the present invention." (*Id.* at ¶ 8.)

(*Id.* at FIG. 2.)
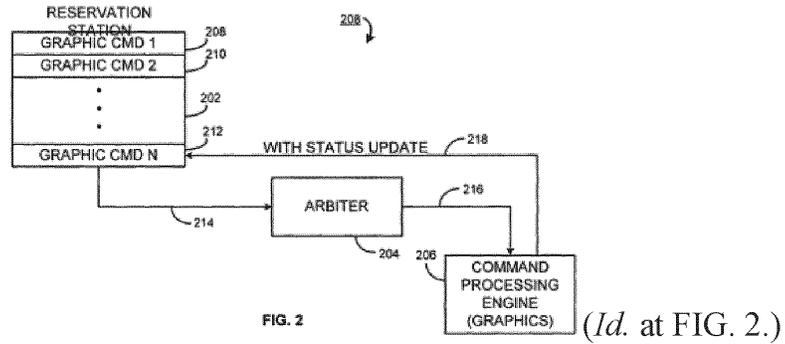
"The system 200 includes a reservation station 202." (*Id.* at ¶16.) "The reservation station includes a plurality of command threads 208, 210 and 212." (*Id.*) "In one embodiment, the command threads 208-212 are graphic command threads" (*Id.*)

"FIG. 4 illustrates a schematic block diagram of a multi-thread command processing system in accordance with one embodiment." (*Id.* at ¶ 10.) |

- 113 -
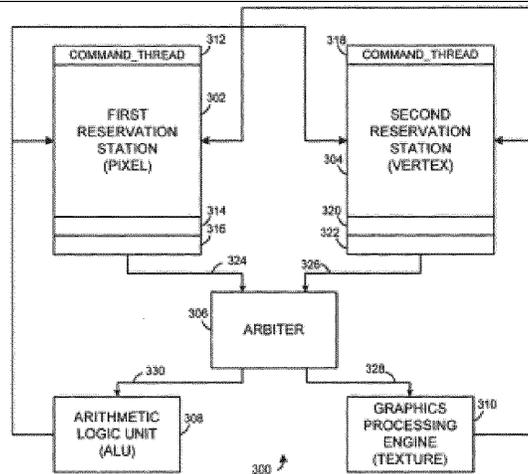
FIG. 4

(*Id.* at FIG. 4.)

"FIG. 4 illustrates another embodiment of a multi-thread command processing system 300 having a first reservation station 304 [and] a second reservation station 304." (*Id.* at ¶ 21.) "In this embodiment, 302 is a pixel reservation station such that the command threads 312, 314 and 316 contain pixel-based commands therein. Furthermore, in this embodiment the second reservation 304 a vertex reservation station is directed towards vertex command threads illustrated as command threads 318, 320 and 322." (*Id.* at ¶ 21.)

"One embodiment, each command thread within the reservation station 302 and 304 may be stored across two

- 114 -

physical pieces of memory." (*Id.* at ¶ 26.)

"FIG. 6 illustrates a flow chart for a method of multi-thread command processing in accordance with one embodiment." (*Id.* at ¶ 31.) "The method begins, step 400, by retrieving a selected command thread from a plurality of first command threads and a plurality of second command threads, step 402." (*Id.*)
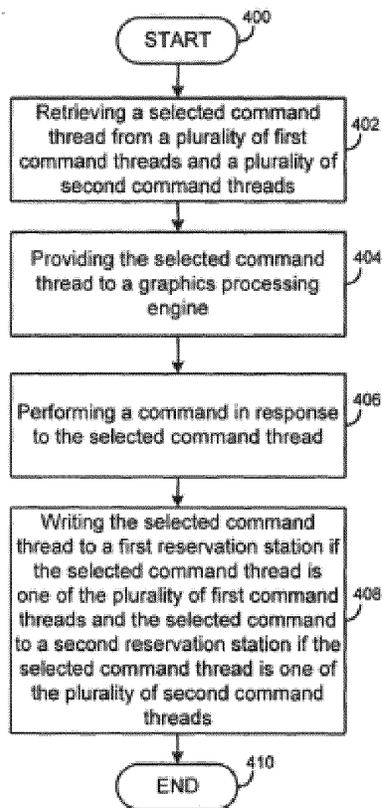
FIG. 6   (*Id.* at FIG. 6.)

- 115 -

|	"FIG. 7 illustrates a flowchart of an alternative method for multi-thread processing. "The second selected command thread may be retrieved from either a first reservation station, such as reservation station 302 of FIG. 4 or a second reservation station, such as reservation station 304 of FIG. 4." (*Id.* at ¶ 34.)
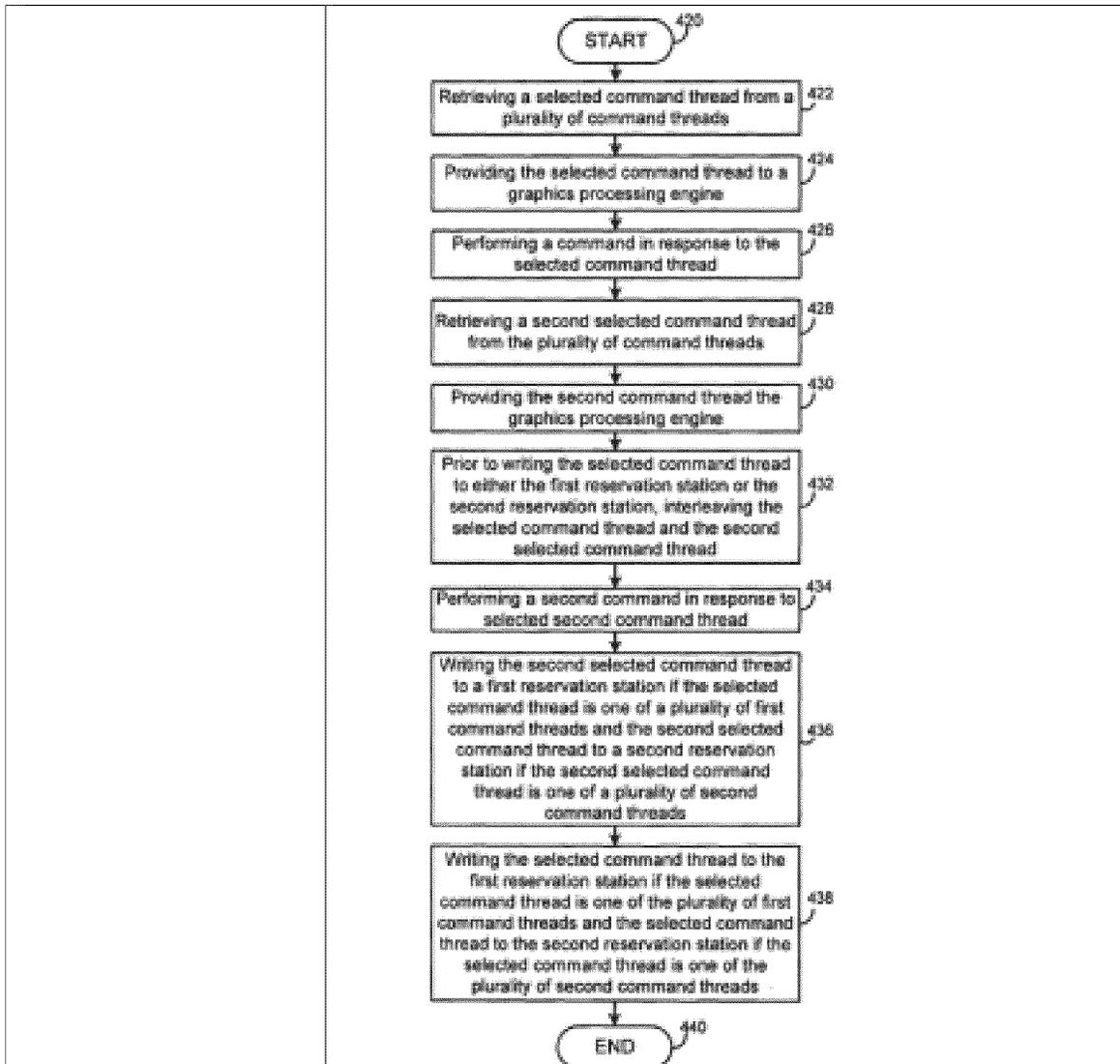
(Id. at FIG. 7.)

"The present invention includes . . . a reservation station having a plurality of command threads stored therein." *Id.* at Abstract.) |
| 1b. an arbiter, | "To improve the operating efficiency of a graphics |

- 117 -

| coupled to the at least one memory device, operable to select a command thread from either of the plurality of pixel command threads and the plurality of vertex command threads based on relative priorities of the plurality of pixel command threads and the plurality of vertex command threads. | processing system, the control of the flow of the multiple command threads is preferred." (*Id.* at ¶ 2.)

"The system and method further includes an arbiter operably coupled to the reservation station such that the arbiter retrieves a first command thread of the plurality of command threads stored therein. The arbiter may be any implementation of hardware, software or combination thereof such that the arbiter receives the command thread." (*Id.* at ¶ 14.)

"FIG. 2 illustrates a schematic block diagram of a multi-thread processing system, in accordance with one embodiment of the present invention." (*Id.* at ¶ 8.)

(*Id.* at FIG. 2.)
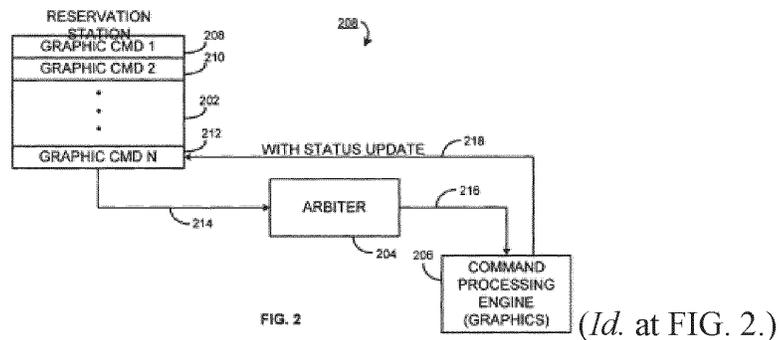
"The system 200 includes . . . an arbiter 204." (*Id.* at 16.) |

"The arbiter 204 retrieves a command thread via

connection 214." (*Id.*)

"In one embodiment, the arbiter 204 retrieves the

command threads 208-212 based on a priority scheme. For

example, the priority may be based on specific commands

that have been executed within a command thread or

specific commands which are to be executed within a

command for the effective utilization of the arbiter 204

and the command processing engine 206. In an alternative

embodiment, the arbiter 204 may always retrieve the

oldest available thread." (*Id.* at ¶ 18.) "[I]n one

embodiment the reservation station 202 operates similar to

a first in first out (FIFO) memory device." (*Id.* at ¶ 16.)

"The ALU arbitration logic chooses one of the pending

ALU clauses to be executed. The arbiter selects the

command thread by looking at the reservation stations,

herein vertex and pixel reservation stations, and picking

the first command thread ready to execute." (*Id.* at ¶ 20.)

- 119 -

"FIG. 4 illustrates a schematic block diagram of a multi-

thread command processing system in accordance with

one embodiment." (*Id.* at ¶ 10.)
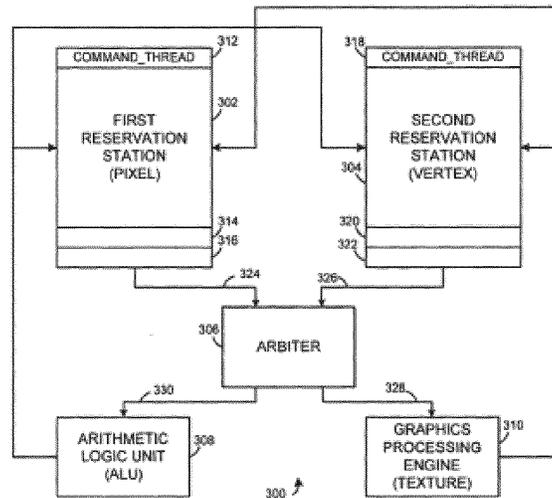
(*Id.* at FIG. 4.)

"FIG. 4 illustrates another embodiment of a multi-thread

command processing system 300 having . . . an arbiter

306." (*Id.* at ¶ 21.) "In this embodiment, the arbiter 306

selectively retrieves either a pixel command thread, such

as command thread 316, or a vertex command thread, such

as command thread 322." (*Id.* at ¶ 22.) "[T]he arbiter 306,

which may be implemented as arbitration logic executed

on a processing device, selects one thread for the graphics

- 120 -

processing engine 310 and one thread for the ALU 308."

(*Id.* at ¶ 23.)

"With respect to FIG. 4, a pixel command thread 324 may

be retrieved by the arbiter 306 and a vertex command

thread 326 may also be retrieved." (*Id.* at 24.)

"[T]he arbiter 306 selects the proper allocation of which

command thread goes to the graphics processing agent 310

in [sic] which command thread goes to the ALU 308." (*Id.*

at ¶ 29.)

"FIG. 6 illustrates a flow chart for a method of multi-

thread command processing in accordance with one

embodiment." (*Id.* at ¶ 31.) "The method begins, step 400,

by retrieving a selected command thread from a plurality

of first command threads and a plurality of second

command threads, step 402. For example, as discussed

above with regard to FIG. 4, the selected command thread

may be retrieved by the arbiter 306." (*Id.*)
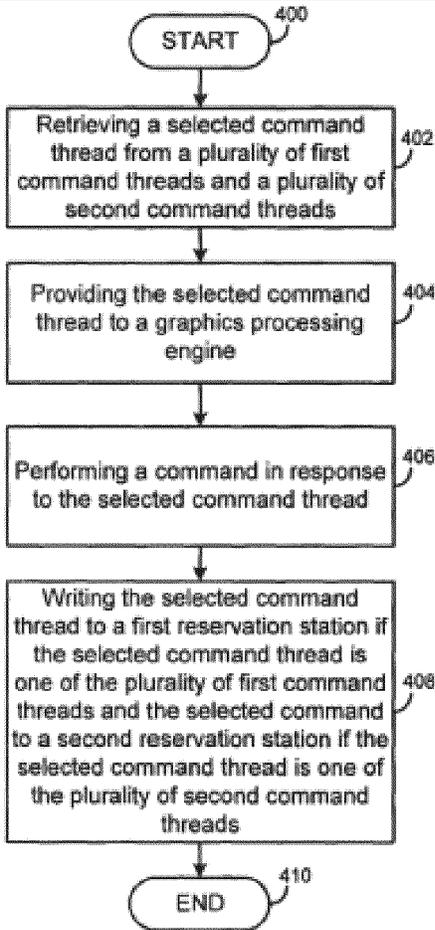
- 121 -

FIG. 6

(*Id.* at FIG. 6.)

"FIG. 7 illustrates a flowchart of an alternative method for multi-thread processing. The method begins, step 420, by retrieving a selected command thread from a plurality of command threads, step 422." (*Id.* at ¶ 33.) "[S]tep 428, is retrieving a second command thread from the plurality of command threads." (*Id.* at ¶ 34.) "The second selected

- 122 -

command thread may be retrieved from either a first

reservation station, such as reservation station 302 of FIG.

4 or a second reservation station, such as reservation

station 304 of FIG. 4." (*Id.*)

FIG. 7

(*Id.* at FIG. 7.)

- 123 -

	"The system . . . includes an arbiter operably coupled to the reservation station, such that the arbiter retrieves a first command thread." (*Id.* at Abstract.)
2. The graphics processing system of claim 1, further comprising: a command processing engine, coupled to the arbiter, wherein the arbiter is further operable to provide the command thread to the command processing engine.	"[T]he arbiter receives the command thread and thereupon provides the command thread to a command processing engine. The system and method further includes the command processing engine coupled to receive the first command thread from the arbiter such that the command processor may perform at least one processing command from the command thread. Whereupon, a command processing engine provides the first command thread back to the associated reservation station." (*Id.* at ¶ 14.) "The command processing engine may be any suitable engine as recognized by one having ordinary skill in the art for processing commands, such as a texture engine, an arithmetic logic unit, or any other suitable processing engine." (*Id.* at ¶ 15.) "FIG. 2 illustrates a schematic block diagram of a multi-

- 124 -

thread processing system, in accordance with one

embodiment of the present invention." (*Id.* at ¶ 8.)

FIG. 2 (*Id.* at FIG. 2.)
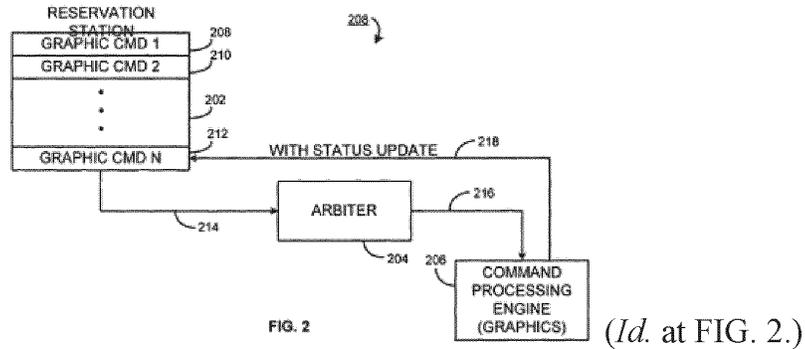
"The system 200 includes . . . a command processing

engine 206." (*Id.* at ¶ 16.) "The arbiter 204 . . . provides

the retrieved command thread to the command processing

engine 206, such as a graphics processing engine via

connection 216." (*Id.*)

"FIG. 4 illustrates a schematic block diagram of a multi-

thread command processing system in accordance with

one embodiment." (*Id.* at ¶ 10.)

- 125 -
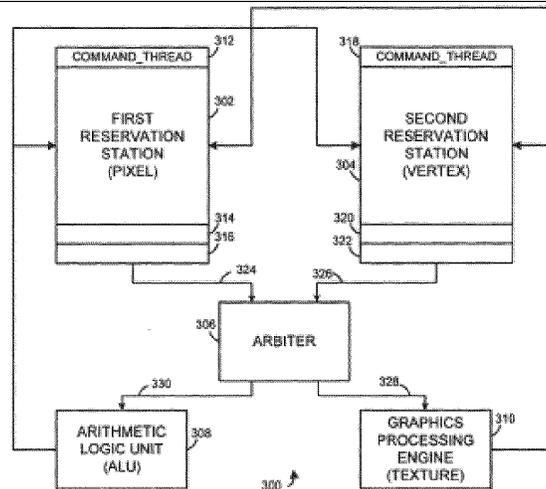
FIG. 4

(*Id.* at FIG. 4.)

"FIG. 4 illustrates another embodiment of a multi-thread command processing system 300 having . . . an ALU 308 and a graphics processing engine 310." (*Id.* at ¶ 21.)

"Once a thread is selected by the arbiter 306, the thread is . . . submitted to the appropriate execution unit 308 or 312." (*Id.* at ¶ 23.)

"The arbiter 306 then provides one thread 326, which may be either 324 or 326 to the graphics processing engine 310, such as a texture engine, and provides the other thread 330 to the ALU 308." (*Id.* at ¶ 24.)

"FIG. 6 illustrates a flow chart for a method of multi-

- 126 -

thread command processing in accordance with one

embodiment." (*Id.* at 31.) "[S]tep 404, is providing the

selected command thread to a graphics command

processing engine. As discussed above regarding FIG. 4,

the arbiter 306 provides the selected command thread to

the graphics processing engine 310, which, in one

embodiment may be a texture engine. In another

embodiment, the arbiter 306 may provide the selected

command thread to the ALU 308." (*Id.*)
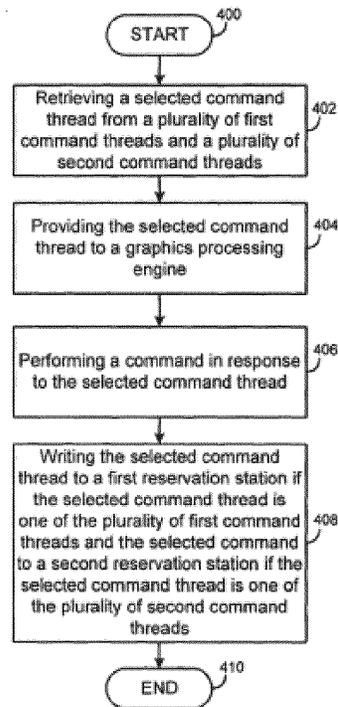

FIG. 6 (*Id.* at FIG. 6.)

- 127 -

FIG. 7 illustrates a flowchart of an alternative method for multi-thread processing." (*Id.* at ¶ 33.) "[S]tep 424, is providing the selected command thread to a graphics processing engine." (*Id.*) "The method further includes providing the second command thread to the graphics processing engine, step 430." (*Id.* at ¶ 35.)

FIG. 7

(*Id.* at FIG. 7.)

- 128 -

|	
	"[T]he arbiter retrieves the command thread and thereupon provides the command thread to a command processing engine." (*Id.* at Abstract.) "The system . . . includes the command processing engine coupled to receive the first command thread from the arbiter." (*Id.* at Abstract.)
5a. at least one memory device comprising a first portion operative to store a plurality of pixel command threads and a second portion operative to store a plurality of vertex command threads;	*See* Claim 1a (showing support for the same claim language).
5b. an arbiter,	*See* Claim 1b (showing support for the same claim

- 129 -

coupled to the at least one memory device, operable to select a command thread from either of the plurality of pixel command threads and the plurality of vertex command threads; and	language).
5c. a plurality of command processing engines, coupled to the arbiter, each operable to receive and process the command thread.	"The system and method further includes the command processing engine coupled to receive the first command thread from the arbiter such that the command processor may perform at least one processing command from the command thread. Whereupon, a command processing engine provides the first command thread back to the associated reservation station." (*Id.* at ¶ 14.) "The command processing engine may be any suitable engine as recognized by one having ordinary skill in the art

- 130 -

for processing commands, such as a texture engine, an

arithmetic logic unit, or any other suitable processing

engine." (*Id.* at ¶ 15.)

"[In the FIG. 2 embodiment,] the command thread may be

provided to a further processing element (not illustrated)

within a graphics processing pipeline." (*Id.* at ¶ 17.)

"FIG. 4 illustrates a schematic block diagram of a multi-

thread command processing system in accordance with

one embodiment." (*Id.* at ¶ 10.)
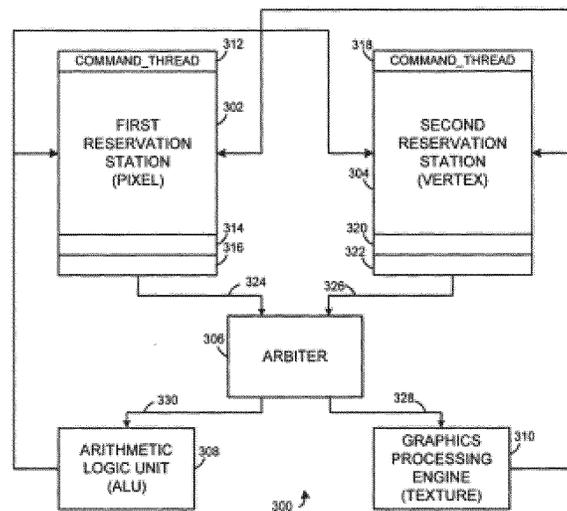
FIG. 4   (*Id.* at FIG. 4.)

"FIG. 4 illustrates another embodiment of a multi-thread

command processing system 300 having . . . an ALU 308

and a graphics processing engine 310." (*Id.* at ¶ 21.)

"Once a thread is selected by the arbiter 306, the thread is .

. . submitted to the appropriate execution unit 308 or 312."

(*Id.* at ¶ 23.) "Upon the execution of the associated

command of the command thread, the thread is thereupon

returned to the station 302 or 304 at the same storage

location with its status updated, once all possible

sequential instructions have been executed." (*Id.*)

"Upon execution of the command, the ALU 308 then

returns the command thread." (*Id.* at ¶ 25.)

"[T]he graphics processing engine 310 performs the

commands." (*Id.* at ¶ 24.)

"FIG. 6 illustrates a flow chart for a method of multi-

thread command processing in accordance with one

embodiment of the present invention." (*Id.* at ¶ 31.) "The

method . . . includes performing a command in response to

the selected command thread, step 406." "In this

- 132 -

embodiment the command is performed by the graphics

processing engine 310." (*Id.* at ¶ 32.)
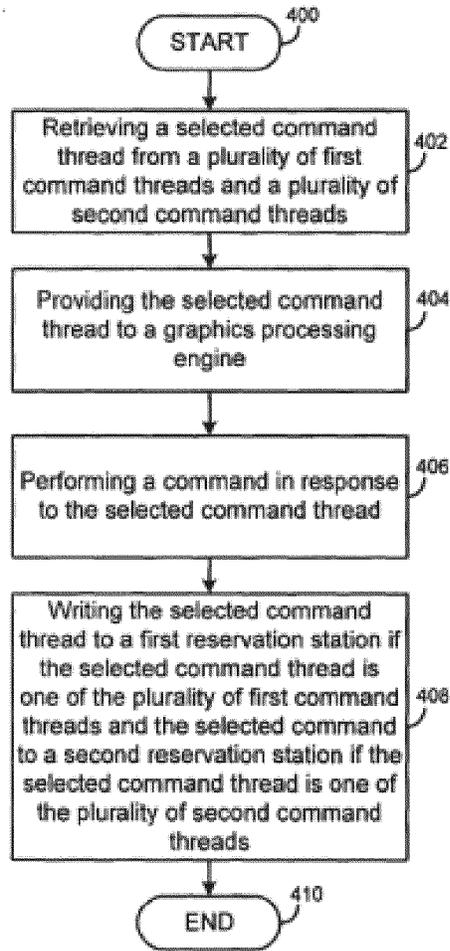
FIG. 6

(*Id.* at FIG. 6.)

"FIG. 7 illustrates a flowchart of an alternative method for

multi-thread processing." (*Id.* at ¶ 33.) "[T]he method . . .

includes performing a command in response to the selected

- 133 -

command thread, step 426." (*Id.* at ¶ 34.) "[T]he method further includes performing a second command in response to the selected command thread, step 434." (*Id.* at ¶ 35.)
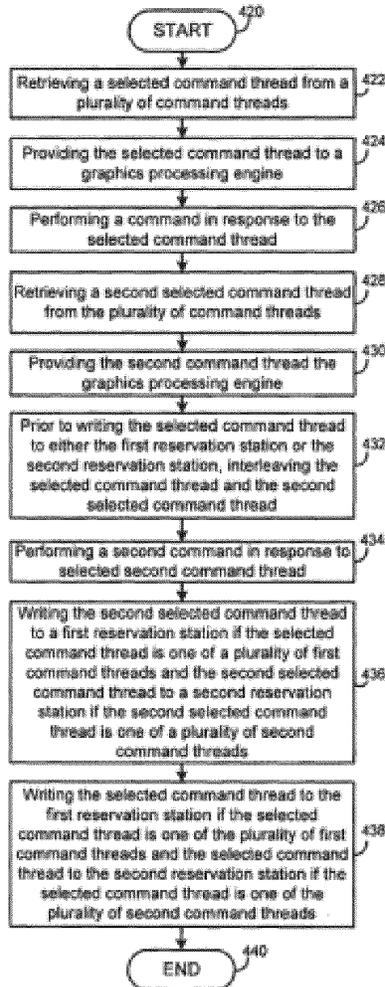
FIG. 7

(*Id.* at FIG. 7.)

"The system . . . includes the command processing engine

- 134 -

	coupled to receive the first command thread from the arbiter such that the command processor may perform at least one processing command from the command thread." (*Id.* at Abstract.)
6. The graphics processing system of claim 5, wherein the plurality of command processing engines comprises at least one arithmetic logic unit.	*See* Claim 5c (showing that the command processing engines can comprise an arithmetic logic unit ("ALU")).
7. The graphics processing system of claim 5, wherein the plurality of command processing engines comprises at least one texture processing	*See* Claim 5c (showing that the command processing engines can comprise a texture engine).

- 135 -

engine.	

IX.   CONCEPTION

238.   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.

239.   I have reviewed a document titled "R400 Top Level specification" (Ex. 2041) and a document titled "Shader Processor" (Ex. 2042). I have also reviewed both an August 24, 2001 revision and an April 19, 2002 revision of a document titled "R400 Sequencer Specification" (Exs. 2010, 2042). Both revisions of the R400 Sequencer Specification, especially when read in view of the R400 Top Level Specification and the Shader Processor specification, show possession of a complete embodiment of the claimed subject matter. Although the R400 Top Level Specification and the Shader Processor specification provide context, each and every claim element are shown in the R400 Sequencer Specifications. Further, the specification documents provide sufficient detail to enable the POSA to reduce

- 136 -

the concept to practice. Reducing the concept to practice could require substantial

work, but would not require undue experimentation.

240.   The following claim charts show that the inventors conceived of the

claimed subject matter at least by both revisions of the R400 Sequencer

Specification.

R400 SEQUENCER SPEC VERSION 0.4	
Claim	
1. A graphics processing system comprising	The R400 Sequencer Specification is an architectural specification for the R400's sequencer block. Ex. 2010, p. 1. 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. at least one memory device comprising a first portion operative to store a plurality	**At Least One Memory Device** The R400 Sequencer Specification describes reservation stations and an instruction store, which collectively are the claimed "at least one memory device." Two sets of the sequencer control flow, reproduced below,

of pixel command threads and a second portion operative to store a plurality of vertex command threads; and	with the reservation stations outlined in red, show sixteen reservation stations for vertices and sixteen reservation stations for pixels. *See* Ex. 2010, p. 5 (showing a figure and stating that "[t]here are two sets of the . . . figure, one for vertices and one for pixels"). Each set of reservation stations store eight ALU clauses and eight texture clauses. *See id.*, p. 4 ("The sequencer looks at all eight alu reservation stations to choose an alu clause to execute and all eight texture stations to choose a texture clause to execute."), 5 (reservation stations include clauses).
	Pixel Reservation Stations Vertex Reservation Stations
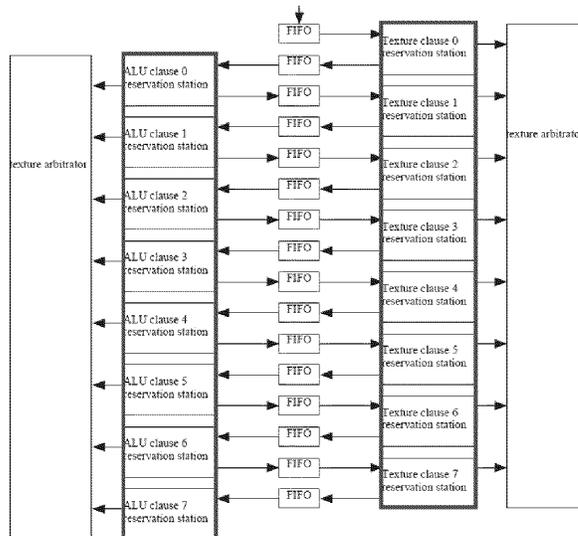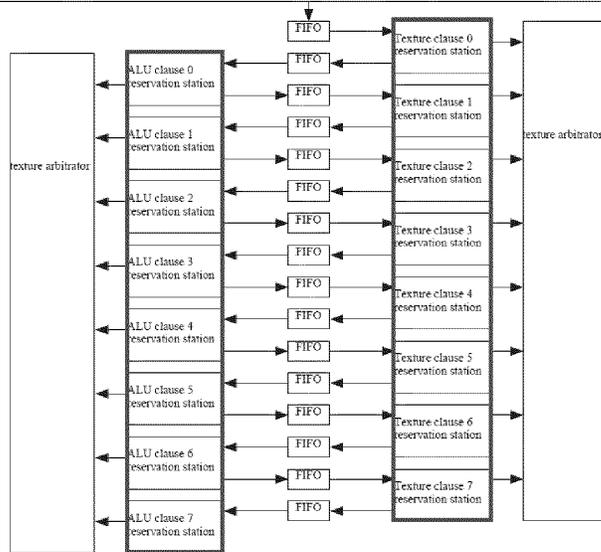
- 138 -

Another memory device disclosed in the R400 Sequencer Specification is an instruction store. *See, e.g.*, Ex. 2010, pp. 11, 12. On the figures shown below, the instruction store is outlined in red. This instruction store is loaded with instructions. *See id.*, pp. 12 ("[the instruction store] may contain up to 2000 instructions of 96 bits each"), 17 (1), 18 (1) (loading pixel and vertex programs into the instruction store).

- 139 -
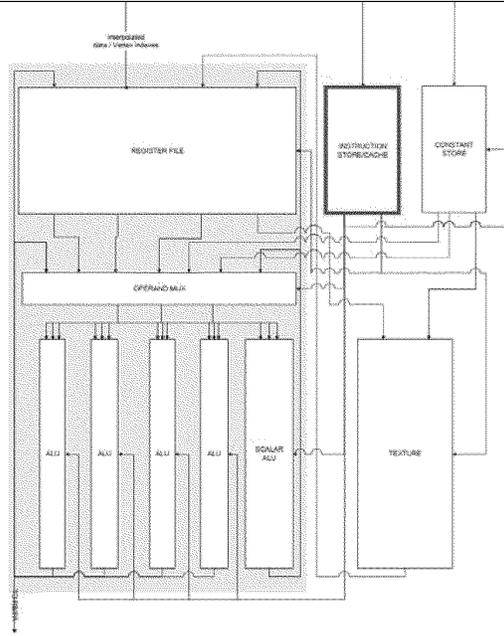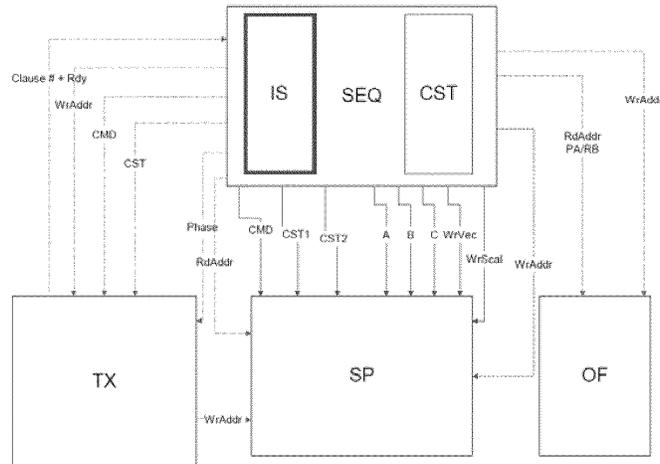
See id., p. 11.

See id., p. 12.

Accordingly, the combination of the reservation stations and

- 140 -

instruction store is the claimed "at least one memory device."

A First Portion Operative to Store Pixel Command Threads and a Second Portion Operative to Store Vertex Command Threads

The instruction store includes storage for vertex instructions and pixel instructions. *See* Ex. 2010, p. 4, 12 ("[t]here is going to be only one instruction store for the whole chip"), 17-19 (using the term "global instruction store"). Since the instruction store is used to store both pixel and vertex command threads, the "first portion" of the claimed "at least one memory device" is the combination of the pixel reservation stations and the portion of the instruction store where pixel instructions reside. The "second portion" of the claimed "at least one memory device" is the combination of the vertex reservation stations and the portion of the instruction store where vertex instructions reside.

The first portion—the pixel reservation stations and the corresponding portion of the instruction store—is operative to store a plurality of pixel command threads as recited in the '053 patent. According to the '053 patent's specification, "a

- 141 -

command thread is a sequence of commands applicable to the corresponding element, such as [a] pixel command thread relative to processing of pixel elements and a vertex command thread relative to vertex processing commands." Ex. 1001, 2:41-45. A clause stored in a pixel reservation station and its corresponding shader program stored in the instruction store are a command thread as described in the '053 patent's specification because the clause and the corresponding shader program are a sequence of commands. *See* Ex. 2010, p. 4 ("[The Sequencer] chooses two ALU clauses and a texture clause to execute, and executes all of the instructions in a clause."); Ex. 2042, p. 8 ("instructions in a clause will be executed sequentially"); Ex. 2010, p. 19 (9) ("TSM0 accepts the control packet and fetches the instructions for texture clause 0 from the instruction store"), 19 (12) ("ASM0 accepts the control packet . . . and gets the instructions for ALU clause 0 from the global instruction store"). Further, the clauses are applicable for the corresponding element because the clauses and the corresponding shader instructions stored in the first portion

- 142 -

are applicable for processing pixel elements. *See id.*, pp. 18-19 (disclosing an example of program executions for pixels).

The second portion—the vertex reservation stations and the corresponding portion of the instruction store—is operative to store a plurality of vertex command threads as recited in the '053 patent. A clause stored in a vertex reservation station and its corresponding shader program stored in the instruction store are a command thread as described in the '053 patent's specification because the clause and the corresponding shader program are a sequence of commands. *See* Ex. 2010, p. 4 ("[The Sequencer] chooses two ALU clauses and a texture clause to execute, and executes all of the instructions in a clause."); Ex. 2042, p. 8 ("instructions in a clause will be executed sequentially"); Ex. 2010, p. 17 (6) ("TSM0 accepts the control packet and fetches the instructions for texture clause 0 from the global instruction store), 17 (9) ("ASM0 accepts the control packet . . . and gets the instructions for ALU clause 0 from the global instruction store"). Further, the clauses are applicable for the corresponding element because

- 143 -

	the clauses and the corresponding shader instructions stored in the second portion are applicable for processing vertex elements. *See id.*, pp. 17-18 (disclosing an example of program executions for vertices).
1b. an arbiter, coupled to the at least one memory device, operable to select a command thread from either of the plurality of pixel command threads and the plurality of vertex command threads based on relative priorities of the plurality of pixel command threads	**An Arbiter Coupled to the At Least One Memory Device** The R400 Sequencer Specification discloses an arbiter. According to the '053 patent's specification, "[t]he arbiter may be any implementation of hardware, software, or combination thereof." Ex. 1001, 2:48-52. The arbiter disclosed in the R400 Sequencer Specification is coupled to both sets of vertex reservation stations and pixel reservation stations. This arbiter comprises multiple levels of arbitration, collectively shown in red on the figure below.

- 144 -
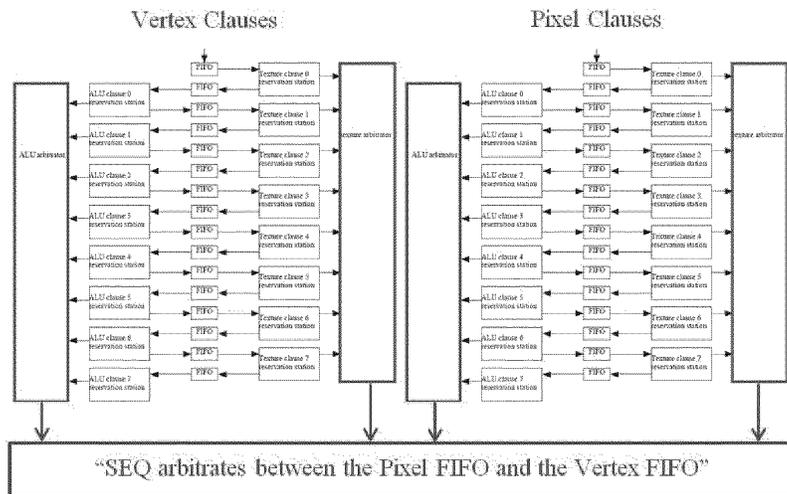
and the plurality of vertex command threads.	Vertex Clauses Pixel Clauses "SEQ arbitrates between the Pixel FIFO and the Vertex FIFO" The first level of arbitration is between ALU clauses and texture clauses, respectively, of a single type. *See* Ex. 2010, p. 4 ("[The Sequencer] chooses two ALU clauses and a texture clause to execute . . . . The sequencer looks at all eight alu reservation stations to choose an alu clause to execute and all eight texture stations to choose a texture clause to execute."), 17 (6, 9), 19 (9, 12). This first arbitration is represented by the ALU arbitrators and the texture arbitrators, each of which are outlined in red in the figure above.[1]

---

[1] The left texture arbitrator is mislabeled in the original specification because this arbitrator corresponds to the ALU reservation stations. *See* Ex. 2010, p. 5.

The second level of arbitration is between the candidate pixel thread and the candidate vertex thread. *See* Ex. 2010, pp. 17 (2), 18 (4) ("SEQ arbitrates between the Pixel FIFO and the Vertex FIFO"). So, the arbiter not only selects which ALU/texture clauses to execute, the arbiter selects which order to execute pixels and vertices. *See id.*, p. 4 ("There are two separate sets of reservation stations . . . . This way a pixel can pass a vertex and a vertex can pass a pixel.") Along with the first arbitration, the second arbitration forms the arbiter disclosed in the '053 patent.

Operable to Select a Command Thread From Either of the Pixel Command Threads and the Vertex Command Threads

The arbiter is operable to select a command thread from either of the pixel command threads and the vertex command threads. As was previously discussed, the arbitration logic has two levels of arbitration. The first level is selecting ALU clauses and texture clauses for both the vertex and the pixel threads. *See, e.g.*, Ex. 2010, pp. 4, 5. The second level is between the vertex and the pixel threads. *See id.*, pp. 17 (2),

- 146 -

18 (4) ("SEQ arbitrates between the Pixel FIFO and the Vertex FIFO"). Collectively these two levels of arbitration make the arbiter operable to select a command thread from either of the pixel command threads and the vertex command threads.

Based on Relative Priorities

The arbiter is operable to select clauses based on relative priorities of the pixel clauses and the vertex clauses. According to the R400 Sequencer Specification, "[t]he arbitrator will give priority to clauses/reservation stations closer to the bottom of the pipeline." Ex. 2010, p. 4. When arbitrating between the pixel and the vertex, the vertex has priority. *Id.*, p. 17 (2) ("SEQ arbitrates between the Pixel FIFO and the Vertex FIFO – basically the Vertex FIFO always has priority . . . . [T]he arbiter is not going to select a vector to be transformed if the parameter cache is full unless the pipe has nothing else to do."). When there are no vertices pending or there is no space left in the register files for vertices, the arbiter selects the pixel. *Id.*, p. 18 (4) ("SEQ

- 147 -

	arbitrates between Pixel FIFO and Vertex FIFO – where there are no vertices pending OR there is no space left in the register files for vertices, the Pixel FIFO is selected.").
2. The graphics processing system of claim 1, further comprising: a command processing engine, coupled to the arbiter, wherein the arbiter is further operable to provide the command thread to the command processing engine.	**Command Processing Engines** As shown in the figures reproduced below, the R400 Sequencer Specification shows ALUs and a texture unit, each of which is a command processing engine. According to the '053 patent's specification, "[a] command processing engine may be any suitable engine as recognized by one having ordinary skill in the art for processing commands, ***such as a texture engine, an arithmetic logic unit***, or any suitable processing engine." Ex. 1001, 2:59-62 (emphasis added).

Ex. 2010, p. 11.

Id., p. 12.

- 149 -
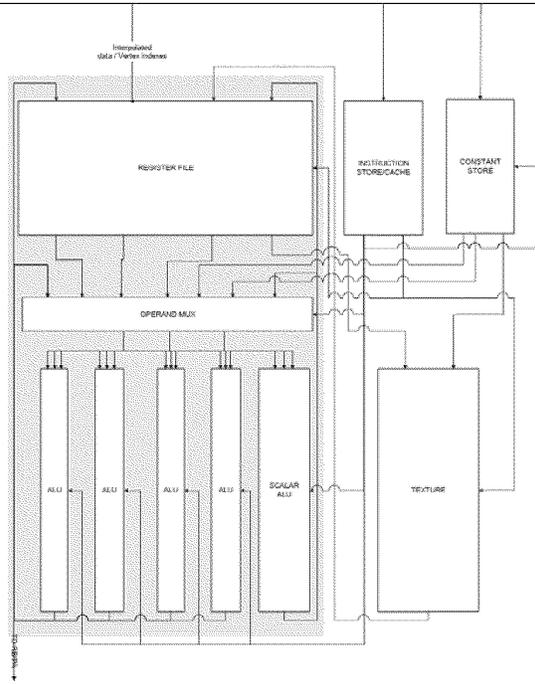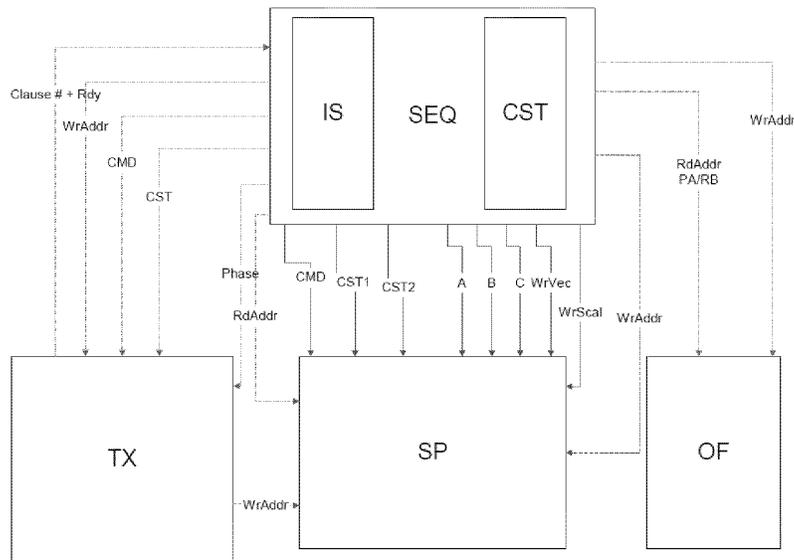
The ALUs are command processing engines. ALUs are command processing engines according to the '053 patent's specification. *See* Ex. 1001, 2:59-62. Further, each ALU is a command processing engine because each ALU "can do simple math, conditional moves, and permutations." *See* Ex. 2041, p. 10.

The texture unit is also a command processing engine because the texture unit processes commands. *See* Ex. 2010, pp. 17 (8), 19 (11) (the texture unit completes requests).

Coupled to the Arbiter

As shown in the figures reproduced in this section, the command processing engines are coupled to the arbiter. The ALUs are part of the shader pipe, and the shader pipe is coupled to the sequencer. *See* Ex. 2010, pp. 11, 12. *See also id.*, pp. 17 (4) ("SEQ sends the vector to the SP register file over the RE_SP interface"), 18 (7) ("SEQ controls the transfer of interpolated data to the SP register file over the RE_SP interface"). Further, the arbiter is part of the sequencer. *See id.*, pp. 5, 17 (2), 18 (4); *supra* Claim 1b (showing support for

- 150 -

the arbiter). So the ALUs are coupled to the arbiter.

The texture unit is also connected to the sequencer. As shown in the figures reproduced in this section, the texture unit is coupled to the sequencer. *See id.*, pp. 11, 12. *See also id.*, pp. 11, 16 (for the shader engine to texture unit bus and the sequencer to texture unit bus). The arbiter is part of the sequencer. *See id.*, pp. 5, 17 (2), 18 (4); *supra* Claim 1b (showing support for the arbiter). So the texture unit is coupled to the arbiter.

Operable to Provide the Command Thread to the Command Processing Engine

The arbiter is operable to provide the command thread to the ALUs because after the arbiter selects a command thread, the arbiter sends the command thread to the shader pipe using a sequencer to shader engine bus. *See* Ex. 2010, p. 15 ("This is a bus that sends the instruction . . . to all 4 Sub-Engines of the Shader."). This bus is shown in red on the annotated figure below.

- 151 -

Interpolated
data / Vertex indexes

REGISTER FILE

INSTRUCTION
STORE/CACHE

CONSTANT
STORE

OPERAND MUX

ALU ALU ALU ALU SCALAR
ALU
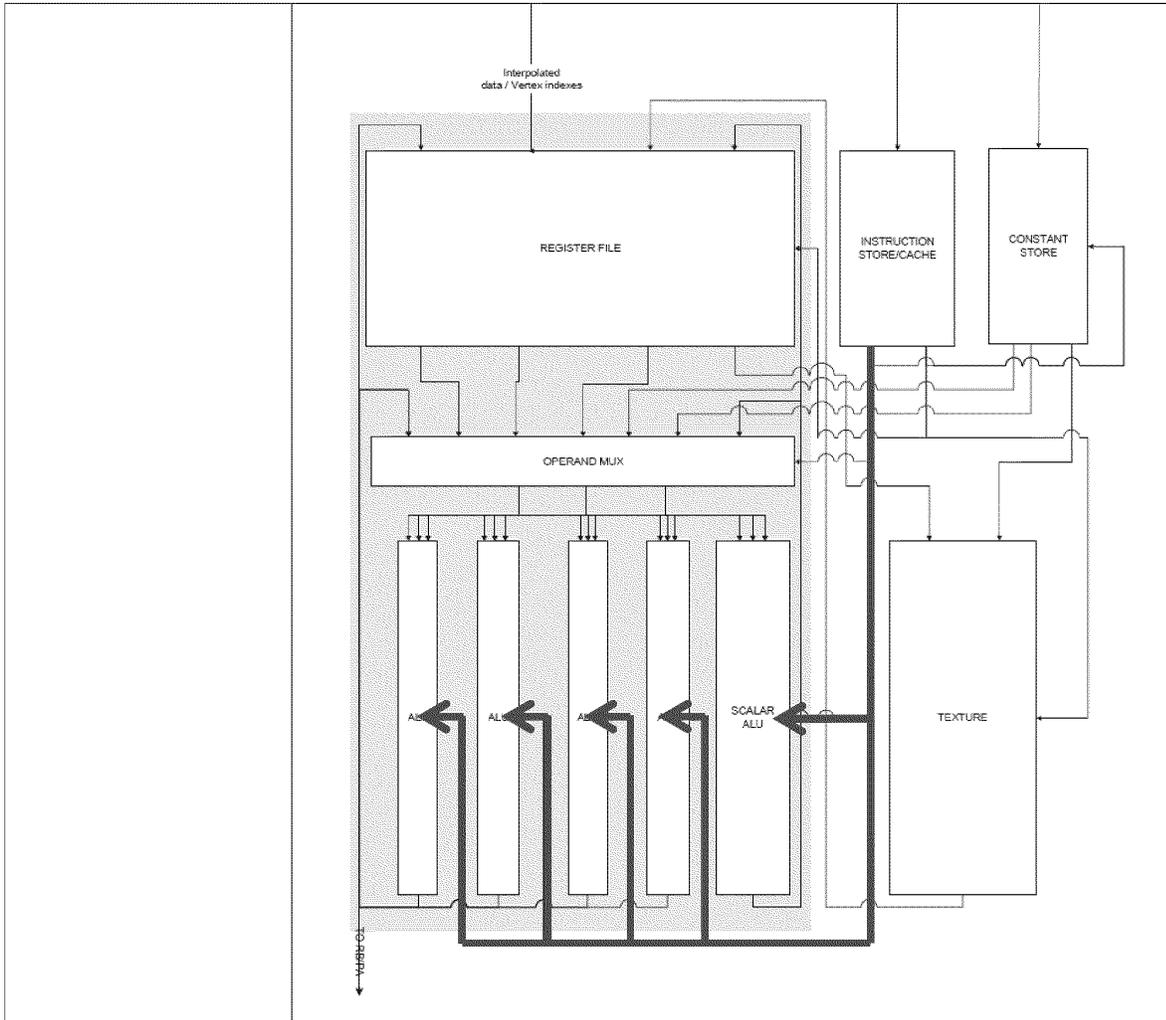
TEXTURE

See Ex. 2010, p. 11.

The R400 Sequencer Specification also shows sending the

"CMD" to the shader pipe in the annotated figure below. *See*

id., p. 12. *See also id.*, pp. 17 (10), 19 (13).

- 152 -
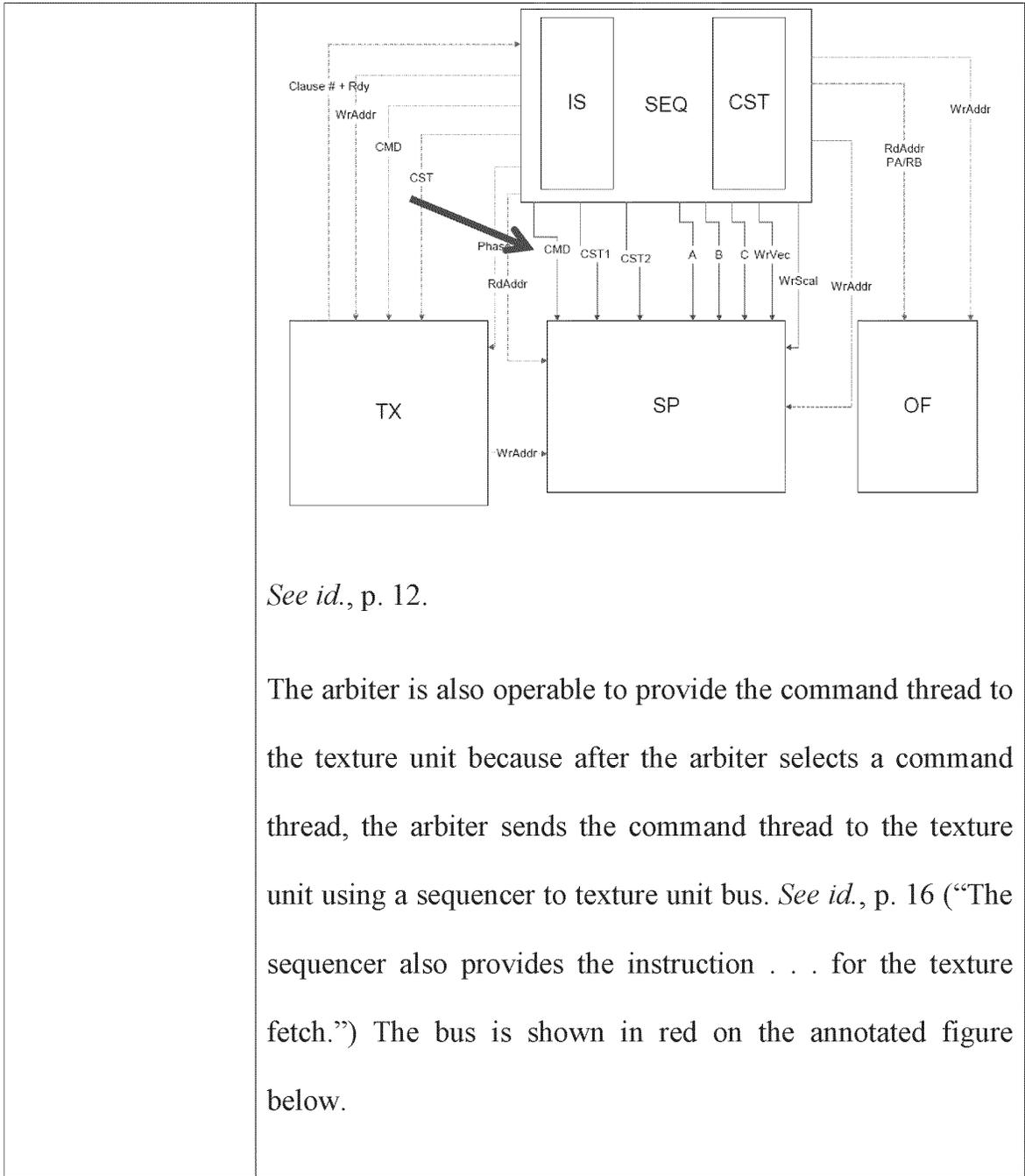
See id., p. 12.

The arbiter is also operable to provide the command thread to the texture unit because after the arbiter selects a command thread, the arbiter sends the command thread to the texture unit using a sequencer to texture unit bus. *See id.*, p. 16 ("The sequencer also provides the instruction . . . for the texture fetch.") The bus is shown in red on the annotated figure below.

- 153 -

See Ex. 2010, p. 11.

The R400 Sequencer Specification also shows sending the "CMD" to the texture unit in the annotated figure below. *See id.*, p. 12. *See also id.*, pp. 17 (7), 19 (10).
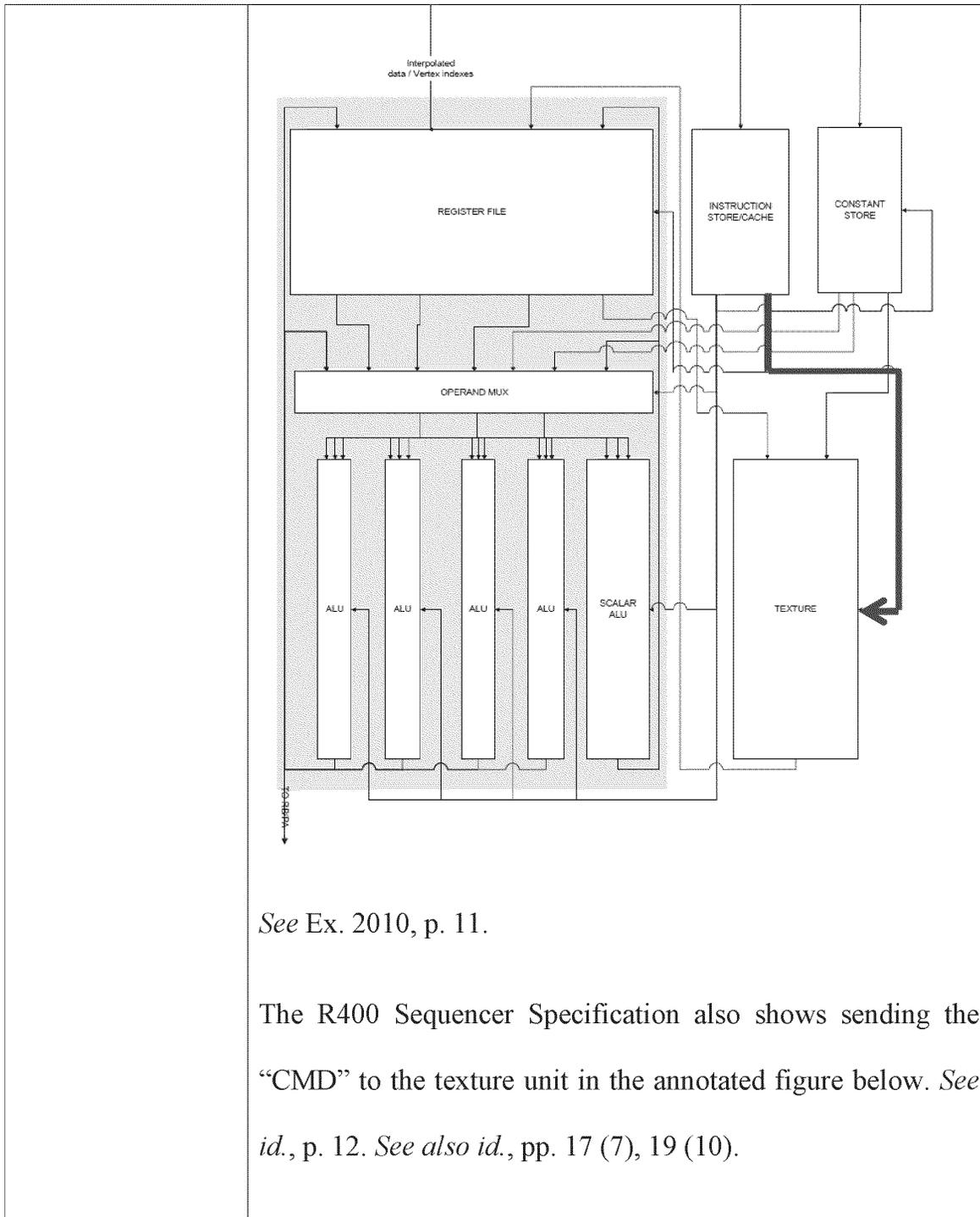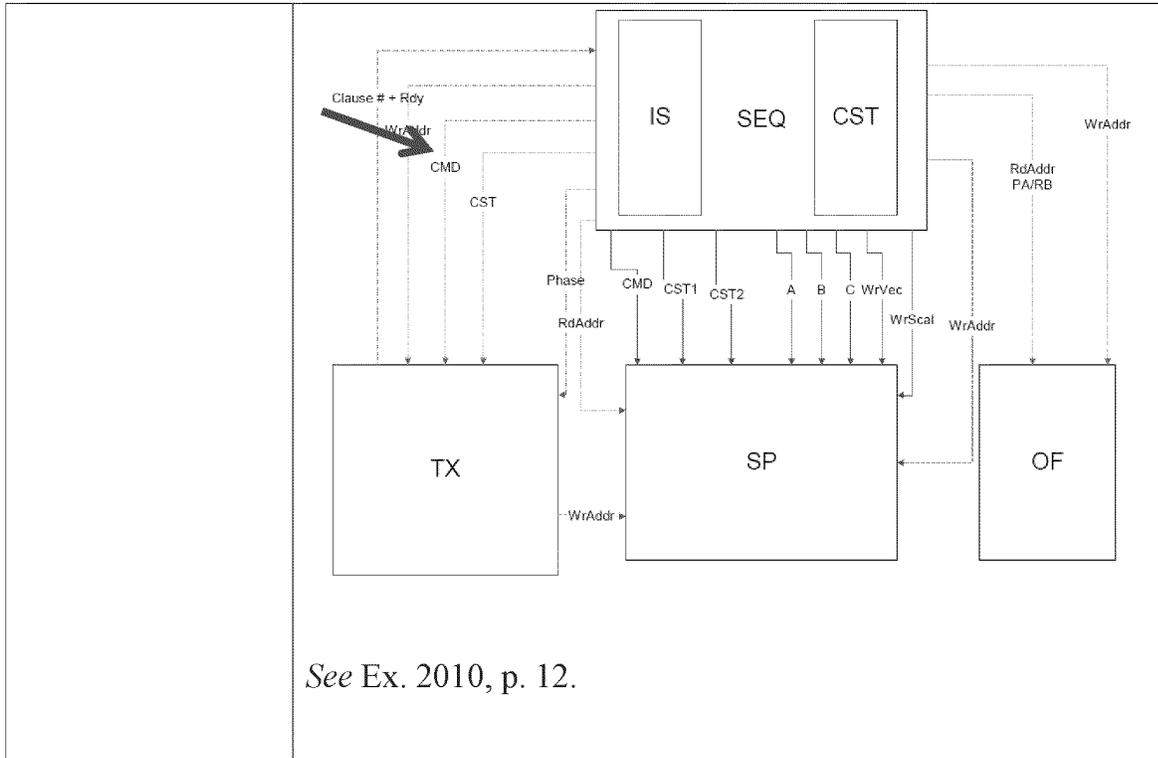
See Ex. 2010, p. 12.

5a. at least one memory device comprising a first portion operative to store a plurality of pixel command threads and a second portion operative to store a plurality of vertex	*See supra* Claim 1a (showing support for the same claim language).

- 155 -

command threads;	
5b. an arbiter, coupled to the at least one memory device, operable to select a command thread from either of the plurality of pixel command threads and the plurality of vertex command threads; and	*See supra* Claim 1b (showing support for the same claim language).
5c. a plurality of command processing engines, coupled to the arbiter, each operable to receive	*See supra* Claim 2 (showing support for "a plurality of command processing engines, coupled to the arbiter" and "the arbiter . . . operable to provide the command thread to the command processing engine"). **Operable to Receive and Process the Command Thread** The ALUs are each operable to receive and process the

- 156 -

| and process the command thread. | command thread. Each ALU is operable to receive the command thread from the sequencer via the sequencer to shader engine bus. *See* Ex. 2010, p. 15. Each ALU is operable to process the command thread because each ALU "can do simple math, conditional moves, and permutations on the registers." *See* Ex. 2041, p. 10.

The texture unit is also operable to receive and process the command thread. The texture unit is operable to receive the command thread from the sequencer via the sequencer to texture unit bus. *See* Ex. 2010, p. 16. The texture unit is operable to process the command thread because the texture unit executes the instructions. *See id.* ("The sequencer . . . provides the instruction . . . for the texture fetch to execute."). |
| 6. The graphics processing system of claim 5, wherein the plurality of command | *See supra* Claims 2 and 5c (showing support for the claim language). |

- 157 -

processing engines comprises at least one arithmetic logic unit.	
7. The graphics processing system of claim 5, wherein the plurality of command processing engines comprises at least one texture processing engine.	*See supra* Claims 2 and 5c (showing support for the claim language).

- 158 -

R400 SEQUENCER SPEC VERSION 2.0	
Claim	
1. A graphics processing system comprising	The R400 Sequencer Specification is an architectural specification for the R400's sequencer block. Ex. 2028, p. 1. The R400 was a graphics-chip product, and the R400 was designed to include a unified pipe (i.e., a single programmable pipeline for 2D video, 3D vertex, and 3D pixel operations). *See* Ex. 2041, pp. 6, 7.
1a. at least one memory device comprising a first portion operative to store a plurality of pixel command threads and a second portion operative to store a plurality of vertex command threads; and	**At Least One Memory Device** The R400 Sequencer Specification describes reservation stations and an instruction store, which collectively are the claimed "at least one memory device." The R400 Sequencer Specification includes at least one memory device. The sequencer's control flow diagram, reproduced below for reference, shows a vertex reservation station (VTX RS) and a pixel reservation station (PIX RS). *See* Ex. 2028, pp. 6 ("[t]here are two separate reservation stations, one for pixel vectors and one for vertices vectors"), 10. The reservation stations are also called buffers. *See id.*, p. 25 ("[T]wo buffers are maintained – one for Vertices and one

for Pixels."). The buffers store threads. *See id.*, p. 25 ("A
thread lives in a given location in the buffer during its entire
life."). The buffers are even called "pixel or vertex memory."
id., p. 23. The R400 Sequencer Specification further states,
"Each entry in the buffer will be stored across two physical
pieces of memory." *id.*, p. 26.
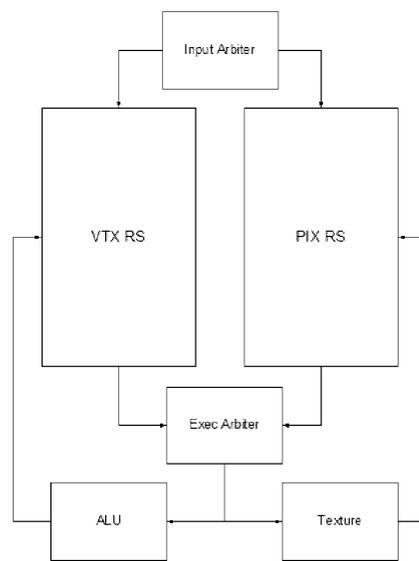
id., p. 10.

Another memory device disclosed in the R400 Sequencer
Specification is an instruction store. *See, e.g., id.*, pp. 7, 14,
17. On the figures shown below, the instruction store is
outlined in red. This instruction store is loaded with

- 160 -

instructions. *id.*, p. 17 ("[the instruction store] will contain

4096 instructions of 96 bits each"). Also, the instruction store

is called a memory. *See id.* ("[the instruction store] is likely to

be a 1 port memory").

See id., p. 7.

- 161 -
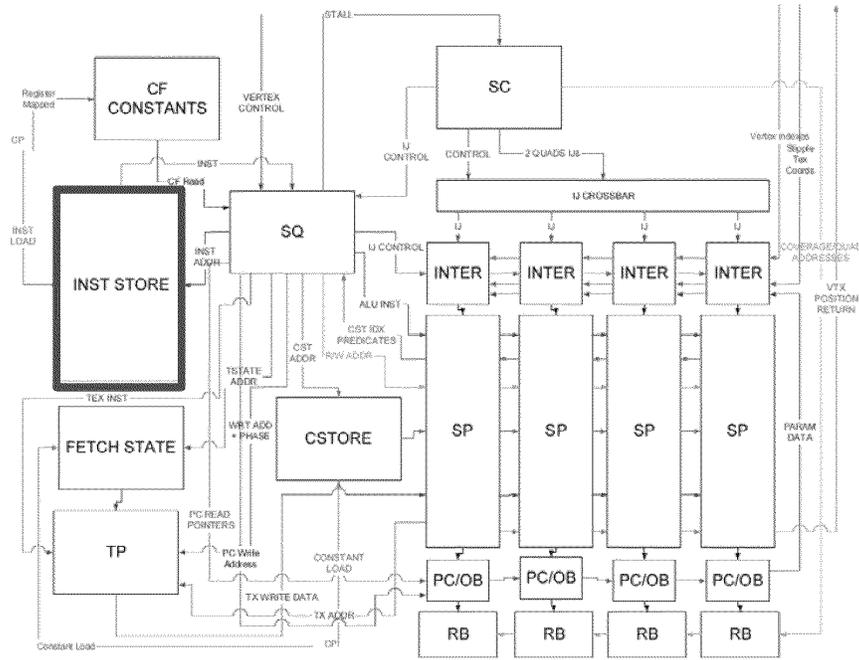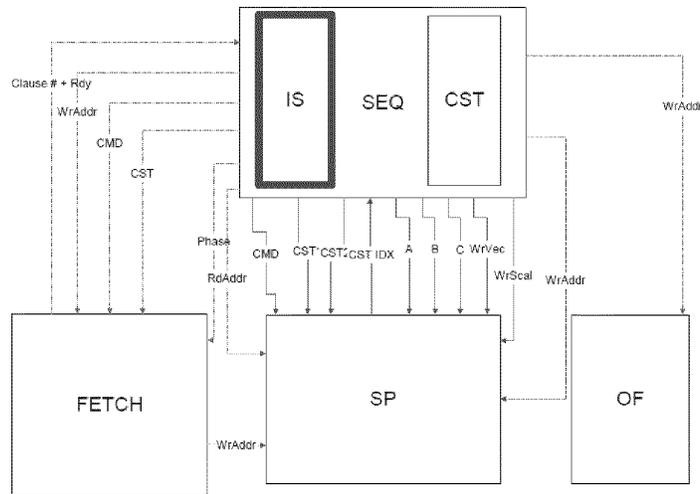
See id., p. 14.

A First Portion Operative to Store Pixel Command Threads and a Second Portion Operative to Store Vertex Command Threads

The instruction store includes storage for vertex instructions and pixel instructions. *See* Ex. 2028, p. 17 ("There is going to be only one instruction store for the whole chip."). Since the instruction store is used to store both pixel and vertex command threads, the "first portion" of the claimed "at least one memory device" is the combination of the pixel reservation station and the portion of the instruction store where pixel instructions reside. The "second portion" of the claimed "at least one memory device" is the combination of

- 162 -

the vertex reservation station and the portion of the instruction

store where vertex instructions reside.

The first portion—the pixel reservation station and the

corresponding portion of the instruction store—is operative to

store a plurality of pixel command threads as recited in the

'053 patent. According to the '053 patent's specification, "a

command thread is a sequence of commands applicable to the

corresponding element, such as [a] pixel command thread

relative to processing of pixel elements and a vertex command

thread relative to vertex processing commands." Ex. 1001,

2:41-45. A thread stored in the pixel reservation station and its

corresponding shader instructions stored in the instruction

store are a command thread as described in the '053 patent's

specification because the thread and the corresponding shader

instruction are a sequence of commands. *See* Ex. 2028, p. 26

("[the thread] is returned to the buffer . . . once all possible

sequential instructions have been executed"). Regarding the

sequence, each entry in the buffer has a "state" and a "status."

See id., p. 26. The "state" includes a "Control Flow

Instruction Pointer." *See id.*, p. 26. The pointer is to the

- 163 -

instruction store. *See id.*, p. 41 (INST_BASE_PIX and PS_BASE). And the corresponding instructions are a sequence of an ALU instruction, a fetch instruction, and control flow instructions. *See id.*, p. 17. Further, the threads are applicable for the corresponding element because the threads and the corresponding instructions stored in the first portion are applicable for processing pixel elements. *See id.*, pp. 10 (showing separate pixel and vertex reservation stations), 12 ("the sequencer (SQ) will only use one global state management machine per vector type (pixel, vertex) that we call the reservation station (RS), 25 ("A thread lives in a given location in the buffer during its entire life.").

The second portion—the vertex reservation stations and the corresponding portion of the instruction store—is operative to store a plurality of vertex command threads as recited in the '053 patent. A thread stored in the vertex reservation station and its corresponding shader instructions stored in the instruction store are a command thread as described in the '053 patent's specification because the thread and the corresponding shader program are a sequence of commands.

	See id., p. 26 ("[the thread] is returned to the buffer . . . once all possible sequential instructions have been executed"). Regarding the sequence, each entry in the buffer has a "state" and a "status." *See id.*, pp. 26. The "state" includes a "Control Flow Instruction Pointer." *See id.*, p. 26. The pointer is to a location in the instruction store. *See id.*, p. 41 (INST_Base_VTX and VTX_BASE). And the corresponding instructions are a sequence of commands that make up an ALU instruction, a fetch instruction, and control flow instructions. *See* Ex. 2028, p. 17. Further, the threads are applicable for the corresponding element because the threads and the corresponding instructions stored in the second portion are applicable for processing vertex elements. *See id.*, pp. 10 (showing separate pixel and vertex reservation stations), 12 ("the sequencer (SQ) will only use one global state management machine per vector type (pixel, vertex) that we call the reservation station (RS), 25 ("A thread lives in a given location in the buffer during its entire life.").
1b. an arbiter,	**An Arbiter Coupled to the At Least One Memory Device**

coupled to the at least one memory device, operable to select a command thread from either of the plurality of pixel command threads and the plurality of vertex command threads based on relative priorities of the plurality of pixel command threads and the plurality of vertex command threads.	The R400 Sequencer Specification discloses an arbiter. According to the '053 patent's specification, "[t]he arbiter may be any implementation of hardware, software, or combination thereof." Ex. 1001, 2:48-52. The sequencer's control flow diagram, reproduced below for reference, shows an Exec Arbiter coupled to the reservation stations. Ex. 2028, p. 10. **Operable to Select a Command Thread From Either of the Pixel Command Threads and the Vertex Command Threads** The arbiter is operable to select a pixel thread or a vertex thread. For both vertices and pixels, the arbitration circuit
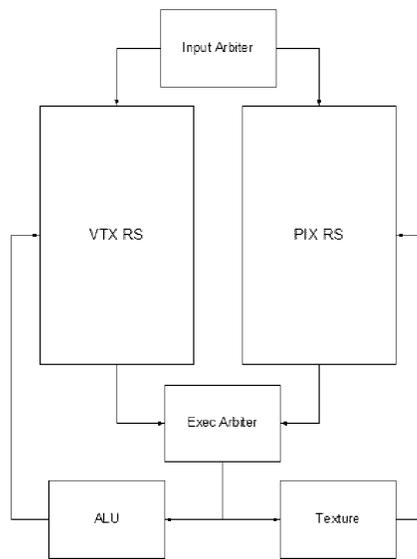
selects a winner for both the texture engine and the ALU engine. *See* Ex. 2028, p. 26 ("The arbitration circuitry will select a winner for both the Texture Engine and for the ALU engine. There are actually two sets of arbitration – one for pixels and one for vertices.") This is the first level of arbitration. The arbiter then selects between the pixel and the vertex. *See id.*, p. 26 ("A final selection is then done between the two."). This is the second level of arbitration. Collectively this arbitration make the arbiter operable to select a command thread from either of the pixel command threads and the vertex command threads.

Based on Relative Priorities

The arbiter is operable to select threads based on relative priorities of the pixel threads and the vertex threads. According to the R400 Sequencer Specification, priority is given to older threads. *See* Ex. 2028, pp. 6 ("The arbitrator will give priority to older threads."), 25 ("the buffer has FIFO qualities in that threads leave in the order that they enter"); 26-27 ("Texture arbitration requires no allocation or ordering

- 167 -

so it is purely based on selecting the 'oldest' thread that requires the Texture Engine. ALU arbitration is a little more complicated. First, only threads where either of Texture_Reads_outstanding or Waiting_on_Texture_Read_to_Complete are '0' are considered. Then if Allocation_Wait is active, these threads are further filtered based on whether space is available. If the allocation is position allocation, then the thread is only considered if all 'older' threads have already done their position allocation (position allocated bits set). If the allocation is parameter or pixel allocation, then the thread is only considered if it is the oldest thread. Also a thread is not considered if it is a parameter or pixel or position allocation, has its First_thread_of_a_new_context bit set and would cause ALU interleaving with another thread performing the same parameter or pixel or position allocation. Finally the 'oldest' of the threads that pass through the above filters is selected. If the thread needed to allocate, then at this time the allocation is done, based on Allocation_Size. If a thread has its 'last' bit set, then it is also removed from the buffer, never

- 168 -

	to return.").
2. The graphics processing system of claim 1, further comprising: a command processing engine, coupled to the arbiter, wherein the arbiter is further operable to provide the command thread to the command processing engine.	**Command Processing Engines** As shown in the figure reproduced below, the R400 Sequencer Specification shows an ALU and a texture engine, each of which is a command processing engine. *See* Ex. 2028, pp. 10, 26 (using the terms "ALU engine" and "Texture engine"). According to the '053 patent's specification, "[a] command processing engine may be any suitable engine as recognized by one having ordinary skill in the art for processing commands, ***such as a texture engine, an arithmetic logic unit***, or any suitable processing engine." Ex. 1001, 2:59-62 (emphasis added). Input Arbiter VTX RS PIX RS Exec Arbiter ALU Texture

- 169 -
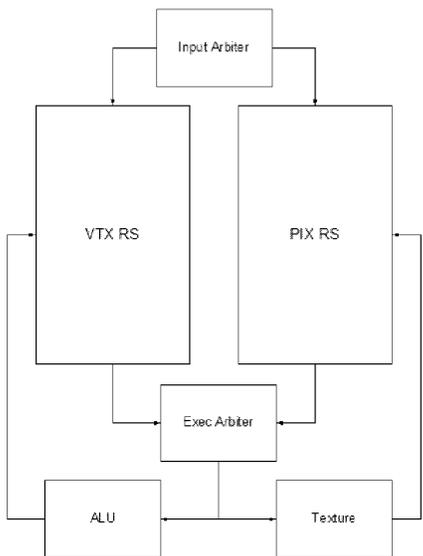
Ex. 2028, p. 10.

The ALU and the texture engine are command processing engines according to the '053 patent's specification. *See* Ex. 1001, 2:59-62. Further, each is a command processing engine because each engine processes commands. *See* Ex. 2028, p. 26 ("Once a thread is selected it is read out of the buffer . . . and submitted to [the] appropriate execution unit. It is returned to the buffer . . . once all possible sequential instruction shave been executed.").

Coupled to the Arbiter

As shown in the figure reproduced in this section, the command processing engines are coupled to the arbiter. *See* Ex. 2028, p. 10 (showing ALU and texture engines coupled to the Exec Arbiter).

Operable to Provide the Command Thread to the Command Processing Engine

The arbiter is operable to provide the command thread to both the ALU engine and the texture engine because after the arbiter selects a command thread, the arbiter submits the command thread to the appropriate execution unit. *See* Ex.

- 170 -

2028, p. 26 ("Once a thread is selected it is read out of the buffer, marked as invalid, and submitted to [sic] appropriate execution unit. It is returned to the buffer . . . once all possible sequential instructions have been executed.").

The figures in the R400 Sequencer Specification, reproduced below with annotations, also show that the arbiter is operable to provide the command thread to the command processing engines. The first figure shows the ALU instruction ("ALU INST") from the sequencer ("SQ") to the shader pipe ("SP"). The second figure shows the command ("CMD") from the SQ to the SP. The third figure shows the texture instruction ("TEX INST") from the SQ to the texture pipe ("TP"). The fourth figure shows the CMD from the SQ to the TP Fetch.

- 171 -

See Ex. 2028, p. 7.

*See id.*, p. 14.

- 172 -

See id., p. 7.

See id., p. 14.

- 173 -

5a. at least one memory device comprising a first portion operative to store a plurality of pixel command threads and a second portion operative to store a plurality of vertex command threads;	*See supra* Claim 1a (showing support for the same claim language).
5b. an arbiter, coupled to the at least one memory device, operable to select a command thread from either of the plurality of	*See supra* Claim 1b (showing support for the same claim language).

- 174 -

pixel command threads and the plurality of vertex command threads; and	
5c. a plurality of command processing engines, coupled to the arbiter, each operable to receive and process the command thread.	*See supra* Claim 2 (showing support for "a plurality of command processing engines, coupled to the arbiter" and "the arbiter . . . operable to provide the command thread to the command processing engine"). **Operable to Receive and Process the Command Thread** The figure showing the ALU engine and the texture engine is reproduced below. Both are operable to receive and process the command thread because, after the command thread is submitted to the appropriate execution unit, the receiving engine executes the instructions. *See* Ex. 2028, p. 26 ("Once a thread is selected it is read out of the buffer, marked as invalid, and submitted to [sic] appropriate execution unit. It is returned to the buffer . . . once all possible sequential
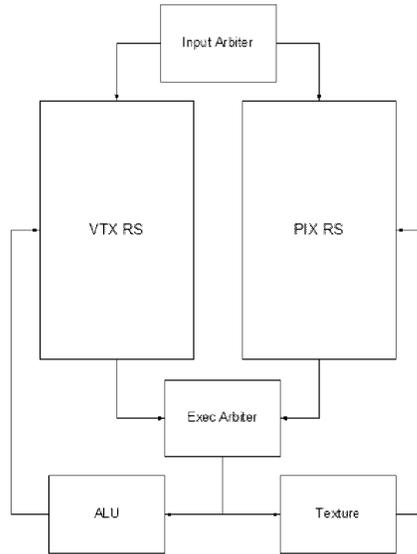
|	
	instructions have been executed."). *Id.*, p. 10.
6. The graphics processing system of claim 5, wherein the plurality of command processing engines comprises at least one arithmetic	(*See supra* Claims 2 and 5c (showing support for the claim language).)

- 176 -

logic unit.	
7. The graphics processing system of claim 5, wherein the plurality of command processing engines comprises at least one texture processing engine.	(*See supra* Claims 2 and 5c (showing support for the claim language).)

- 177 -

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 '053 patent.

Executed this 8th day of September in Los Gatos, CA

Respectfully submitted,

Andrew Wolfe