(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 778 536 B1

(12) EUROPEAN PATENT SPECIFICATION

(45) Date of publication and mention of the grant of the patent:
21.05.2003 Bulletin 2003/21

(51) Int Cl.[7]: **G06F 19/00**, G06T 15/00

(21) Application number: **96308481.9**

(22) Date of filing: **22.11.1996**

(54) High performance low cost video game system with coprocessor providing high speed efficient 3-D graphics and digital audio signal processing

Preisgünstiges Hochleistungsvideospielsystem mit Koprozessor zur Bereitstellung von effizienten Hochgeschwindigkeits-dreidimensionalen Graphiken und digitaler Audio-Signalverarbeitung

Système de jeu vidéo à haute performance et de faible coût avec coprocesseur permettant une efficacité à haute vitesse graphique en trois dimensions et traitement numérique de signaux audio

(84) Designated Contracting States:
DE FR GB

(30) Priority: **22.11.1995 US 561718**

(43) Date of publication of application:
11.06.1997 Bulletin 1997/24

(73) Proprietors:
- **Nintendo Co., Limited**
  Minami-ku, Kyoto 601 (JP)
- **SILICON GRAPHICS, INC.**
  Mountain View, CA 94039-7311 (US)

(72) Inventors:
- **Van Hook, Timothy J.**
  Atherton, California 94027 (US)
- **Delaurier, Anthony P.**
  Sunnyvale, California 94086 (US)
- **Cheng, Howard Hao**
  Virginia 98052 (US)
- **Gosset, Carroll Philip**
  California 94043 (US)
- **Moore, Robert Jordan**
  Florida 32779 (US)
- **Anderson, Harold Stephen**
  California 95037 (US)
- **Shepard, Stephen J.**
  California 95014 (US)
- **Princen, John**
  California 95124 (US)
- **Doughty, Jeffrey Chandler**
  California 94301 (US)
- **Sheppard, Byron**
  California 95060 (US)
- **Pooley, Nathan F.**
  California 94041 (US)
- **Takeda, Genyo, c/o Nintendo Co., Ltd**
  Kyoto-shi, Kyoto-fu (JP)
- **Kato, Shuhei, c/o Nintendo Co., Ltd**
  Kyoto-shi, Kyoto-fu (JP)

(74) Representative: **Perkins, Sarah et al**
Stevens, Hewlett & Perkins
Halton House
20/23 Holborn
London EC1N 2JD (GB)

(56) References cited:
EP-A- 0 627 699 **EP-A- 0 681 267**
US-A- 5 357 604

- **JOHNSON M: "A FIXED-POINT DSP FOR GRAPHICS ENGINES" IEEE MICRO, vol. 9, no. 4, August 1989 (1989-08), pages 63-77, XP000049920**
- **WAYNER P: "SILICON FOR 3-D" BYTE, vol. 19, no. 9, 1 September 1994 (1994-09-01), page 191/192 XP000473308**

EP 0 778 536 B1

Description

FIELD OF THE INVENTION

[0001] The present invention relates to low cost video game systems. More particularly, the invention relates to a video game system that can model a world in three dimensions and project the model onto a two dimensional viewing plane selected based on a changeable viewpoint.

BACKGROUND AND SUMMARY OF THE INVENTION

[0002] People's imaginations are fueled by visual images. What we actually see at sunset, what we dream at night, the pictures we paint in our mind when we read a novel -- all of these memorable scenes are composed of visual images. Throughout history, people have tried to record these images with pencils or paints or video tape. But only with the advent of the computer can we begin to create images with the same vividness, detail and realism that they display in the teal world or in the imagination.

[0003] Computer-based home video game machines such as the Nintendo Entertainment System and the Super Nintendo Entertainment System have been highly successful because they can interactively produce exciting video graphics. However, without additional add-on hardware, these prior video graphics systems generally operated in two dimensions, creating graphics displays from flat (planar) image representations in a manner somewhat analogous to tacking flat paper cutouts onto a bulletin board. Although very exciting game play can be created using two dimensional graphics techniques, a 2D system cannot provide the realism offered by three-dimensional graphics system.

[0004] 3D graphics arc fundamentally different from 2D graphics. In 3D graphics techniques, a "world" is represented in three dimensional space. The system can allow the user to select a viewpoint within the world. The system creates an image by "projecting" the world based on the selected viewpoint. The result is a true three-dimensional image having depth and realism.

[0005] For many years, specialists have used super computers and high end workstations to create incredible realistic 3D images -- for example, ultra-detailed models of cars, planes and molecules; virtual reality as seen from the cockpit of a jet fighter or the front seat of an Olympic bobsled; and dinosaurs of "Jurassic Park." However, in the past, computer systems required to produce such images interactively cost tens of thousands of dollars -- well beyond the reach of the average consumer.

[0006] The low cost high performance 3D graphics system disclosed herein is intended to for the first time give millions of game players, not just the specialists, the chance to interact right inside these magnificent virtual 3D worlds with a richly featured high performance low cost system. What players get is truly amazing -- many times the power of any home computer system, far more realistic 3-dimensional animation, stunning graphics -- all delivered at a sufficiently low coat to be within the reach of the average consumer.

[0007] The following are a few examples of the many advantageous features provided by a system in accordance with the present invention:

- Realistic interactive 3D graphics in a low price system
- Optimum feature set/architecture for a low cost system for use with a color television set to provide video game play and other graphics applications in a low cost system and/or to produce particular screen effects
- Coprocessor that provides high performance 3D graphics and digital sound processing
- Signal processor sharing between graphics digital processing and audio signal processing to achieve high quality stereo sound and 3-D graphics in a low cost color television based system
- Unified RAM approach increases flexibility
- All major system components can communicate through the shared RAM
- Techniques/structures for compensating for narrow main memory bus width
- Executable code from a storage device (e.g., a portable memory cartridge) can be loaded into the common RAM and accessed by the main processor through coprocessor memory access/arbitration circuitry
- Graphics coprocessor loadable microcode store receives microcode from a portable storage medium to provide

additional flexibility and simplify compatibility issues

- Microcode is loaded via execution of "boot ROM" instructions
- Optimal commands and associated formats are used to invoke graphics and audio functions within the coprocessor and provide an interface between the graphics coprocessor and the rest of the system
- Coprocessor register set including particular hardware register definitions, formats and associated functions
- Microcode graphics and audio structure/processes provide efficient high performance operation
- Vector unit provides optimal performance for graphics and audio digital processing in a low cost package
- Pipelined rasterizing engine provides a one-pixel-per-cycle and two-pixel-per-cycle modes to minimize hardware cost while providing a rich feature set
- Low coprocessor pin out

BRIEF DESCRIPTION OF THE DRAWINGS

[0008] These and other features and advantages of the present invention will be better and more completely understood by referring to the following detailed description of a presently preferred exemplary embodiment in connection with the drawings, of which:

Figure 1 shows an overall video game system capable of generating 3-D images and digitally processed stereo sound;
Figures 1A-1F show example 3-D screen effects achievable using the Figure 1 system;
Figure 2 shows an example of principal components of an overall video game system;
Figure 3 shows example major processing operations of an overall video game system;
Figure 4 shows example overall operation of a video game system;
Figure 4A shows example overall steps performed by a video game system to generate graphics images;
Figure 5 shows a detailed overall system architecture example;
Figure 5A shows an example main processor initialization routine;
Figure 5B shows an example main processor memory map;
Figure 6 shows an example coprocessor internal architecture;
Figure 6A shows an example coprocessor internal bus architecture;
Figure 7 shows an example signal processor internal architecture;
Figure 7A shows an example signal processor instruction format;
Figure 7B shows an example slicing of the Figure 7A source or destination field for processing by the vector unit shown in Figure 7;
Figure 7C shows an example add operation performed by the example signal processor vector unit;
Figure 7D-7L show example signal processor registers;
Figure 8 shows an example hierarchical task list including graphics display lists and audio play lists;
Figure 9 shows an example microcode load routine;
Figure 10 shows an example simple signal processor display list processing example;
Figure 11 shows an example signal processor graphics microcode control step sequence;
Figure 12A shows an example double precision representation;
Figure 12B shows an example matrix format;
Figure 13A shows an example signal processor vertex buffer format;
Figure 13B shows an example vertex data definition;
Figure 13C shows an example signal processor segment addressing arrangement;
Figure 14 shows an example audio software architecture;
Figure 15 shows an example simple signal processor play list processing example;
Figure 16 shows an example signal processor audio microcode control step sequence;
Figure 17 shows an example signal processor audio processing construct;
Figure 18 shows example overall display processor processing steps;
Figures 19A and 19B show example display processor pipeline configurations;
Figure 20 shows an example display processor architecture;

Figures 21A-21J show example display processor registers;
Figure 22 shows an example texture memory tile descriptor arrangement;
Figure 23 shows an example texture unit process;
Figure 24 shows an example texture coordinate unit and texture memory unit architecture;
Figure 25 shows an example texture memory color index mode lookup;
Figure 26 shows an example more detailed use of the texture memory to store color indexed textures;
Figure 27 shows an example color combiner operation;
Figure 28 shows an example alpha combiner operation;
Figure 29 shows an example alpha fix up operation;
Figure 30 shows an example of blending different types of primitives;
Figure 31 shows an example blender operation;
Figure 32 shows an example color pixel format;
Figure 33 shows an example depth (z) pixel format;
Figure 33A shows an example write enable generation process;
Figure 34 shows an example video interface architecture;
Figure 34A shows an example video interface operating sequence;
Figures 35A-35P show example video interface control registers;
Figure 36 shows an example main memory interface architecture;
Figures 37A-37H show example memory interface controller registers;
Figure 38 shows an example main processor interface architecture;
Figures 39A-39D show example main processor interface registers;
Figure 40 shows an example audio interface architecture;
Figures 41A-41F show example audio interface registers;
Figure 42 shows an example serial interface architecture;
Figures 43A-43D show example serial interface registers;
Figure 44 shows an example peripheral interface architecture; and
Figures 45A-45I show example peripheral interface control/status registers.

DETAILED DESCRIPTION OF A PRESENTLY PREFERRED EXAMPLE EMBODIMENT

[0009] Figure 1 shows an example embodiment video game system 50 in accordance with the present invention(s). Video game system 50 in this example includes a main unit 52, a video game storage device 54, and handhold controllers 56 (or other user input devices). In this example, main unit 52 connects to a conventional home color television set 58. Television set 58 displays 3D video game images on its television screen 60 and reproduces stereo sound through its loud speakers 62.

[0010] In this example, the video game storage device 54 is in the form of a replaceable memory cartridge insertable into a slot 64 on a top surface 66 of main unit 52. Video game storage device 54 can comprise, for example, a plastic housing 68 encasing a read only memory (ROM) chip 76. The read only memory 76 contains video game software in this example. When the video game storage device 54 is inserted into main unit slot 64, cartridge electrical contacts 74 mate with corresponding "edge connector" electrical contacts within the main unit. This action electrically connects the storage device's read only memory 76 to the electronics within main unit 52.

[0011] "Read only memory" chip 76 stores software instructions and other information pertaining to a particular video game. The read only memory chip 76 in one storage device 54 may, for example, contain instructions and other information for an adventure game. The read only memory chip 76 in another storage device 54 may contain instructions and information to play a driving or car race game. The read only memory chip 76 of still another storage device 54 may contain instructions and information for playing an educational game. To play one game as opposed to another, the user of video game system 50 simply plugs the appropriate storage device 54 into main unit slot 64-thereby connecting the storage device's read only memory chip 76 (and any other circuitry the storage device may contain) to the main unit 52. This enables the main unit 52 to access the information contained within read only memory 76, which information controls the main unit to play the appropriate video game by displaying images and reproducing sound on color television set 58 as specified under control of the video game software in the read only memory.

[0012] To play a video game using video game system 50, the user first connects main unit 52 to his or her color television set 58 by hooking a cable 78 between the two. Main unit 52 produces both "video" signals and "audio" signals for controlling color television set 58. The "video" signals are what controls the images displayed on the television screen 60, and the "audio" signals are played back as sound through television loudspeakers 62. Depending on the type of color television set 58, it may be necessary to use an additional unit called an "RF modulator" in line between main unit 52 and color television set 58. An "RF modulator" (not shown) converts the video and audio outputs of main unit 52 into a broadcast type television signal (e.g., on television channel 2 or 3) that can be received and processed

using the television set's internal "tuner."

[0013] The user also needs to connect main unit 52 to a power source. This power source may comprise a conventional AC adapter (not shown) that plugs into a standard home electrical wall socket and converts the house current into a lower DC voltage signal suitable for powering main unit 52.

[0014] The user may then connect hand controllers 56a, 56b to corresponding connectors 80 on main unit front panel 82. Controllers 56 may take a variety of forms. In this example, the controllers 56 shown each include various push buttons 84 and a directional switch or other control 86. The directional switch 88 can be used, for example, to specify the direction (up, down, left or right) that a character displayed on television screen 60 should move and/or to specify a point of view in a 3D world. Other possibilities include, for example, joysticks, mice pointer controls and other conventional user input devices. In this example, up to four controllers 56 can be connected to main unit 52 to allow 4-player games.

[0015] The user then selects a storage device 54 containing the video game he or she wants to play, and inserts that storage device into main unit slot 64 (thereby electrically connecting read only memory 76 to the main unit electronics via a printed circuit board 70 and associated edge contacts 74). The user may then operate a power switch 88 to turn on the video game system 50. This causes main unit 52 to begin playing the video game based on the software stored in read only memory 54. He or she may operate controllers 86 to provide inputs to main unit 52 and thus affect the video game play. For example, depressing one of push buttons 84 may cause the game to start. As mentioned before, moving directional switches 86 can cause animated characters to move on the television screen 60 in different directions or can change the user's point of view in a 3D world. Depending upon the particular video game stored within the storage device 54, these various controls 84, 86 on the controller 56 can perform different functions at different times. If the user wants to restart game play, he or she can press a reset button 90.

EXAMPLE 3D SCREEN EFFECTS

[0016] System 50 is capable of processing, interactively in real time, a digital representation or model of a three-dimensional world to display the world (or portions of it) from any arbitrary viewpoint within the world. For example, system 50 can interactively change the viewpoint in response to real time inputs from game controllers 86. This can permit, for example, the game player to see the world through the eyes of a "virtual person" who moves through the world, and looks and goes wherever the game player commands him or her to go. This capability of displaying quality 3D images interactively in real time can create very realistic and exciting game play.

[0017] Figures 1A-1F show just one example of some three-dimensional screen effects that system 50 can generate on the screen of color television set 58. Figures 1A-1F are in black and white because patents cannot print in color, but system 50 can display these different screens in brilliant color on the color television set. Moreover, system 50 can create these images very rapidly (e.g., seconds or tenths of seconds) in real time response to operation of game controllers 86.

[0018] Each of Figures 1A-1F was generated using a three-dimensional model of a "world" that represents a castle on a hilltop. This model is made up of geometric shapes (i.e., polygons) and "textures" (digitally stored pictures) that are "mapped" onto the surfaces defined by the geometric shapes. System 50 sizes, rotates and moves these geometric shapes appropriately, "projects" them, and puts them all together to provide a realistic image of the three-dimensional world from any arbitrary viewpoint. System 50 can do this interactively in real time response to a person's operation of game controllers 86.

[0019] Figures 1A-1C and 1F show aerial views of the castle from four different viewpoints. Notice that each of the views is in perspective. System 50 can generate these views (and views in between) interactively in a matter of seconds with little or no discernible delay so it appears as if the video game player is actually flying over the castle.

[0020] Figures 1D and 1E show views from the ground looking up at or near the castle main gate. System 50 can generate these views interactively in real time response to game controller inputs commanding the viewpoint to "land" in front of the castle, and commanding the "virtual viewer" (i.e., the imaginary person moving through the 3-D world through whose eyes the scenes are displayed) to face in different directions. Figure 1D shows an example of "texture mapping" in which a texture (picture) of a brick wall is mapped onto the castle walls to create a very realistic image.

Overall Video Game System Electronics

[0021] Figure 2 shows that the principal electronics within main unit 52 includes a main processor 100, a coprocessor 200, and main memory 300. Main processor 100 is a computer that runs the video game program provided by storage device 54 based on inputs provided by controllers 56. Coprocessor 200 generates images and sound based on instructions and commands it gets from main processor 100. Main memory 300 is a fast memory that stores the information main processor 100 and coprocessor 200 need to work, and is shared between the main processor and the coprocessor. In this example, all accesses to main memory 300 are through coprocessor 200.

[0022] In this example, the main processor 100 accesses the video game program through coprocessor 200 over a communication path 102 between the main processor and the coprocessor 200. Main processor 100 can read from storage device 54 via another communication path 104 between the coprocessor and the video game storage device. The main processor 100 can copy the video game program from the video game storage device 54 into main memory 300 over path 106, and can then access the video game program in main memory 300 via coprocessor 200 and paths 102, 106.

[0023] Main processor 100 generates, from time to time, lists of commands that tell the coprocessor 200 what to do. Coprocessor 200 in this example comprises a special purpose high performance application-specific integrated circuit (ASIC) having an internal design that is optimized for rapidly processing 3-D graphics and digital audio. In response to commands provided by main processor 100 over path 102, coprocessor 200 generates video and audio for application to color television set 58. The coprocessor 200 uses graphics, audio and other data stored within main memory 300 and/or video game storage device 54 to generate images and sound.

[0024] Figure 2 shows that coprocessor 200 in this example includes a signal processor 400 and a display processor 500. Signal processor 400 is an embedded programmable microcontroller that performs graphics geometry processing and audio digital signal processing under control of a "microcode" computer program supplied by video game storage device 54. Display processor 500 is a high speed state machine that renders graphics primitives, thereby creating images for display on television 58. The signal processor 400 and display processor 500 work independently, but the signal processor can supervise the display processor by sending graphics commands to it. Both signal processor 400 and display processor 500 can be controlled directly by main processor 100. The following are examples of functions and operations the signal processor 400 and display processor 500 can perform:

SIGNAL PROCESSOR

[0025]

- Matrix control
- 3D transformations
- Lighting
- Clipping, perspective and viewport application
- Display processor command generation

DISPLAY PROCESSOR

[0026]

- Rasterization
- Texture coordinate generation
- Texture application and filtering
- Color combining
- Blending
- Fogging
- Antialiasing
- Frame buffer and frame buffer control

[0027] Figure 3 shows the main processes performed by the main processor 100, coprocessor 200 and main memory 300 in this example system 50. The main processor 100 receives inputs from the game controllers 56 and executes the video game program provided by storage device 54 to provide game processing (block 120). It provides animation, and assembles graphics and sound commands for use by coprocessor 200. The graphics and sound commands generated by main processor 100 are processed by blocks 122, 124 and 126—each of which is performed by coprocessor 200. In this example, the coprocessor signal processor 400 performs 3D geometry transformation and lighting processing (block 122) to generate graphics display commands for display processor 500. Display processor 500 "draws" graphics primitives (e.g., lines, triangles and rectangles) to create an image for display on color TV 58. Display processor 500 performs this "drawing" or rendering function by "rasterizing" each primitive and applying a texture to it if desired (block 126). It does this very rapidly-e.g., on the order of many millions of "pixels" (color television picture elements) a second. Display processor 500 writes its image output into a frame buffer in main memory 300 (block 128). This frame buffer stores a digital representation of the image to be displayed on the television screen 60. Additional circuitry within coprocessor 200 reads the information from the frame buffer and outputs it to television 58 for display (block 130).

[0028] Signal processor 400 also processes sound commands received from main processor 100 using digital audio

signal processing techniques (block 124). Signal processor 400 writes its digital audio output into a sound buffer in main memory 300. The main memory temporarily "buffers" (i.e., stores) the sound output (block 132). Other circuitry in coprocessor 200 reads this buffered sound data from main memory 300 and converts it into electrical audio signals (stereo left and right channels) for application to and reproduction by television speakers 62a, 62b (block 134).

[0029] Television 58 displays 30 or 60 new images a second. This "frame rate" fools the human eye into seeing continuous motion, allowing main unit 52 to create animation effects on television screen 60 by changing the image slightly from one frame to the next. To keep up with this television frame rate, coprocessor 200 must create a new image every 1/30 or 1/60 of a second. Coprocessor 200 must also be able to produce a stream of continuous sound to go along with the animation effects on screen 60.

Overall System Operation

[0030] Figure 4 shows the overall operation of system 50 in more detail, and Figure 4A shows overall steps performed by the system to generate graphics. In this example, main processor 100 reads a video game program 108 stored in main memory 300 (generally, this video game program will have originated in video game storage device 54 and have been copied from the video game storage device into the main memory). In response to executing this video game program 108 (and in response to inputs from game controllers 56), main processor 100 creates (or reads from storage device 58) a list 110 of commands for coprocessor 200 (Figure 4A, block 120a). This list 110, in general, includes two kinds of commands:

(1) graphics commands
(2) audio commands.

Graphics commands tell coprocessor 200 what images to generate on TV screen 60. Audio commands tell sound coprocessor 200 what sounds it should generate for reproduction on TV loudspeakers 62.

[0031] The list of graphics commands is called a "display list" because it controls the images coprocessor 200 displays on the TV screen 60. The list of audio commands is called a "play list" because it controls the sounds that are played over loudspeaker 62. Generally, main processor 100 specifies both a new display list and a new play list for each video "frame" time of color television set 58.

[0032] In this example, main processor 100 provides its display/play list 110 to coprocessor 200 by storing it into main memory 300 and then telling the coprocessor where to find it (Figure 4A, block 120c). Main processor 100 also makes sure the main memory 300 contains a graphics and audio database 112 that includes all of the data coprocessor 200 will need to generate the graphics and sound requested in the display/play list 110. Some or all of this graphics and audio database 112 can come from storage device 54. The display/play list 110 specifies which portions of graphics and audio database 112 the coprocessor 200 should use. Main processor 100 also is responsible for making sure that signal processor 400 has loaded "microcode"—i.e., a computer program that tells the signal processor what to do.

[0033] Signal processor 400 reads the display/play list 110 from main memory 100 (Figure 4A, block 122a) and processes this list—accessing additional data within the graphics and audio database 112 as needed (Figure 4A, block 122b). Signal processor 400 generates two main outputs: graphics display commands 112 for further processing by display processor 500 (Figure 4A, block 122c); and audio output data 114 for temporary storage within main memory 300. Signal processor 400 processes the audio data in much less than the time it takes to play the audio through loudspeakers 62. Another part of the coprocessor 200 called an "audio interface" (not shown) subsequently reads the buffered audio data and outputs it in real time for reproduction by television loudspeakers 62.

[0034] The signal processor 400 can provide the graphics display commands 112 directly to display processor 500 over a path internal to coprocessor 200, or it may write those graphics display commands into main memory 300 for retrieval by the display processor (not shown). These graphics display commands 112 command display processor 500 to draw ("render") specified geometric shapes with specified characteristics (Figure 4a, block 126a). For example, display processor 500 can draw lines, triangles or rectangles (polygons) based on these graphics display commands 112, and may fill triangles and rectangles with particular colors and/or textures 116 (e.g., images of leaves of a tree or bricks of a brick wall)—all as specified by the graphics display commands 112. Main processor 100 stores the texture images 116 into main memory 300 for access by display processor 500. It is also possible for main processor 100 to write graphics display commands 112 directly into main memory 300 for retrieval by display processor 500 to directly command the display proccssor.

[0035] Display processor 500 generates, as its output, a digitized representation of the image that is to appear on television screen 60 (Figure 4A, block 126b). This digitized image, sometimes called a "bit map," is stored within a frame buffer 118 residing in main memory 300. Display processor 500 can also store and use a depth (Z) buffer 118b in main memory 300 to store depth information for the image. Another part of coprocessor 200 called the "video interface" (not shown) reads the frame buffer 118 and converts its contents into video signals for application to color tele-

vision set 58 (Figure 4a, block 127). Typically, frame buffer 118 is "double buffered," meaning that coprocessor 200 can be writing the "next" image into half of the frame buffer while the video interface is reading out the other half.

[0036] The various steps shown in Figure 4A and described above are "pipelined" in this example. "Pipelining" means that different operations are performed concurrently for different stages in the graphics generation process. A simple analogy is the way most people do laundry. A non-pipeline mode of doing laundry would involve completing all relevant tasks (washing, drying, ironing/folding, and putting away) for one load of laundry before beginning the next load. To save time, people with multiple loads of laundry "pipeline" the laundry process by performing washing, drying, ironing/folding and putting away operations concurrently for different loads of laundry.

[0037] Similarly, the operations performed by main processor 100, signal processor 400, display processor 500 and video interface 210 are "pipelined" in this example. For example, main processor 100 in this example can be assembling a display list two video frames ahead while signal processor 400 and display processor 500 are processing data for one video frame ahead and video interface 210 is processing data for the current video frame in progress. As is explained below, the detailed graphics rendering steps performed by display processor 500 in block 126a are also pipelined to maximize speed performance.

More Detailed System Architecture

[0038] Figure 5 shows a more detailed architecture of video game system 50. This diagram shows video game main unit 52 including, in addition to main processor 100, coprocessor 200 and main memory 300, additional components such as a clock generator 136, a serial peripheral interface 138, an audio digital-to-analog converter (DAC) 140, an audio amplifier/mixer 142, a video digital-to-analog converter 144, and a video encoder 146.

[0039] In this example, the clock generator 136 (which may be controlled by a crystal 148) produces timing signals to time and synchronize the other components of main unit 52. Different main unit components require different clocking frequencies, and dock generator 136 provides suitable such clock frequency outputs (or frequencies from which suitable clock frequencies can be derived such as by dividing). A timing block 216 within coprocessor 200 receives clocking signals from clock generator 136 and distributes them (after appropriate dividing as necessary) to the various other circuits within the coprocessor.

[0040] In this example, the game controllers 58 are not connected directly to main processor 100, but instead are connected to main unit 52 through serial peripheral interface 138. Serial peripheral interface 138 demultiplexes serial data signals incoming from up to four (or five) game controllers 56 (or other serial peripheral devices) and provides this data in a predetermined format to main processor 100 via coprocessor 200. Serial peripheral interface 138 is bidirectional in this example, i.e., it is capable of transmitting serial information specified by main processor 100 in addition to receiving serial information.

[0041] Serial peripheral interface 138 in this example also includes a "boot ROM" read only memory 150 that stores a small amount of initial program load (IPL) code. This IPL code stored within boot ROM 150 is executed by main processor 100 at time of startup and/or reset to allow the main processor to begin executing game program instructions 108a within storage device 54 (see Figure 5A, blocks 160a, 160b). The initial game program instructions 108a may, in turn, control main processor 100 to initialize the drivers and controllers it needs to access main memory 300 (see Figure 5A, blocks 160c, 160d) and to copy the video game program and data into the faster main memory 300 for execution and use by main processor 100 and coprocessor 200 (see Figure 5A, blocks 160e, 160f, 160g).

[0042] Also in this examplo, serial peripheral interface 138 includes a security processor (e.g., a small microprocessor) that communicates with an associated security processor 152 (e,g., another small microprocessor) within storage device 54 (see Figure 5). This pair of security processors (one in the storage device 54, the other in the main unit 52) perform an authentication function to ensure that only authorized storage devices may be used with video game main unit 52. See U.S. Patent No. 4,799,635. In this example, the security processor within serial peripheral interface 138 may process data received from game controllers 56 under software control in addition to performing a security function under software control.

[0043] Figure 5 shows a connector 154 within video game main unit 52. This connector 154 connects to the electrical contacts 74 at the edge of storage device printed circuit board 70 in this example (see Figure 1). Thus, connector 154 electrically connects coprocessor 200 to storage device ROM 76. Additionally, connector 154 connects the storage device security processor 152 to the main unit's serial peripheral interface 138. Although connector 154 in the particular example is used primarily to read data and instructions from a non-writable read only memory 76, system 52 is designed so that the connector is bidirectional, i.e., the main unit can send information to the storage device 54 in addition to reading information from it.

[0044] Figure 5 also shows that the audio and video outputs of coprocessor 200 are processed by some electronics outside of the coprocessor before being sent to television set 58. In particular, in this example coprocessor 200 outputs its audio and video information in digital form, but conventional home color television sets 58 generally require analog audio and video signals. Therefore, the digital outputs of coprocessor 200 are converted into analog form-a function

performed for the audio information by DAC 140 and for the video information by VDAC 144. The analog audio output of DAC 140 is amplified by an audio amplifier 142 that may also mix audio signals generated externally of main unit 52 and supplied through connector 154. The analog video output of VDAC 144 is provided to video encoder 146, which may, for example, convert "RGB" input signals to composite video outputs. The amplified stereo audio output of amplifier 142 and the composite video output of video encoder 146 are provided to home color television set 58 through a connector not shown.

[0045] As shown in Figure 5, main memory 300 stores the video game program in the form of CPU instructions 108b. These CPU instructions 108b are typically copied from storage device 54. Although CPU 100 in this example is capable of executing instructions directly out of storage device ROM 76, the amount of time required to access each instruction from the ROM is much greater than the time required to access instructions from main memory 300. Therefore, main processor 100 typically copies the game program/data 108a from ROM 76 into main memory 300 on an as-needed basis in blocks, and accesses the main memory in order to actually execute the instructions (see Figure 5A, blocks 160c, 160f). The main processor 100 preferably includes an internal cache memory to further decrease instruction access time.

[0046] Figure 5 shows that storage device 54 also stores a database of graphics and sound data 112a needed to provide the graphics and sound of the particular video game. Main processor 100 reads the graphics and sound data 112a from storage device 54 on an as-needed basis and stores it into main memory 300 in the form of texture data 116, sound data 112b and graphics data 112c. In this example, display processor 500 includes an internal texture memory 502 into which the texture data 116 is copied on an as-needed basis for use by the display processor.

[0047] Storage device 54 also stores coprocessor microcode 156. As described above, in this example signal processor 400 executes a computer program to perform its various graphics and audio functions. This computer program or "microcode," is provided by storage device 54. Because the microcode 156 is provided by storage device 54, different storage devices can provide different microcodes-thereby tailoring the particular functions provided by coprocessor 200 under software control. Typically, main processor 100 copies a part of the microcode 156 into main memory 300 whenever it starts the signal processor, and the signal processor 400 then accesses other parts of the microcode on an as-needed basis. The signal processor 400 executes the microcode out of an instruction memory 402 within the signal processor 400. Because the SP microcode 156 may be too large to fit into the signal processor's internal instruction memory 402 all at once, different microcode portions may need to be loaded from main memory 300 into the instruction memory 402 to allow signal processor 400 to perform different tasks. For example, one part of the SP microcode 156 may be loaded into signal processor 400 for graphics processing, and another part of microcode may be loaded into the signal processor for audio processing. In this example, the signal processor microcode RAM 402 (and an additional signal processor data memory RAM not shown in Figure 5) is mapped into the address space of main processor 100 so the main processor can directly access the RAM contents under software control through load and store instructions.

Main Processor 100

[0048] Main processor 100 in this example is a MIPS R4300 RISC microprocessor designed by MIPS Technologies, Inc., Mountain View, California. This R4300 processor includes an execution unit with a 64-bit register file for integer and floating-point operations, a 16 KB Instruction Cache, a 8 KB Write Back Data Cache, and a 32-entry TLB for virtual-to-physical address calculation. The main processor 100 executes CPU instructions (e.g., a video game program) 108 in kernel mode with 32-bit addresses. 64-bit integer operations are available in this mode, but 32-bit calling conventions are preferable to maximize performance. For more information on main processor 100, see, for example, Heinrich, MIPS Microprocessor R4000 User's Manual (MIPS Technologies, Inc., 1994, Second Ed.).

[0049] Main processor 100 communicates with coprocessor 200 over bus 102, which in this example comprises a bi-directional 32-bit SysAD multiplexed address/data bus, a bi-directional 5-bit wide SysCMD bus, and additional control and timing lines. See chapter 12 et seq. of the above-mentioned Heinrich manual.

[0050] The conventional R4300 main processor supports six hardware interrupts, one internal (timer) interrupt, two software interrupts, and one non-maskable interrupt (NMI). In this example, three of the six hardware interrupt inputs (INT0, INT1 and INT2) and the non-maskable interrupt (NMI) input allow other portions of system 50 to interrupt the main processor. Specifically, main processor INT0 is connected to allow coprocessor 200 to interrupt the main processor, main processor interrupt INT1 is connected to allow storage device 54 to interrupt the main processor, and main processor interrupts INT2 and NMI are connected to allow the serial peripheral interface 138 to interrupt the main processor. Any time the processor is interrupted, it looks at an internal interrupt register to determine the cause of the interrupt and then may respond in an appropriate manner (e.g., to read a status register or perform other appropriate action). All but the NMI interrupt input from serial peripheral interface 138 are maskable (i.e., the main processor 100 can selectively enable and disable them under software control).

[0051] Main processor 100 reads data from and writes data to the rest of system 50 via the CPU-to-coprocessor bus

102. The coprocessor 200 performs a memory mapping function, allowing the main processor 100 to address main memory 300, the storage device cartridge ROM 76, the "boot ROM" 150 within serial peripheral interface 138 (and other parts of the serial peripheral interface), various parts of coprocessor 200 (including signal processor RAM 402), and other parts of system 50.

[0052] In the example, the operations performed by main processor 100 are completely dependent on videogame program 108. In this example, all "system" software is supplied by the storage device 58 to provide maximum flexibility. Different video games (or other applications) may run more efficiently with different kinds of high level software. Therefore, main unit 52 in this example does not provide any standard software libraries -- or any software at all for that matter -- since such libraries could limit flexibility. Instead, all software in this example is supplied by storage device 54.

[0053] Developers of video game software 108 may wish to employ advanced software architecture such as, for example, device drivers, schedulers and thread libraries to manage the various resources within system 50. Since main processor 100 is a state-of-the-art RISC processor/computer, it is appropriate to use such software architecture/constructs and to implement video game program 108 in a high level software environment.

[0054] An example system "memory map" of the main processor 100 address space is shown in Figure 5B. As shown in this Figure 5B, main memory 300 is divided into two banks (bank 0 and bank 1) in this example. In addition, certain configuration registers 307 within the main memory 300 are mapped into the main processor address space, as are registers within coprocessor 200. Main processor 100 in this example can control each of the various coprocessor subblocks by writing, under control of video game program 108, into control registers associated with each coprocessor 200 sub-block.

[0055] As shown in Figure 5B, storage device 54 address space is divided into two "domains" (for two different devices, for example). These "domains" are mapped into several parts of the main processor 100 address space. Various parts of the serial peripheral interface 138 (i.e., PIF boot ROM 150, a PIF buffer RAM, and a PIF status register) are also mapped into the main processor 100 address space.

Unified Main Memory 300

[0056] Main memory 300 in this example comprises a RDRAM dynamic random access memory available from Rambus Inc. of Mountain View, California. In this example, main memory 300 is expandable to provide up to 8 megabytes of storage, although main unit 52 may be shipped with less RAM (e.g., 2 or 3 MB) to decrease cost

[0057] Main memory 300 provides stotage for the entire system 50 in this example. It provides a single address space (see Figure 5B above) for storing all significant data structures, including for example (as shown in Figure 5):

- Main processor instructions 108
- Signal processor microcode 156
- Display list graphic commands 110a
- Play list audio commands 110b
- Texture maps 116 and other graphics data 112c
- Color image frame buffer 118a
- Depth (z) buffer 118b
- sound data 112b
- Audio output buffer 114
- Main processor working values
- Coprocessor working values
- Data communicated between various parts of the system.

Advantages and disadvantages in using single address space memory architectures for raster scan display systems are known (see, for example, Foley et al, Computer Graphics: Principles and Practice at 177-178 (2d Ed. Addison-Weslcy 1990). Many video game (and other graphics) system architects in the past rejected a single address space architecture in favor of using dedicated video RAM devices for graphics data and using other types of memory devices for other types of data. However, a unified main memory 300 provides a number of advantages in this particular example of a video game system 50. For example:

[0058] Data communications between system elements is simplified. Once data is stored in main memory 300, there is little or no additional overhead in communicating the data to another part of the system. The overhead of transferring data between different pans of the system is thus minimized. For example, since the main processor 100 and each sub-block within the coprocessor 200 can each access system main memory 300, the main memory used by all system elements for data structure storage can also be used as a general purpose communication channel/data buffer between elements.

[0059] For example, display lists 110 main processor 100 stores within main memory 300 can be directly accessed

by signal processor 400. Similarly, display commands the main processor (and/or the signal processor) stores within the main memory can be directly accessed by display processor 500. The main processor 100 working data.(which can automatically be written into the main memory 300 via a "cache flush") is immediately available to all other parts of the system.

[0060] The unified memory provides memory allocation flexibility. Main memory 300 locations look alike, and therefore each location can be used for storing any type of data structure. All main memory 300 allocation decisions are left to the application programmer. This provides great flexibility in terms of data structure sizes and memory usage. Data structures can be stored anywhere in main memory 300, and each location in memory 300 can be allocated however the application programmer specifies.

[0061] For example, one video game programmer might provide a large frame buffer for high resolution images and/or image scrolling and panning, while another programmer may decide to use a smaller frame buffer so as to free up memory space for other data structures (e.g., textures or audio data). One application may devote more of main memory 300 storage for audio data structures and less to graphics data, while another application may allocate most of the storage for graphics related data. The same video game program 108 can dynamically shift memory allocation from one part of game play to another (e.g., at the time the game changes levels) to accomplish different effects. Application flexibility is not limited by any fixed or hardwired memory allocation.

[0062] The Unified RAM architecture supports flexible data structure sharing and usage. Since all significant data structures are stored within common main memory 300, they can all be accessed by main processor 100 and other system elements. There is no hardware distinction between display images and source images. For example, main processor 100 can, if desired, directly access individual pixels within frame buffer 118. The scan conversion output of display processor 500 can be used as a texture for a texture mapping process. Image source data and scan converted image data can be interchanged and/or combined to accomplish special effects such as, for example, warping scan-converted images into the viewpoint.

[0063] The shortcomings of a unified memory architecture (e.g., contention for access to the main memory 300 by different parts of the system) have been minimized through careful system design. Even though main memory 300 is accessed over a single narrow (9-bit-wide) bus 106 in this example, acceptable bandwidth has been provided by making the bus very fast (e.g., on the order of 240 MHz). Data caches are provided throughout the system 50 to make each subcomponent more tolerant to waiting for main memory 300 to become available.

Coprocessor 200

[0064] Figure 5 shows that coprocessor 200 includes several components in addition to signal processor 400 and display processor 500, namely:

- CPU interface 202,
- a serial interface 204,
- a parallel peripheral interface 206,
- an audio interface 208,
- a video interface 210,
- a main memory DRAM controller/interface 212,
- a main internal bus 214 and
- a timing block 216.

In this example, main bus 214 allows each of the various main components within coprocessor 200 to communicate with one another.

[0065] Figure 6, a more detailed diagram of coprocessor 200, shows that the coprocessor is a collection of processors, memory interfaces and control logic all active at the same time and operating in parallel. The following briefly describes the overall functions provided by each of these other sub-blocks of coprocessor 200:

- Signal processor 400 is a microcoded engine that executes audio and graphics tasks.
- Display processor 500 is a graphics display pipeline that renders into frame buffer 118.
- Coprocessor serial interface 204 provides an interface between the serial peripheral interface 128 and coprocessor 200 in this example.
- Coprocessor parallel peripheral interface 206 interfaces with the storage device 54 or other parallel devices connected to connector 154.
- Audio interface 208 reads information from audio buffer 114 within main memory 300 and outputs it to audio DAC 140.
- Coprocessor video interface 210 reads information from frame buffer 118a within main memory 300 and outputs

it to video DAC 144.

- The CPU interface 202 is the gateway between main processor 100, coprocessor 200 and the rest of system 50.
- DRAM controller/interface 212 is the gateway through which coprocessor 200 (and main processor 100) accesses main memory 300. Memory interface 212 provides access to main memory 300 for main processor 100, signal processor 400, display processor 500, video interface 210, audio interface 208, and serial and parallel interfaces 204, 206.

Each of these various processors and interfaces may be active at the same time.

[0066] Signal processor 400 in this example includes the instruction memory 402 discussed above, a data memory 404, a scalar processing unit 410 and a vector processing unit 420. Instruction memory 402 stores microcode for execution by scalar unit 410 and/or vector unit 420. Data memory 404 stores input data, work data and output data for the scalar unit 410 and for the vector unit 420. Signal processor 400 can execute instructions only out of instruction memory 402 in this example, but has access to main memory 300 via direct memory accessing (DMA) techniques.

[0067] In this example, scalar unit 410 is a general purpose integer processor that executes a subset of the MIPS R4000 instruction set. It is used to perform general purpose operations specified by microcode within instruction memory 402. Vector unit 420 comprises eight 16-bit calculating elements capable of performing numerical calculations in parallel. Vector unit 420 is especially suited for graphics matrix calculations and certain kinds of digital audio signal processing operations.

[0068] Display processor 500 in this example is a graphics display pipelined engine that renders a digital representation of a display image. It operates based on graphics display commands generated by the signal processor 400 and/or main processor 100. Display processor 500 includes, in addition to texture memory 502, a rasterizer 504, a texture unit 506, a color combiner 508, a blender 510 and a memory interface 512. Briefly, rasterizer 504 rasterizes polygon (e.g., triangle, and rectangle) geometric primitives to determine which pixels on the display screen 60 are within these primitives. The texture unit can apply texture maps stored within texture memory 502 onto textured areas defined by primitive edge equations solved by rasterizer 504. The color combiner 508 combines and interpolates between the texture color and a color associated with the graphic primitive. Blender 510 blends the resulting pixels with pixels in frame buffer 118 (the pixels in the frame buffer are accessed via memory interface 512) and is also involved in performing Z buffering (i.e., for hidden surface removal and anti-aliasing operations). Memory interface 512 performs read, modify and write operations for the individual pixels, and also has special modes for loading/copying texture memory 502, filling rectangles (fast clears), and copying multiple pixels from the texture memory 502 into the frame buffer 118. Memory interface 512 has one or more pixel caches to reduce the number of accesses to main memory 300.

[0069] Display processor 500 includes circuitry 514 that stores the state of the display processor. This state information is used by the rest of display processor 500 to, for example, select rendering modes and to ensure that all previous rendering effected by a mode change occurs before the mode change is implemented.

[0070] The command list for display processor 500 usually comes directly from signal processor 400 over a private "X bus" 218 that connects the signal processor to the display processor. More specifically, X-bus 218 in this example is used to transfer graphics display commands from the signal processor data memory 404 into a command buffer (not shown in Figure 6) within display processor 500 for processing by the display processor. However, in this example it is also possible for signal processor 400 and/or main processor 100 to feed graphics display commands to display processor 500 via main memory 300.

[0071] Display processor 500 accesses main memory 300 using physical addresses to load its internal texture memory 502, read frame buffer 118 for blending, read the Z buffer 118B for depth comparison, to write to the Z-buffer and the frame buffer, and to read any graphics display commands stored in the main memory.

Coprocessor Internal Bus Architecture

[0072] Figure 6A is a more detailed diagram showing an example coprocessor bus 214 arrangement, which in this example comprises a 32-bit address ("C") bus 214C and a 64-bit data ("D") bus 214D. These busses 214C, 214D are connected to each of signal processor 400, display processor 500, CPU interface 202, audio interface 208, video interface 210, serial interface 204, parallel peripheral interface 206, and main memory (RAM) interface 212. As shown in Figure 6A, main processor 100 and each of the sub-blocks of coprocessor 200 communicates with main memory 300 via internal coprocessor busses 214C, 214D, and main memory interface/controller 212a/212b.

[0073] In this example, main memory interface/controller 212a, 212b converts main memory addresses asserted on coprocessor address bus 214C into 9-bit-wide format for communication over the 9-bit-wide main memory multiplexed address/data bus 106, and also converts between the main memory bus 106 9-bit-wide data format and the coprocessor data bus 214D 64-bit wide data format. In this example, the DRAM controller/interface 212 includes, as a part thereof, a conventional RAM controller 212b (sec Figure 6A) provided by Rambus Inc. The use of a 9-bit-wide main memory bus 106 reduces the chip pin count of coprocessor 200.

[0074] In this example, each of the coprocessor 200 sub-blocks shown has an associated direct memory access (DMA) circuit that allows it to independently address and access main memory 300. For example, signal processor DMA circuit 454, display processor DMA circuit 518, audio interface DMA circuit 1200, video interface DMA circuit 900, serial interface DMA circuit 1300, and parallel peripheral interface DMA circuit 1400 each allow their associated coprocessor sub-block to generate addresses on coprocessor address bus 214C and to communicate data via coprocessor data bus 214D (additionally, display processor 500 has a further memory interface block 512 for access to the main memory frame buffer 118 and texture data 116).

[0075] Although each of the coprocessor 200 sub-blocks can independently access main memory 300, they all share common busses 214C, 214D in this example -- and only one of the subblocks can use these shared busses at a time. Accordingly, coprocessor 200 has been designed to make most efficient use of the shared busses 214. For example, the coprocessor 200 sub-blocks may buffer or "cache" information to minimize the frequency of different bus accesses by the same sub-block and to make the subblocks more tolerant of temporary bus unavailability. A private bus 218 allows signal processor 400 to communicate with display processor 500 without having to wait for main bus 214 to become available.

[0076] Also as shown in Figure 6A, each of the sub-blocks of coprocessor 200 includes control/status registers that can be accessed by main processor 100 via CPU interface 202. For example, signal processor registers 407, display processor registers 507, audio interface registers 1207, video interface registers 907, serial interface registers 1307, parallel peripheral interface registers 206, RAM interface registers 1007a, and RAM controller registers 1007b are each mapped into the main processor 100 address space. The main processor 100 can read from and/or write to these various registers under control of game program 108 to directly control the operation of sub-blocks within coprocessor 200.

Signal Processor 400

[0077] Figure 7 shows the architecture of signal processor 400 of this example in more detail. As explained above, signal processor 400 includes a scalar unit 410, a vector unit 420, an instruction memory 402 and a data memory 404. In this example, scalar unit 410 is a 32-bit integer processor that executes a sub-set of the MIPS 4000 instruction set. Vector unit 420 (which is defined as a "CP1" coprocessor of scalar unit 410 under the MIPS 4000 architecture) performs integer calculations (e.g., multiplications, additions, subtractions and multiply/accumulates) on eight 16-bit sets of values in parallel.

[0078] Vector unit 420 can perform the same operation on eight pairs of 16-bit operands in parallel simultaneously. This makes signal processor 400 especially suited for "sum of products" calculations such as those found in matrix multiplications, texture resampling, and audio digital signal processing such as, for example, digital audio synthesis and spatial and frequency filtering.

[0079] Signal processor 400 uses a RISC (reduced instruction set computer) architecture to provide high performance machine control based on instructions residing in the instruction memory 402. In this example, execution unit includes a program counter 432 that is used to address instruction memory 402 over path 434. This program counter 432 can access only the 4 kilobyte instruction space within instruction memory 402 in this example--requiring that all instructions to be executed by the signal processor first be placed into the instruction memory. Execution unit 430 generates output control signals 436 based on the particular instructions currently being executed. These output control signals 436 control all other parts of signal processor 400, and arc sequenced to manage pipelined instruction processing. Scalar unit 410 and vector unit 420 are controlled by these control signals 436. For example, scalar unit 410 may address data memory 404 via path 438 to read data from and/or write data into the data memory using load/store block 440. Data path 414 may perform tests based on results of calculations and provide resulting condition outputs to execution unit 430 via path 442. This execution unit 430 may use these condition outputs to perform a conditional branch or jump, loading program counter 432 with the appropriate (next) address into instruction memory 402. Because scalar processor 410 has these more general capabilities, it is used in this example for general purpose functions such as, for example, control flow, address calculation and the like-in addition to providing 32-bit integer calculations.

[0080] Execution unit 430 executes intermediate, jump and register instruction formats in accordance with the standard MIPS R4000 instruction set. Figure 7A shows an example of a register instruction format 450 and how signal processor 400 uses that register instruction format to access three 128-bit wide words 452 within data memory 404. Register instruction format 450 may include a 6-bit operation code field 450(a), a 5-bit source register specifier 450(b), a 5-bit target (source/destination) register specifier 450(c), a 5-bit destination register specifier 450(d), and a parameter field 450(e). The parameter field 450(e) may specify shift amounts and/or functions, and together with operation code 450(a) defines the operation to be performed. Each of fields 450(b), 450(c) and 450(d) specifies a location within data memory 404--and thus each designates 128-bit word.

[0081] As shown in Figure 7B, vector unit 420 treats each of these 128-bit words as a concatenated sequence of eight 16-bit values, and operates on each of the 16-bit values in parallel. The operations of vector unit 420 are invoked

by instructions within the CP1 type instructions typically reserved for floating point operations in the MIPS R4000 instruction set (signal processor 400 has no floating point unit in this example).

[0082] Scalar unit 410 includes a register file 412 comprising 32 registers, each register being 32 bits wide. Scalar unit also includes a data path 414 comprising adders, shifters, and other logic required to execute integer calculations and other operations. Register file 412 is similar to the general purpose register file defined by the MIPS R4000 architecture, and accepts instructions in R4000 format. Data path 414 includes an integer multiplier/divider, and operates in conjunction with an execution unit 430 that receives 64-bit wide instructions from instruction memory 402.

[0083] Vector unit 420 includes eight sets of register files 422(0)-422(7) and eight sets of corresponding data paths 423 (0)-423(7). Data paths 423 each include a 16-bit multiplier, a 16-bit adder and a 48-bit accumulator (48 bit accumulation accommodates audio filters with a large number of taps, and also accommodates partial products wherein a series of 16-bit multiplies and sums is used to obtain a 32-bit result for certain graphics calculations requiring more than 16-bit precision). Each of register files 422 comprises 32 registers each of which are 32-bits wide. A 128 bit wide data path 444 connects vector unit 420 to load/store block 440, and another 128 bit wide data path 446 connects the load/store block 440 to data memory 404. Data memory 404 stores 4096 (4KB) words, each word being 128 bits wide. When a word in data memory 404 is retrieved for use by vector unit 420, it is sliced into eight 16-bit segments, with each segment being sent to a different register file 422 within vector unit 420 (see Figure 7B). Figure 7C shows an example add operation performed by vector unit 420. When vector unit 420 writes to a destination addressed within data memory 404, each of register files 422 contributes 16-bits which are combined into a 128 bit word before being written into the data memory (see Figure 7A). Alternatively, load/store block 440 includes a steering multiplexer arrangement (not shown) that can steer 16-bit sub-words within the data memory 128-bit word to/from different vector unit register files 422 -- with the particular sub-word and the particular vector unit register file being selectable based on instructions from instruction memory 402. Similarly, load/store block 440 includes a further steering multiplexer arrangement (not shown) that can steer different sized data units (e.g., bytes, 16-bit half-words, or 32-bit words) between data memory 408 and scalar unit 410 -- with the particular data unit and size being specified by instructions within instruction memory 402. See, for example, description of Load and Store "Byte", "Halfword", "Word", "Word Left" and "Word Right" in Heinrich, MIPS R4000 Microprocessor User's Manual (2d Ed. 1994).

[0084] Signal processor 400 also includes a DMA controller 454 and CPU control registers 456. DMA controller 454 is connected to the coprocessor internal bus 214, and is used to transfer data into and out of instruction memory 402 and/or data memory 404. For example, DMA controller 454 can copy microcode modules 156 from main memory 300 into signal processor instruction memory 402. DMA controller 454 may also be used to transfer information between data memory 404 and main memory 300. DMA controller 454 can be commanded by execution unit 430, and receives DMA address and data information from scalar unit data path 414 over path 438. DMA controller 454 may also be commanded by main processor 100 via CPU control registers 456, CPU control registers 456 are mapped into the main processor 100 address space, and can be accessed by signal processor 400 and execution unit 430 using MIPS "CP0" instruction formats.

[0085] Figures 7D-7L show example CPU control registers 756. The registers shown in Figures 7D-7H are used to control and/or monitor the DMA controller 454.

[0086] For example, the SP-DRAM DMA address register 458 shown in Figure 7D can be written to or read from by main processor 100 (as well as SP execution unit 430), and is used to specify a starting DMA address within instruction memory 402 or data memory 404. SP memory DMA address 460 shown in Figure 7E is used to specify a starting DMA address in main memory 300. Read and write DMA length registers 462, 464 shown in Figure 7F and 7G, respectively, specify the length of a block of data to be transferred between signal processor 400 and main memory 300—with the direction of transfer depending upon which one of these two registers is used to specify the block length. DMA status registers 466, 468 shown in Figures 7H and 7I respectively, can be read by main processor 100 to determine whether DMA controller 454 is full or busy, respectively.

[0087] Figure 7J shows the main SP status register 470 within CPU control registers 456. SP status register 470 acts as an SP control register when it is written to by main processor 100 (top diagram of Figure 7J), and indicates SP status when read by the main processor (bottom diagram in Figure 7J). When used as a status register, SP status register 470 tells main processor 100 whether the SP is halted (field 471), whether the SP is operating in a breakpoint mode (field 472), whether the DMA controller 454 is busy (field 474) or full (field 475), whether SP I/O is full (field 476), whether the SP is operating in single step mode (field 477), whether the SP is operating in a mode in which it won't generate an interrupt upon reaching a breakpoint (block 478), and whether the SP has generated various general purpose "signals" 479 that can be defined under software control to provide status concerning various software-dependent parameters. Main processor 100 can write to register 470 to stop or start signal processor 400 (fields 480, 481), to clear breakpoint mode (field 482), to dear or set an interrupt mode (fields 483, 484), to dear or set single step mode (fields 485, 486), to clear or set an interrupt on breakpoint mode (fields 487, 488), and to clear or set the various software-dependent "signals" (fields 489, 490).

[0088] Figure 7K shows an additional SP register 491 used as a "semaphore" for general purpose communications

between the main processor 100 and the signal processor 400. This register 491 contains a flag that main processor 100 sets upon reading the register and clears upon writing to the register. Signal processor 400 can also set or clear this flag.

[0089] Figure 7L shows-an SP instruction memory BIST status register 492 that is used as a BIST control register when written to by main processor 100 (top diagram in Figure 7L) and indicates BIST status when read by the main processor (bottom diagram of Figure 7L). Program counter 432 is preferably also mapped into the CPU control registers 456 so that it can be written to and read from by main processor 100.

Signal Processor Microcode

[0090] The particular functions signal processor 400 performs depend on the SP microcode 156 provided by storage device 54. In this example, SP microcode 156 provides both graphics and audio processing functions. As explained above, the main tasks performed by signal processor 400 for graphics processing include reading a display list, performing 3-dimensional geometry transformation and lighting calculations, and generating corresponding graphics display commands for use by display processor 500. In more detail, signal processor 400 performs the following overall graphics functions under control of microcode 156:

- Display list processing
- Matrix definition
- Vertex generation and lighting
- Texture definition/loading
- Clipping and culling
- Display processor command setup
- Flow control

Signal processor 400 performs the following overall functions under control of microcode 156 to process audio:

- Play list processing
- Digital audio synthesis/processing
- Writing digital audio samples to main memory audio buffer 114

Task Lists

[0091] Main processor 100 tells signal processor 400 what to do by providing the signal processor with a task list. The microcode 156 program that runs on signal processor 400 is called a task. Main processor 100 (and thus the video game program 108 supplied by storage device 54) is responsible for scheduling and invoking tasks on signal processor 400. The task list contains all of the information signal processor 400 needs to begin task execution, including pointers to the microcode 156 routines it needs to run in order to perform tasks. Main processor 100 provides this task list under control by game program 108.

[0092] Figure 8 shows an example of a task list 250. The task list 250 may reference one or more display lists and/or play lists 110. These display lists or play lists 110, in turn, may reference additional data structures including other display lists or play lists. A display list 110 can point to other display lists and/or graphics data. Similarly, a play list can reference other play list and/or sound data. In this example, display lists and play lists can be thought of as hierarchical data structures up to ten levels deep. Signal processor 400 processes the display lists and play lists of the stack, pushing and popping the current display list pointer. All display lists must terminate with an "end" command. For example, display list 110(1) shown in Figure 8 references another display list 110(2). Display list 110(2) references graphics data 112 needed to execute the list. Similarly, play list 110(4) shown in Figure 8 references sound data 112B.

[0093] For graphics animation, it is desirable to "double buffer" only parts of the display list 110 that change from one frame to another. In this way, only the data that changes from one frame to the next needs to be "double buffered"—thus conserving space in main memory 300. Swapping between double buffers is efficiently done by changing segment base addresses within task lists 250 and by organizing the hierarchical display lists in an appropriately efficient manner. Display lists or fragments of display lists can be chained together for more efficient memory utilization.

[0094] Figure 9 shows an example process performed by main processor 100 to invoke processing of a new task list by signal processor 400. Main processor 100 first loads the task (display) list into main memory 300 (block 601). It then halts signal processor 400 (or checks to insure that the signal processor is halted) by writing to and/or reading from SP status register 470 (block 602). Main processor 100 then writes to SP DMA registers 458, 460, 462 to load an initial microcode module into signal processor instruction memory 402 (604, Figure 9). Main processor 100 next stores the address in main memory 300 of the task (display) list loaded by block 601 into signal processor data memory

404 (block 606, Figure 9). Main processor 100 then resets the signal processor program counter 432 (block 608, Figure 9), and writes to SP status register 470 to start the signal processor 400 (block 610, Figure 9). The signal processor 400 typically then uses its DMA controller 454 to fetch the task (display) list from main memory 300 into its data memory 404.

[0095] Now that signal processor 400 has a task list and is started, it proceeds to perform each of the operations requested in the task list. It continues to execute the task list until it reaches the end of the task list, at which time it stops and waits for main processor 100 to provide a new task list. Generally, main processor 100 provides a new task list once each video frame--although, as discussed above, in many cases only a portion of the task list and/or the display and/or play lists the task list references may actually change from one frame to the next. Portions of the task list in main memory 300 may be "double buffered" so the main processor 100 can be writing to one buffer while signal processor 400 reads from another buffer. Before the next video frame, the main processor 100 can change a pointer to give the signal processor 400 access to the new buffer.

[0096] As signal processor 400 executes the task list, it retrieves additional SP microcode 156 modules from main memory 300 as needed to perform the specified tasks. For example, signal processor 400 may use its DMA facility 454 to load particular graphics microcode into instruction memory 402 to execute graphics commands specified by a task list, and may similarly retrieve and load audio processing microcode routines to perform audio processing specified by the task list. Different microcode routines or "overlays" may be loaded on an as-needed basis to more optimally handle particular types of graphics and/or audio processing operations. As one example, the signal processor 400 may load special lighting graphics routines as overlays to perform particular lighting operations, and may load clipping routines or overlays to perform particular culling operations. Microcode loading and reloading into signal processor 400 during execution of the single task list 250 is necessary in this example because signal processor instruction memory 402 is not large enough to store all of SP microcode 156, and the signal processor is designed so that it can execute instructions only out of its internal instruction memory.

[0097] Figure 10 shows an example of a simplified graphics process performed by signal processor 400 based on a display list 110. In this simplified process, the display list 110 first commands signal processor 400 to set various attributes defining the overall graphical images that are to be rendered by the co-processor. Such attributes include, for example, shading, lighting, Z buffering, texture generation, fogging and culling (Figure 10 block 612). The display list next commands signal processor 400 to define a modeling/ viewing matrix and a projection matrix (Figure 10, block 614). Once the appropriate matrices have been defined, the display list commands signal processor 400 to transform a set of vertices based on the modeling/viewing matrix and the projection matrix defined by block 614 and also based on the attributes set by block 612 (Figure 10, block 616). Finally, the display list commands signal processor 400 to generate a graphics display (e.g., triangle) command that directs display processor 500 to render a primitive based on the vertices generated by block 616 and the attributes set by block 612 (Figure 10, block 618). Signal processor 400 may, in response to step 618, transfer the display processor command it has generated (or the address of the command, which the signal processor may store in its data memory 404 or in main memory 300) for access and execution by display processor 500.

[0098] Figure 11 shows an overall process 620 performed by signal processor graphics microcode 156 to process a display list 110 (e.g., to perform the type of process shown in Figure 10). Signal processor 400 gets the next display list command and determines what kind of a command it is (Figure 11, block 622). Display lists commands in this example generally have five different types:

- Signal processor attribute command
- Display processor command
- Matrix command
- Vertex command
- Triangle command
- Flow control command

[0099] If the display list command is a signal processor attribute command, signal processor 400 sets signal processor attributes as specified by the command (Figure 11, block 624). In this example, the following types of SP attribute command are defined:

- shading
- lighting
- Z-buffering
- texturing
- fogging
- culling.

[0100] The following are example SP attribute command formats and associated definitions:

SIGNAL PROCESSOR ATTRIBUTE COMMANDS:	
G SETGEOMETRYMODE:	
command	
	command

[0101] This command "sets" some of the rendering pipeline state. This state is maintained in the signal processor 400, and a SET/CLEAR interface is presented to the user.

[0102] Bits which are "on" in the command field are turned ON in the internal state.

G_SHADE	Enable vertex shading or use primitive color to paint the polygon (default is vertex shading).
G_LIGHTING	Enable lighting calculations.
G_SHADING_SMOOTH	Enable smooth or flat shading (the default, with this bit cleared is flat shading).
G_ZBUFFER	Enable z-buffer depth calculations.
G_TEXTURE_GEN	Enable automatic generation of the texture coordinates S & T. After transformations, a spherical mapping will be used to replace any S & T value originally given with the vertex.
G_FOG	Enable fog coefficient to be generated and replace the vertex alpha. Large alphas are more foggy (farther).
G_TEXTURE_GEN_LINEAR	Enable linearization of the texture coordinates generated when G_TEXTURE_GEN is set. For example, this allows the use of a panoramic texture map when performing environment mapping.
G_LOD	Enable generation level of detail (LOD) value for mipmapped textures and texture-edge mode.
G_CULL_FRONT	Cull the front-facing polygons.
G_CULL_BACK	Cull the back-facing polygons.

G_CLEARGEOMETRY MODE:

[0103] Same as G_SETGEOMETRYMODE, but this command "dears" some of the rendering pipeline state (bits which are "on" in the command field are turned OFF in the internal state).

G_LIGHT:

Command	param	length = 16
	seg	address

↓

light.r	light.g	light.b	0x00
light.r	light.g	light.b	0x00
light.x	light.y	light.z	0x00

[0104] This command passes a light to the rendering pipeline. There can be up to 7 directional lights (numbered 1-7) plus an ambient light. The param specifies which light number (n) to replace with this light description. Use the G_NUM_LIGHTS command to specify how many of the 8 lights to use. If the number of lights specified is N, then the first N lights (1-N) will be the ones used, and the Nth+1 lights will be the ambient light. The "param" field should be set based on a value maintained in data memory 404 (n-1) x 2.

[0105] The ambient light is defined by a color: light.r, light.g, light.b (unsigned 8 bit integers) which should be set to the color of the ambient light multiplied by the color of the object which is to be drawn (If you are lighting a texture mapped object just use the color of the ambient light). (For ambient lights the light.x, light.y, and light.z fields are ignored). The ambient light cannot be turned off except by specifying a color of black in this example.

[0106] Directional lights are specified by a color: light.r, light.g, light.b (unsigned 8 bit integers) which, like the ambient light color, should be set to the color of the light source multiplied times the color of the object which is to be drawn. Directional lights also have a direction. The light.x, light.y, light.z fields (signed 8 bit fractions with 7 bits of fraction) indicates the direction from the object to light. There must be at least one directional light (if G_LIGHTING is enabled in G_SETGEOMETRYMODE command) turned on, but if its color is black it will have no effect on the scene.

[0107] The G_NUM_LIGHTS command should always be used sometime after G_LIGHT command(s) before the next G_VTX command even if the number of lights has not changed.

G_NUM_LIGHTS:

Command	param	length=8
	seg	address

↓

0x8000	32 x (1+N)
0x00000000	

[0108] N = number of diffuse light sources (1-7).

[0109] This command specifies how many lights should be used. It should always be used after the G_LIGHT command before the next G_VTX command. The parameter specifies the number of diffuse light sources (N) which must be at least 1 and not more than 7. The ambient light source will be light number N+1 and the directional light sources will be lights numbered 1 through N.

G_SETOTHERMODE_H:

command | shift | len

word

[0110] This command sets the high word of the "other" modes in the display processor, including blending, texturing, and frame buffer parameters. The signal processor 400 remembers the high and low words of the display processor 500 "other" state, in order to present a simple set-command interface. Although this is a display processor command, it must be parsed and interpreted by the signal processor 400 and therefore cannot be sent directly to the display processor without first going through the signal processor.

[0111] The shift and len parameters in this command are used to construct a mask:

$$(((0x01 << len) - 1) << shift)$$

This mask is used to clear those bits in the display processor 500 status word. New bits, from the word parameter are OR'd into the status word. (the parameter word must be pre-shifted).

G_SETOTHERMODE_L

[0112] Same as G_SETOTHERMODE_H, but affects the low word of the "other" modes on the display processor 500.

G_TEXTURE:

command | s scale | t scale | mipmap level | tile num | on

[0113] This command turns texture mapping ON/OFF, provides texture coordinate scaling, and selects the tile number (within a tiled texture). Scale parameters are in the format of (.16) and scale the texture parameters in vertex commands. Texture on/off turns on and off the texture coordinate processing in the geometry pipeline. Tile number corresponds to tiles chosen in the raster portion of the pipeline. The tile num also holds the maximum levels for level of detail (LOD) (mid-mapping).

G_LOOKAT_X:

Command	param	length=16
seg	address	

↓

0x00000000			
0x00000000			
X	Y	Z	0x00

[0114] This command is used for automatic texture coordinate generation. It is used to describe the orientation of the eye so that the signal processor 400 knows with respect to what to generate texture coordinates. The XYZ values (8 bit signed fractions with 7 bits of fraction) describe a vector in woridspace (the space between the MODELVIEW

matrix and the PROJECTION matrix) which is perpendicular to the viewer's viewing direction and pointing towards the viewer's right.

G_LOOKAT_Y:

[0115] Same as G_LOOKAT_X, but the first zero words in the addressed segment are zero (0x00000000).

DP Command Generation

[0116] Referring back to Figure 11, if the next display list command is one intended for display processor 500, signal processor 400 simply writes the command to the display processor (block 626 of Figure 11). Block 626 can either DMA the display processor command into display processor 500 via the X-bus 218, or it can deposit the display processor command in a buffer within main memory 300 for access by the display processor.

MATRIX COMMANDS

[0117] If the next display list command is a matrix command, signal processor 400 updates the state of the current matrix it is using (Figure 11, block 628) and places the updated matrix on the matrix stack (block 630). As mentioned above, in this example signal processor 400 maintains a 10-deep modeling/viewing matrix stack. New matrices can be loaded onto the stack, multiplied (concatenated) with the top of the stack, or popped off of the stack. In this example, signal processor 400 maintains a "one-deep" projection matrix. Therefore, new matrices can be loaded onto or multiplied with the current projection matrix, but cannot be pushed or popped.

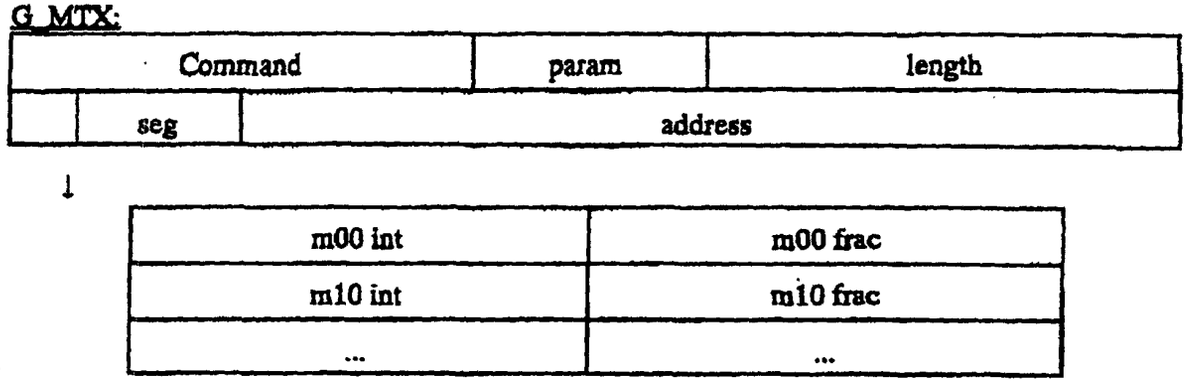
[0118] In this example, the modeling/viewing matrix stack resides in main memory 300. The video game program 108 must allocate enough memory for this stack and provide a pointer to the stack area in task list 250. The format of the matrix is optimized for the signal processor's vector unit 420. To provide adequate resolution, signal processor 400 in this example represents each matrix value in 32-bit "double precision"—with an upper 16 bit signed integer portion (indicating the part of the value greater than 1) and a lower 16-bit fractional portion (indicating the part of the value between 0 and 1). However, vector unit 420 in this example operates on 16-bit wide values and cannot directly multiply 32-bit wide values. The matrix format (which is shown in Figure 12B) groups all of the integer parts of the elements, followed by all of the fractional parts of the elements. It allows signal processor 400 to more efficiently manipulate the matrix by multiplying 16 bit integer parts and 16 bit fractional parts separately without have to repeatedly "unpack" or "pack" the matrix.

[0119] For example, vector unit 420 can multiply each of the 16-bit fixed point signed integer values in a matrix row in one operation, it can multiply each of the 16-bit fractional portions of the same row in another operation. These two partial results can be added together to obtain a 32-bit double precision value, or they can be used separately (e.g., for operations that require only the integer part of the result or only the fractional part of the result). Thus, matrix representations thus allows signal processor 400 to efficiently process 32-bit precision values even though vector unit 420 in this example, operates on 16-bit values and as no explicit "double precision" capability.

[0120] The following are example signal processor matrix commands and associated formats:

Example Matrix Commands:

[0121]



[0122] The matrix command points to a 4x4 transformation matrix (See Figure 12B) that will be used to transform the subsequent geometry, in a manner controlled by the flags in the parameter field. The length is the size of the incoming matrix in bytes. A 4x4 matrix pointed to by this command has the following format: It is a contiguous block of memory, containing the 16 elements of the matrix in ROW MAJOR order. Each element of the matrix is in a fixed point format, S15,16. The length of a 4 x 4 matrix in bytes should be 64 bytes. The segment id and address field are used to construct the main memory 300 address of the actual matrix. (see G_SEGMENT SP command for more information).

[0123] The following flags in the parameter field are used:

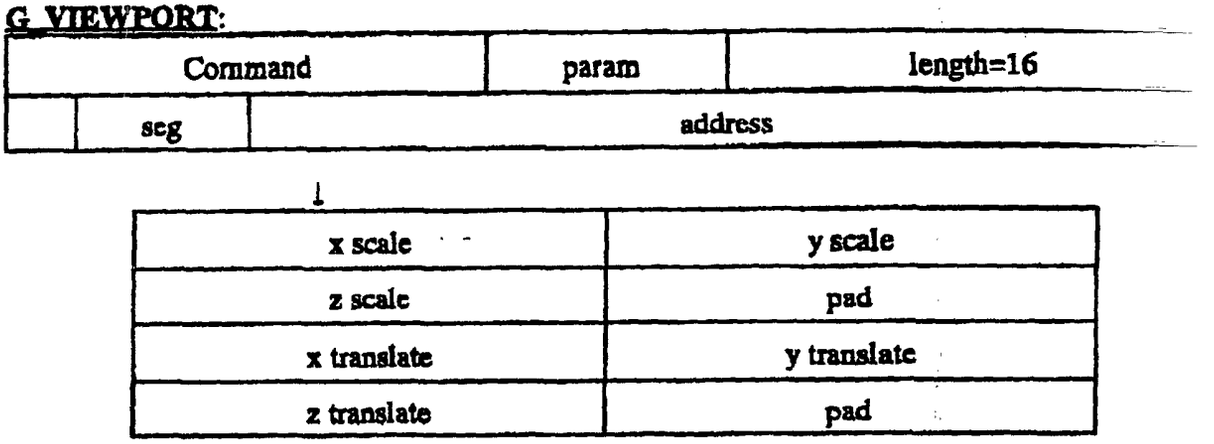
G_MTX_MODELVIEW	Identifies the incoming matrix as a modelview matrix, which is necessary to provide efficient transformation of polygon normals for shading, etc. (default)
G_MTX_PROJECTION	Identifies the incoming matrix as a projection matrix, which does not affect the transformation of the polygon normals for shading, etc.
G_MTX_MUL	The incoming matrix is concatenated with the current top of the matrix stack. (default)
G_MTX_LOAD	The incoming matrix replaces the current top of the (modelview or projection) matrix stack.
G_MTX_NOPUSH	The current top of the matrix stack is not pushed prior to performing the load or concat operation with the top of the stack. (default)
G_MTX_PUSH	The current top of the matrix stack is pushed prior to performing the load or concat operation with the top of the stack. Push is only supported with G_MTX_MODELVIEW, and not with G_MTX_PROJECTION.--Since there is no projection matrix stack (the projection must be explicitly reloaded)

[0124] This single command with the combination of parameters allows for a variety of commonly used matrix operations. For example, (G_MTX_LOAD| G_MTX_NOPUSH) replaces the top of the stack. (G_MTX_MUL| G_MTX_PUSH) performs a concatenation while pushing the stack for typical modeling hierarchy construction.

[0125] For lighting and texturing, the polygon normal also must be transformed by the inverse transpose of the modelview matrix (reference the "OpenGL Programming Guide"). This is the reason separate modelview and projection stacks are kept, and incoming matrices must be identified.

G POPMTX:	
command	
	param

[0126] This command pops the modelview matrix stack. The parameter field should be 0. Popping an empty stack results in...(doesn't pop). Since there is no projection matrix stack, this command is supported only for the modelview matrix.



[0127] This command sends a viewport structure to the graphics pipeline.

[0128] The segment id and address field are used to construct the main memory 300 address of the actual VIEWPORT structure (see G_SEGMENT for more information).

[0129] The viewport transformation is a scale-translation of the normalized screen coordinates. In general, the viewport must be constructed in cooperation with the projection matrix in order to meet the hardware requirements for screen device coordinates.

[0130] The scale and translation terms for x and y have 2 bits of fraction, necessary to accommodate the sub-pixel positioning in the hardware. The z values have no fraction.

[0131] Accounting for the fractional bits, using one of the default projection matrices, the viewport structure can be initialized like this:

```
(SCREEN_WD/2*4, (SCREEN_HT/2)*4, G_MAXZ, 0,/* scale */
(SCREEN_WD/2*4, (SCREEN_HT/2) * 4,0,0, /* translate */
```

Vertex Command Processing

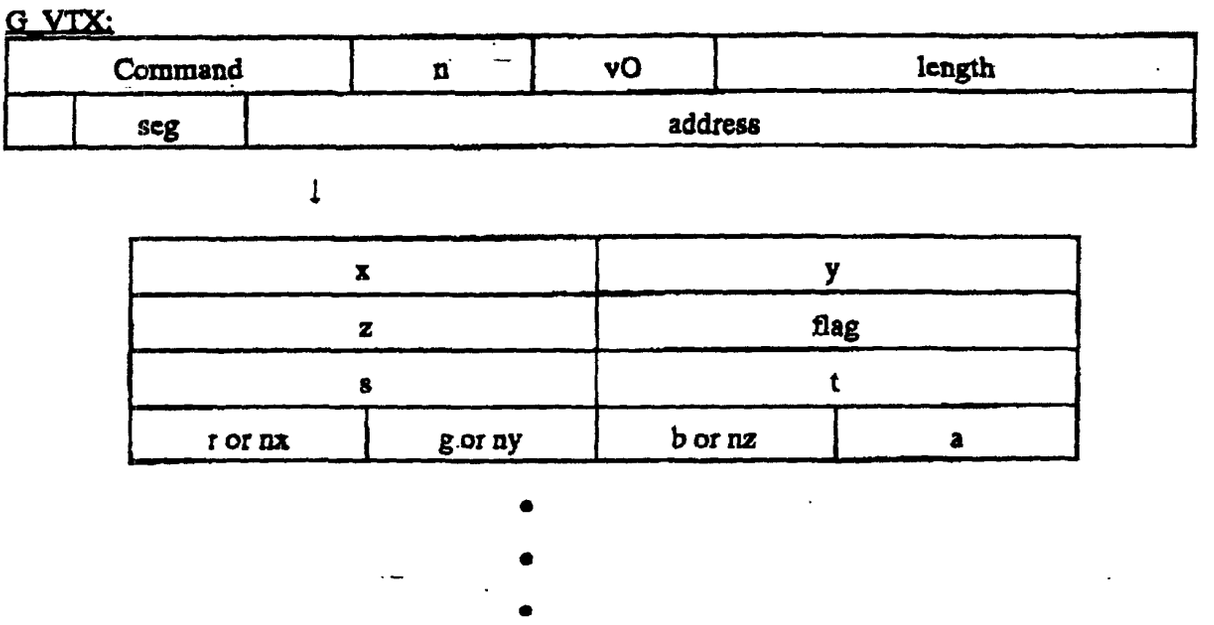
[0132] Referring once again to Figure 11, if the next display list command is a "vertex command", signal processor 400 transforms the vertices specified by the vertex command by the current matrix state and possibly shaded by the current lighting state, performs a clip test on the vertices, and loads the resulting vertices into a vertex buffer 408 within data memory 404. Signal processor 400 in this example has a vertex buffer that holds up to sixteen vertices. Figure 13A shows the signal processor 400 vertex buffer, which is fully exposed to main processor 100 and thus to video game program 108. This internal vertex buffer 404, which can hold up to 16 points, is stored in signal processor data memory 404 and can be read by main processor 100.

[0133] Although signal processor 400 in this example, can handle only lines, triangles or rectangles (i.e., surfaces defined by 2, 3, or 4 vertices), vertex buffer 408 in this example, stores up to 16 vertices so that the signal processor can reuse transformed vertex values instead of having to recalculate the vertices each time. 3D authoring/modeling software used to create video game program 108, in this example, should preferably organize display list 110 to maximize vertex reuse (and thus speed performance).

[0134] Figure 13B shows an example vertex data structure signal processor 400 uses to represent each of the vertices stored in vertex buffer 408. In this example, the transformed x, y, z, and w, values corresponding to the vertex are stored in double precision format, with the integer parts first followed by the fractional parts (fields 408 (1)(a)-408 (1)(h)). With vertex color (r, g, b, α) are stored in fields 408(1)(l)-408(1)(l), and vertex texture coordinates (s, t) are stored in fields 408(1)(m), 408(1)(n). Additionally, from this example, the vertex values in screen space coordinates (i. c., transformed and projected onto the viewing plane) are stored in fields 408(1)(o)-408(1)(t) (with the one/w value stored in double precision format). The screen coordinates are used by display processor 500 to draw polygons defined by the vertex. The transformed 3-dimensional coordinates are maintained in vertex buffer 408 for a clipping test. Since polygons (not vertices) are clipped, and since the vertices in vertex buffer 408 may be re-used for multiple polygons, these transformed 3D vertex values are stored for multiple possible clipping test to be performed. In addition, the vertex data structure 408(1) includes flags 408(1)(v) that signal processor 400 can use, for example, to specify clip test results (i.e., whether the vertex falls inside or outside of each of six different clip planes). The perspective projection factor stored in fields 408(1)(s), 408(1)(t) is retained for perspective correction operations performed by the display processor

texture coordinate unit (explain below).

[0135] The following is an example of a vertex command format used to load the internal vertex buffer with some points:



[0136] This command loads (n+1) points into the vector buffer beginning at location v0 in the vertex buffer. The segment id and address field are used to construct the main memory 300 address of the actual VTX structure. (see G_SEGMENT for more information). The number of vertices n, is encoded as "the number minus one", in order to allow a full 16 vertices to be represented in 4 bits. The length is the number of points times 16, the size of the VTX structure (in bytes). Vertex coordinates are 16-bit integers, the texture coordinates s and t are S10.5. The flag parameter is ignored in this example. A vertex either has a color or a normal (for shading). Colors are 8 bit unsigned numbers. Normals are 8 bit signed fractions (7 bits of fraction). (0x7f maps to +1.0, 0x81 maps to -1.0, and 0x0 maps to 0.0). Normal vectors must be normalized, i.e.,

$$\sqrt{x^2+y^2+z^2} \leq 127$$

[0137] Upon receiving a vertex command, signal processor 400 transforms the vertices specified in the vertex command using the current modeling/viewing matrix (Figure 11, block 632). See Neider et al, Open GL Programming Guide (Silicon Graphics 1993) at chapter 3 ("viewing"). These transformations orient the object represented by the vertices in 3-dimensional space relative to the selected view point. For example, they may translate, rotate and/or scale the represented object relative to a selected point of view. Such transformation calculations make heavy use of the signal processor vector unit 420 and its ability to perform eight parallel calculations simultaneously. The transformed results are stored in vertex data structure fields 408(1)(a)-408(1)(h) in double precision format in this example.

Clip Test:

[0138] Signal processor 400 then performs a clip test (Figure 11, block 636) to determine whether the transformed vertex is inside or outside of the scene. Six clipping planes define the sides and ends of the viewing volume. Each transformed vertex is compared to each of these six planes, and the results of the comparison (i.e., on which side of the clip plane the vertex is located) are stored in vertex buffer "flags" field 408(v) (sec Figure 13B). These results are used by clipping block 646 in response to a "triangle command" (see below). Note that because this example clips polygons and not vertices, Figure 11 block 636 does not actually perform clipping, it simply tests vertex position relative to the clip planes.

Projection:

[0139] Signal processor 400 then transforms the vertex values using the projection matrix (Figure 11, block 638). The purpose of the projection transformation is to define a viewing volume, which is used in two ways. The viewing volume determines how an object is projected onto the 2-dimensional viewing screen (that is, by using a perspective or an orthographic projection). (See Open GL Programming Guide at 90 *et seq*.) The resulting transformed vertices have now been projected from 3-dimensional space onto the 2-dimensional viewing plane with the proper for shortening (if the projection matrix defines a perspective projection) or orthographically (if the projection matrix defines an orthographic projection). These screen coordinates values arc also written to the vertex buffer data structure at fields 408(1)(o)-408(1)t) (the"1/w" value is retained for later perspective correction).

Lighting:

[0140] Signal processor 400 next performs lighting calculations in order to "light" each of the vertices specified in the vertex command. System 50 supports a number of sophisticated real-time lighting effects, including ambient (uniform) lighting, diffuse (directional) lights, and specular highlights (using texture mapping). In order to perform lighting calculations in this example, signal processor 400 must first load an SP microcode 108 overlay to perform the lighting calculations. The G_SETGEOMETRYMODE command must have specified that lighting calculations are enabled, and the lights must have been defined by the G_NUM_LIGHTS command discussed above. The part of microcode 108 that performs the lighting calculations is not normally resident within signal processor 400, but is brought in through an overlay when lighting calls are made. This has performance implications for rendering scenes with some objects lighted and others colored statically. In this example, the lighting overlay overwrites the clipping microcode, so to achieve highest performance it is best to minimize or completely avoid clipped objects in lighted scenes.

[0141] To light an object, the vertices which make up the objects must have normals instead of colors specified. In this example, the normal consists of three signed 8-bit numbers representing the x, y and z components of the normal (see the G_VTX command format described above). Each component ranges in value from -128 to +127 in this example. The x component goes in the position of the red color of the vertex, the y into the green and the z into the blue. Alpha remains unchanged. The normal vector must be normalized, as discussed above.

[0142] Lighting can help achieve the effect of depth by altering the way objects appear as they change their orientation. Signal processor 400 in this example supports up to seven diffused lights in a scene. Bach light has a direction and a color. Regardless of the orientation of the object and the viewer, each light will continue to shine in the same direction (relative to the open "world") until the light direction is changed. In addition, one ambient light provides uniform illumination. Shadows are not explicitly supported by signal processor 400 in this example.

[0143] As explained above, lighting information is passed to signal processor 400 in light data structures. The number of diffuse lights can vary from 0 to 7. Variables with red, green and blue values represent the color of the light and take on values ranging from 0 to 255. The variables with the x, y, z suffixes represent the direction of the light. The convention is that the direction points toward the light. This means the light direction indicates the direction to the light and not the direction that the light is shining (for example, if the light is coming from the upper left of the world the direction might be x = -141, y = -141, z = 0). To avoid any ambient light, the programmer must specify the ambient light is black (0, 0, 0,).

[0144] The G_light command is used to activate a set of lights on a display list. Once lights are activated, they remain on until the next set of lights is activated. This implies that setting up a new structure of lights overwrites the old structure of lights in signal processor 400. To turn on the lighting computation so that the lights can take effect, the lighting mode bit needs to be turned on using the G_SETGEOMETRYMODE command.

[0145] The lighting structures discussed above are used to provide color values for storing into vertex buffer fields 408(1)(i)-408(1)(1).

Texture Coordinate Scaling/Creation:

[0146] Signal processor 400 next performs texture coordinate scaling and/or creation (Figure 11, block 642). In this example, the operations performed by block 642 may be used to accomplish specular highlighting, reflection mapping and environment mapping. To render these effects, coprocessor 200 in this example uses a texture map of an image of the light or environment, and computes the texture coordinates s,t based on the angle from the viewpoint to the surface normal. This texture mapping technique avoids the need to calculate surface normals at each pixel to accomplish specular lighting. It would be too computationally intensive for system 50 in this example to perform such surface normal calculations at each pixel.

[0147] The specular highlight from most lights can be represented by a texture map defining a round dot with an exponential or Gaussian function representing the intensity distribution. If the scene contains highlights from other, oddly shaped lights such as fluorescent tubes or glowing swords, the difficulty in rendering is no greater provided a

texture map of the highlight can be obtained.

[0148] Although display processor 500 performs texture mapping operations in this example, signal processor 400 performs texture coordinate transformations for each vertex when these effects are required. Activation or de-activation of the signal processor texture coordinate transformations is specified by a value within the G_SETGEOMETRYMODE Command (see above). In addition, the G_SETGEOMETRYMODE Command can specify linearization of the generated textured coordinates, e.g., to allow use of a panoramic texture map when performing environment mapping.

[0149] In this example, signal processor 400 texture coordinate generation utilizes the projection of the vertex normals in the x and y directions in screen space to derive the s and t indices respectively for referencing the texture. The angle between the viewpoint and the surface normal at each vertex is used to generate s, t. The normal projections are scaled to obtain the actual s and t values in this example. Signal processor 400 may map the vertices "behind" the point of view into 0, and may map positive projections into a scaled value.

[0150] In this example, texturing is activated using the G_TEXTURE command described above in the signal processor attribute command section. This command provides, among other things, scaling values for performing the texture coordinate mapping described above.

[0151] As explained above, the texture coordinate mapping performed by signal processor 400, in this example, also requires information specifying the orientation of the eye so that the angle between the vertex surface normal and the eye can be computed. The G_LOOKAT_X and the G_LOOKAT_Y commands supply the eye orientation for automatic texture coordinate generation performed by signal processor 400. The transformed texture coordinate values, if they are calculated, are stored by signal processor 400 in the vertex data structure at fields 408(1)(m), 408(1)(n). These texture coordinate values are provided to display processor 500 to perform acquired texture mapping using a texture specified by the G_TEXTURE command.

[0152] Since these effects use texture mapping, they cannot be used with objects which are otherwise texture mapped.

Vertex Buffer Write:

[0153] After performing all of these various steps, signal processor 400 writes the transformed, lighted, projected vertex values into vertex buffer 408 (Figure 11, block 644), and returns to parse the next display list command (block 622).

Triangle Command Processing:

[0154] Once signal processor 400 has written vertices into its vertex buffer 408, the display list 110 can provide a "triangle command". The "triangle command," which specifies a polygon defined by vertices in vertex buffer 408, is essentially a request for signal processor 400 to generate a graphics display command representing a polygon and to send that command to display processor 500 for rendering. In this example, signal processor 400 can render three different kinds of primitives: lines, triangles and rectangles. Different modules of microcode 108 need to be loaded in this example to render lines or triangles. In this example, all rectangles are 2-dimensional primitives specified in screen-coordinates, and are neither clipped nor scissored.

[0155] The following is an example of a format and associated function of triangle commands:

Example of Triangle Commands

[0156] The following command specifies a triangle defined by 3 vertices in the vertex buffer:

G_TRI1:

command				
	N	v0	v1	v2

[0157] This command results in one triangle, using the vertices v0, v1, and v2 stored in the internal vertex buffer. The N field identifies which of the three vertices contains the normal of the face (for flat shading) or the color of the face (for flat shading).

[0158] The following command is used to control signal processor 400 to generate display processor 500 commands for rendering a line defined by two vertices in vertex buffer 408:

G_LINE3D:

command				
	N	v0	v1	

[0159] This command generates one line, using the vertices v0 and v1 in the internal vertex buffer. The N field specifies which of the two vertices contain the color of the face (for flat shading).

[0160] Textured and filled rectangles require intervention by signal processor 400 and are thus a signal processor operation. The following is an example command format and associated function of a texture rectangle command:

G_TEXRECT	
x1	y1

[0161] This command draws a 2D rectangle in the current fill color. The parameters x0, y0 specify the upper left comer of the rectangle; x1, y1 are the lower right comers. All coordinates are 12 bits.

Clipping/Setup:

[0162] Referring back to Figure 11, upon receipt of a triangle command, signal processor 400 performs any necessary clipping of the vertices (Figure 11, block 646). This clipping operation eliminates portions of geometric primitives that lie outside of the six clipped planes defining the view plane.

[0163] As explained above, the results of the clip test 636 performed for each vertex are stored and available in vertex buffer 408. With the triangle command now defining a primitive defined by those vertices, signal processor 400 can proceed to clip the primitive. If all of the vertices of a primitive lay within the space defined by the six clip planes, the entire primitive exists within the display space and does not need to be clipped. If all of the vertices defining a primitive lay outside of the same clip plane (as indicated by the flags field of vertex data structure 408(1) shown in Figure 13B), the entire primitive can be excluded from display and thus discarded. If some of the vertices defining a primitive lie within the display space and some lay outside of it (or if all vertices lay outside of the display space but define a primitive which passes through the displayed space), the primitive needs to be clipped and new vertices defined. These tests and operations are performed by clipping block 646 in this example.

[0164] Signal processor 400 next performs backface culling (Figure 11, block 647). This operation maximizes drawing speed by discarding polygons that can be determined to be on the backface of an object and thus hidden from view. In this example, either front-facing, back-facing, neither or both types of primitives can be culled (i.e., discarded) by block 647. The types of primitives to cull are specified by parameters in the G_SETGEOMETRYMODE command described above--allowing geometry to be ordered in any direction or where used with different culling flags to achieve various effects (e.g., interior surfaces, two-sided polygons, etc.).

[0165] Signal processor 400 also performs some set up operations (Figure 11, block 648), and may then pass a graphics display command to display processor 500 to control the display processor to render the primitive (Figure 11, block 650). As part of the set up operation (block 648), signal processor 400 in this example translates "segmented" addresses in the display list 110 into physical addresses that the display processor 500 can use (the display processor is a physical address machine in this example).

[0166] In this example, signal processor 400 uses a segment table 416 (see Figure 13C) to assist it in addressing main memory 300. More specifically, addresses within signal processor 400 may be represented by a table entry 417A and a 26-bit offset 417B. The table entry 417A references one of 16 base addresses within segment address table 416. The referenced base address may be added to the offset 417b to generate a physical address into main memory 300. Signal processor 400 constructs a main memory 300 address by adding the base address for the segment and a 26-bit offset (which could be provided, for example, by a display list 110). The segment table 416 is constructed based on the following example G_SEGMENT command:

G_SEGMENT

command		
	seg	address

[0167] This command adds an entry in the segment table 416 discussed above.

[0168] The segmented addressing used by signal processor 400 in this example can be useful to facilitate double-buffered animation. For example, video game program 108 can keep two copies of certain display list fragments within main memory 300, with the same offsets in two different segments. Switching copies of them is as easy as swapping the segment pointers in signal processor 400. Another use is to group data and textures in one segment and to group static background geometry in another segment. Grouping data might help optimize memory caching in main processor 100. All data which contains embedded addresses must be preceded by the appropriate G_SEGMENT command that loads the signal processor 400 segment table with the proper base address.

[0169] Although signal processor 400 can use the segment addressing scheme shown in Figure 13C, this arrangement is not available to display processor 500 in this example. Hence, part of set up processing 648 is to translate any segment addresses that point to data structures required for rendering into physical addresses that can be used directly by display processor 500.

DP Command Write:

[0170] The primary output of signal processor 400 for graphics purposes is one or more commands to display processor 500 that are outputted by Figure 11, block 650. Although main processor 100 (or storage device 54) can directly supply display processor 500 commands, for 3D images the signal processor 400 generally needs to perform the transformation processes described above to generate display processor commands representing transformed, projected lighted, clipped, culled primitives.

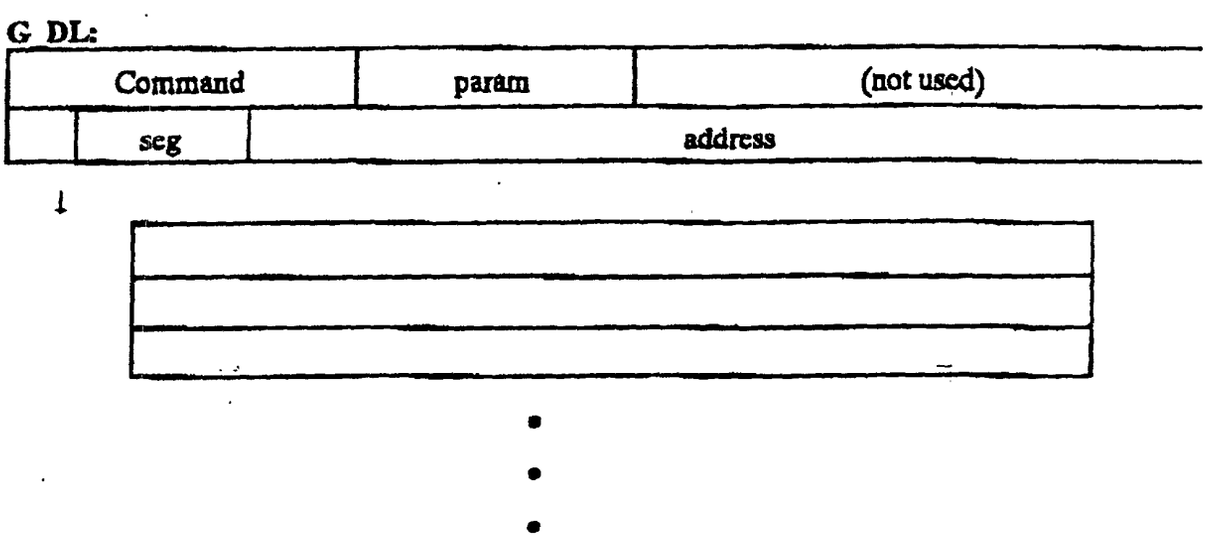
[0171] The repertoire of display processor commands is set forth in Appendix A. Signal processor 400 is responsible for formatting appropriately the display processor commands it generates, and for including the appropriate information and address information in the commands. In addition, signal processor 400 may generate and provide certain appropriate mode and attribute commands the display processor may require to render a particular primitive specified by the signal processor using the appropriate parameters (although many of the mode and attribute commands for the display processor 500 arc typically supplied directly by main processor 100 under control of game program 108). As mentioned above, main processor 100 can provide any display processor 500 directly, but in general, needs to rely on the signal processor to generate at least some display processor commands whenever 3D objects need to be transformed.

Flow Control Command Processing:

[0172] Referring once again to Figure 11, if the display list command received by signal processor 400 is a flow control command, then signal processor 400 will respond to this command in an appropriate manner to navigate through or traverse the display list 110. The following example commands and formats provide flow control.

Example Flow Control Commands:

[0173]



[0174] This command points to another display list and is used to create display list hierarchies, nested display lists, indirect references, etc. The segment field identifies a memory segment. The address field is the offset from the base of that segment. Together, these form an address in main memory 300 pointing to the new display list. A length field (not shown) may describe the length of the new display list in bytes—although in this example it is preferred that all display lists are terminated by a G_ENDDL command. The parameter field holds flags which control the behavior of the transfer. If the flag G_DL_NOPUSH is set, the current display list is not pushed onto the stack before transferring control. This behaves more like a branch or go to, rather than a hierarchial display list (this may be useful to break up a larger display list into non-contiguous memory pieces, then just connect them with display list branches).

G ENDDL:	
command	

[0175] The end display list command terminates this branch of the display list hierarchy, causing a "pop" in the processing of the display list hierarchy. This command is most useful for constructing display list pieces of variable or unknown size, terminated with an end command instead of providing a display list length a priori. All display lists must terminate with this command.

G_NOOP:			
command			

[0176] This command does nothing. It is generated internally under some circumstances.

[0177] Figure 11, block 652 performs the function of maintaining a display list stack in main memory 300 and, pushing and nooping (traversing) this display list stack. Block 652 halts signal processor 400 when the signal processor encounters an "open end" display list command.

SIGNAL PROCESSOR MICROCODE AUDIO PROCESSING

[0178] Signal processor 400 in this example performs digital audio processing in addition to the graphics processing discussed above. Signal processor vector unit 420 is especially suited for performing "sum of products" calculations that are especially useful in certain types of digital signal processing for audio signals such as, for example, audio decompression, wavetable resampling, synthesis and filtering. Digital spatial and/or frequency filtering with a relatively large number of taps can be accommodated without loss of precision because of the 48-bit-wide accumulators contained with vector unit data paths 423. As one example of a particular optimum usage of vector unit 420 for audio processing, the eight separate register files 422 and associated data paths 423 of signal processor vector unit 420 can be used to simultaneously process eight different MIDI voices in parallel. The following are examples of additional audio processing that can be efficiently performed using vector unit 420:

- solving polynomial equations,
- processing 8 audio voices or 8 time samples in parallel,
- wavetable synthesis using cubic interpolation, wherein four of the vector unit data paths 423 are used to process one sample, and the other four vector unit data paths are used to process a second sample,
- audio enveloping processing wherein the 8 vector unit data paths can each multiply a different audio sample by a different weighting factor, and
- audio mixing processing wherein the 8 vector unit data paths can each multiply a different audio sample by a corresponding mixer weighting factor.

[0179] Because signal processor 400 can perform audio digital signal processing efficiently at high speed, it takes the signal processor only a small fraction of an audio playback real time interval to perform and complete the digital audio processing associated with that time interval. For example, signal processor 400 takes much less than 1/30th of a second to digitally process audio that coprocessor audio interface 208 will playback in real time over a 1/30th of a second time interval. Because of this capability, signal processor 400 in this example can be time-shared between graphics processing and digital audio processing.

[0180] Generally, main processor 100 gives signal processor 400 a task list 250 at the beginning of a video frame

that specifies the image and sound to be produced during the next succeeding video frame. Coprocessor 200 must be finished with both the audio and graphics processing for this next succeeding frame by the time that next succeeding frame begins. Because video display and audio playback is a real time continuous process (i.e., a new video image must be provided each video frame time, and audio must be continuously provided), coprocessor 200 needs to finish all audio and video signal processing associated with each next succeeding video frame by the time that next frame begins.

[0181] In this example, signal processor 400 is shared between graphics processing and digital audio signal processing. Because of the high speed calculating capabilities of signal processor vector unit 420, signal processor 400 is able to complete processing of the audio to be played during the next succeeding video frame in much less than the current video frame time, and is also able to complete graphics processing for the image to be displayed during the next succeeding image in less than the current frame time. This allows task list 250 to specify both graphics display lists and audio play lists that all must be completed by signal processor 400/coprocessor 200 by the beginning of the next video frame time. However, in this example there is nothing to prevent main processor 100 from giving coprocessor 200 a task list 250 that the coprocessor cannot complete before the next video frame begins. If the combined audio and graphics processing required by signal processor 400 is sufficiently intensive and time-consuming, the signal processor 400 can work on processing the task list for the entire current video frame time and still not be done by the beginning of the next video frame. It is up to video game program 108 to avoid overtaxing coprocessor 200, and to handle any overtaxing in an appropriate manner should it occur. A video game programmer can avoid overtaxing signal processor 400 by ensuring that all display lists 110 are organized efficiently, modeling the objects in 3-D in an efficient manner, and taking precautions to ensure that extensive time consuming processing (e.g., clipping) is avoided or minimized. Even with such precautions, however, it may take coprocessor 200 more than a single video frame time to complete especially complicated images. A video game programmer can handle this situation by slowing down the effective frame rate so that television 58 redisplays the same image stored in one part of frame buffer 118 for multiple video frames during which time coprocessor 200 can complete processing the next image. Because the user may perceive a variable frame rate as undesired delay, it is often best to slow down the overall effective frame rate to the rate required for coprocessor 200 to complete the most processing-intensive images—thus preventing more complex images from appearing more slowly than less complex images.

[0182] With respect to audio processing, it is generally unacceptable to fail to provide audio for a given video frame time since the user will hearing a disturbing "click" in a stream of otherwise continuous audio. Such audio disruptions are easily heard and can be annoying. Therefore, they should be avoided. One way to avoid an easily detectable audio disruption in a situation where signal processor 400 has failed to complete its assigned audio processing in time is for main processor 100 to command audio interface 208 to replay the last frame's worth of audio during the next succeeding frame. Acceptable audio can be produced in this way without the user noticing a disruption if done carefully. Other strategies include having signal processor 400 process multiple video frames worth of audio within a single video frame time—thereby providing an effective audio "frame" rate that is different (faster) than the effective video frame rate. By "effective frame rate" we mean the rate at which coprocessor 200 produces a frame's worth of information (in this example, the television actual video frame rate stays constant).

Example Audio Software Architecture

[0183] Figure 14 shows an example of the overall software architecture provided by system 50 to synthesizc and manipulate audio. This overall software architecture 700 includes four software objects, in this example a sequence player 702, a sound player 704, a synthesis driver 706 and audio synthesis microcode 708. In this example, sequence player 702, sound player 704, and synthesis driver 706 all execute on main processor 100, and audio synthesis microcode 708 runs on coprocessor signal processor 400. Thus, sequence player 702, sound player 704 and synthesis driver 706 are each supplied as part of game program 108 of storage device 54, and audio synthesis microcode 708 is supplied as part of SP microcode 156.

[0184] Sequence player 702, sound player 704 and synthesis driver 706 may differ depending on the particular video game being played. In general, sequence player 702 is responsible for the playback of Type 0 MIDI music sequence files. It handles sequence, instrument bank and synthesizer resource allocation, sequence interpretation, and MIDI message scheduling. Sound player 704 is responsible for the playback of all ADPCM compressed audio samples. It is useful for sound effects and other streamed audio. Synthesis driver 706 is responsible for creating audio play lists 110 which are packaged into tasks by main processor 100 under software control and passed to coprocessor 200 in the form of task lists 250. In this example, synthesis driver 706 allows sound player 704 or other "clients" to assign wave tables to synthesizer voices, and to control playback parameters. As discussed above, the audio synthesis microcode 708 processes tasks passed to it and synthesizes L/R stereo 16-bit samples, which signal processor 400 deposits into audio buffers 114 within main memory 300 for playback via audio interface 208, audio DAC140 and amplifier/mixer 142.

[0185] In this example, synthesis driver 706 passes audio tasks to signal processor 400 in the form of audio "frames." A "frame" is a number of audio samples-usually something close to the number of samples required to fill a complete video frame time at the regular video frame rate (for example, 30 or 60 Hz). Although television set 58 receives and processes audio signals in a continuous stream unconstrained by any video frame rate parameter (e.g., the television can generate audio during horizontal and vertical video blanking and retrace), system 50 in this example organizes audio processing in terms of video frame rate because signal processor 400—which is shared between audio and graphics processing—must operate in accordance with the video frame rate because the graphics related tasks it performs are ded to the video frame rate.

Example Play List Processing

[0186] Figure 15 shows an example of a simple signal processor play list process. The Figure 15 process is specified by a play list 110 generated by main processor 100 under control of video game program 108, and specified as part of a task list 250. Thus, the Figure 15 SP play list process is an example of an output of synthesis driver 706 that is provided to signal processor 400 in the form of an audio play list 110.

[0187] Because of the limited size of instruction memory 402 in this example, audio synthesis microcode 708 is generally not continuously resident within signal processor 400. Instead, the initialization microcode main processor 100 arranges to be loaded into instruction memory 402 (see Figure 9, block 604), ensures that the appropriate audio microcode routine is loaded into the instruction memory for audio processing (also ensures that the appropriate graphics microcode routine is loaded into the instruction memory for graphics processing). The steps shown in Figure 15 assume that the audio synthesis microcode 708 is resident within the signal processor instruction memory 402, and that the signal processor 400 is reading an audio play list 110 specifying the steps shown.

[0188] Generally, the first task of an audio play list 110 is to set up buffers within signal processor data memory 408 required to perform the audio processing task (Figure 15, block 710). Generally, this buffer set up process involves allocating areas within data memory 404 to be used as one or more audio input buffers, and allocating an audio output buffer within the data memory. Generally, main processor 100 also commands signal processor 400 to use its DMA facility 454 to retrieve audio input data 112b from main memory into the allocated input buffer(s) for processing. Main processor 100 may next set certain attributes (e.g., volume ranges and change rates) to be used for the audio processing (Figure 15, block 712). Main processor 100 then specifies the types of signal processing to be performed by signal processor 400 along with appropriate parameters (Figure 15, block 714). In this example, main processor 100 can specify decompression, resampling, envelope/pan, mixing, and other processing (e.g., reverb) to be performed individually or in combination. The audio play list 110 typically will terminate with a command to save the contents of the output audio buffer stored in signal processor data memory 404 into main memory 300 (block 716).

Example Audio Synthesis Microcode

[0189] Figure 16 shows the overall tasks performed by audio synthesis microcode 708 in this example. Signal processor 400 under microcode control retrieves the next play list command from the current audio play list 110, and determines what kind of command it is (Figure 16, block 718). In this example, the audio command within an audio play list 110 may fall into the following general types:

- buffer command
- flow control command
- attribute command
- decompress command
- resample command
- envelope/pan command
- mix command
- special signal processing/effects command.

Buffer Command Processing:

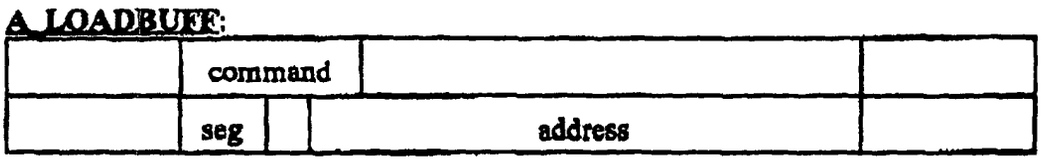
[0190] Buffer commands manage audio buffers within signal processor data memory 404, and permit audio data to be transferred between the data memory and main memory 300. The following are examples of buffer command formats and associated functions:

Example Buffer Commands:

[0191]

A SETBUFF:				
	command		dmemin	
	dmemout		count	

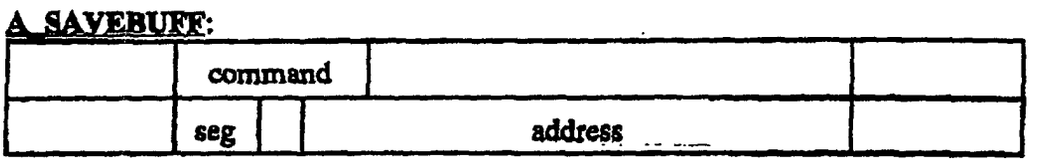
This command sets the internal signal processor data memory 404 buffer pointers and count value used by the processing commands. This command is typically issued before any processing command. *dmemin* points to an input buffer, *dmemout* to an output buffer and *count* defines the number of 16 bit samples to process.



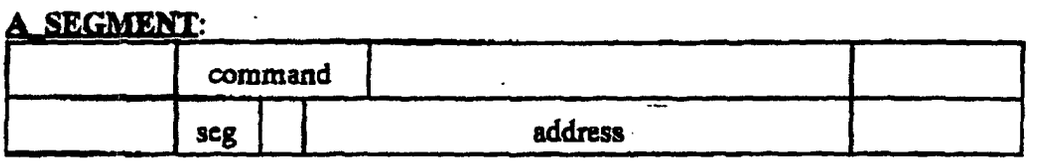
This command loads a signal processor data memory 404 buffer from the main memory 300 address given by the *seg+address* fields. The SP data memory buffer location and the number of 16 bit samples to load are defined by issuing an A_SETBUFF command prior to the A_LOADBUFF command.

A CLEARBUFF:				
	command		dmemin	
			count	

[0192] This command clears an area of size *count* 16 bit samples starting at the signal processor 400 data memory address given by *dmem*.



[0193] This command saves a buffer of 16 bit samples in the signal processor data memory 404 to the main memory 300 address given by the seg+address field. The input SP data memory buffer and number of samples are defined by issuing a A_SETBUFF command.



See graphics G-SEGMENT command. This command is used to map indirect "segment" addresses into main memory 300 physical addresses.

[0194] Referring again to Figure 16, signal processor audio synthesis microcode 708 performs the specified buffer command by establishing, managing, writing data into, or reading data from the associated data memory buffer 409 (Figure 16, block 720). Typically, signal processor 400 may use its DMA facility 454 to transfer data between main

memory 300 and signal processor memory 404 in order to retrieve audio input data for processing or save audio data into main memory for playback by audio interface 208.

Flow Control Command Processing:

[0195] If the next play list command is a flow control command, signal processor 400 responds to the command by traversing the current audio play list in the manner specified by the command. Nesting of audio play lists 110 is preferably permitted, and signal processor 400 may maintain an audio play list stack in main memory 300 (just as it may do for graphics display lists).

Attribute Command Processing:

[0196] If the next audio play list command is an attribute command, signal processor 400 processes the command by establishing appropriate mode and/or attribute conditions to be used for subsequent audio processing (Figure 16, block 724). In this example, audio synthesis microcode 708 supports the following example attribute command format and associated function:

Example Attribute Commands:

[0197]

A SETVOL:				
	command		volume	
	volume target		volume rate	

This command is used to set the volume parameters for subsequent processing commands. Currently this should be issued prior to A_ENVELOPE, A_PAN and A_RESAMPLE.

Decompress Command Processing

[0198] If the next audio play list command retrieved by signal processor 400 is a decompression command, the signal processor performs a decompression operation to decompress a compressed audio binary stream stored in an input buffer within data memory 404 to produce 16-bit audio samples which it stores in a defined audio output buffer within its data memory (Figure 16, block 726). In this example, audio synthesis microcode 708 supports the following audio decompression command format and associated function:

Example Decompression Command:

[0199]

A_ADPCM:

	command	flags	gain	
	seg		address	

[0200] This command decompresses a binary stream in signal processor data memory 404 to produce 16 bit samples. The addresses in the data memory 404 for the input and output buffers and the number of samples to process are defined by issuing a A_SETBUFF command prior to the A_ADPCM command. The *seg+address* field points to a main memory 300 location which is used to save and restore state. The gain parameter is used to scale the output and is represented as S.15.

[0201] The *flags* define the behavior of the command. Currently defined *flags* are:

A_INIT, The *seg+address* field is used to restore state at the beginning of the command. If not set the pointer to

state is ignored upon initiation, however, state is saved to this address at the end of processing.

A_MIX, The results are mixed into the output buffer. If not set results are put into the output buffer.

Resample Command Processing:

[0202] If the next audio play list command signal processor 400 reads is a resample command, then the signal processor provides pitch shifting/resampling as well as integral envelope modulation based on the parameters specified in the command (Figure 16, block 728). The following is an example of a resample command and associated function supported by audio synthesis microcode 708.

Example Resample Command:

[0203]

A_RESAMPLE:

	command	flags	pitch	
	seg		address	

[0204] This command provides pitch shifting/resampling as well as integral envelope modulation. The signal processor data memory 404 input and output buffers and the number of samples are defined by issuing an A_SETBUFF command, and the volume envelope parameters are defined by issuing an A_SETVOL command. Resampling factor is defined by *pitch*.

[0205] The *flags* define the behavior of the command. Currently defined flags are:

A_INIT, The *seg+address* field field is used to restore state at the beginning of the command. If not set the pointer to state is ignored upon initiation, however, state is saved to this address at the end of processing.

A_MIX, The results are mixed into the output buffer. If not set results are put into the output buffer.

Envelope/Pan Command Processing:

[0206] If the next audio play list command signal processor 400 reads is an envelope/pan command, the signal processor performs that command by modulating one or two audio signal streams using a linear envelope (Figure 16, block 730). An envelope command multiplies an audio input sample stream by a linear function, and is thus able to ramp the volume of the audio up or down. A "pan" command generally applies inverse linear functions to audio in left and right stereo channels—accomplishing the effect of moving the perceived source of a sound or voice in space (i. e., from left to right or from right to left). The following examples of envelope/pan command formats and associated functions are supported by audio synthesis microcode 708 in this example of system 50.

Example Envelope/Pan Commands:

[0207]

A_ENVELOPE:

	command	flags		
	seg		address	

[0208] This command modulates a sample stream using a linear envelope. The parameters for the volume envelope are defined by issuing A_SETVOL and the signal processor data memory 404 buffer locations and number of samples

to process are defined by issuing an A_SETBUFF prior to issuing the A_ENVELOPE command.

[0209] The *flags* define the behavior of the command. Currently defined flags are:

A_INIT, The *seg+address* field field is used to restore state at the beginning of the command. If not set the pointer to state is ignored upon initiation, however, state is saved to this address at the end of processing.

A_MIX, The results are mixed Into the output buffer. If not set results are put into the output butter.

A_PAN:

	command	flags	dmemout2	
	seg		address	

[0210] This command provides 1 input, 2 output panning. Input, first output and number of samples are defined by issuing an A_SETBUFF command and the panning parameters are defined by issuing an A_SETVOL command. The second output is defined by *dmemout2*.

[0211] The *flags* defined the behavior of the command. Cuirently defined flags are:

A_INIT, The *seg* + *address* field field is used to restore state at the beginning of the command. If not set the pointer of state is ignored upon initiation, however, state is saved to this address at the end of processing.

A_MIX, The results are mixed into the output buffer. If not sct results are put into the output buffer.

Mixing Command Processing:

[0212] If the next audio play list command is a mixing command, signal processor 400 performs a mixing function to mix two audio input sample streams into the output audio buffer (Figure 16, block 732). The following example mixing command format and associated function is supported by signal processor 400 and audio synthesis microcode 708 in this example.

A MIXER:				
	command		gain	
			dmemoutf	

This command provides a double precision mixing function. The single precision input is added to the double precision output after multiplication by gain. *dmemoutf* points to a signal processor data memory 404 area which stores the fractional part of the mixed stream. The input buffer, number of samples and integer part of the mixed output are defined by issuing an A_SETBUFF prior to the A_MIX.

Special Audio Effects Processing:

[0213] If the next audio play list command is a special signal processing/effects command, signal processor 400 executes the command by providing the specified special effect or signal processing (Figure 16, block 734). An example special signal processing/effect is the addition of reverberation to create presence. This special effect simulates sound reflection in caves, concert halls, etc., and can also be used for various other special effects. Signal processor 400 and audio synthesis microcode 708 supports the following example reverberation special effects command format and associated function:

Example Effects Command:

[0214]

A_REVERB:

	command	flags		
	seg		address	

[0215] This command applies the reverb special effect to a sample stream. Signal processor data memory 404 input, output and number of samples are defined by issuing an A_SETBUFF command.

[0216] *The flags* define the behavior of the command. Currently defined flags are:

A_INIT, The *seg+address* field field is used to restore state at the beginning of the command. If not set the pointer to state is ignored upon initiation, however, state is saved to this address at the end of processing.

A_MIX, The results are mixed into the output buffer. If not set results are put into the output buffer.

Audio Processing Structure:

[0217] To accomplish each of audio processing functions 728, 730, 732, 734 in this example, audio synthesis microcode 708 uses a general purpose effects implementation that manipulates data in a single delay line. Figure 17 shows an example general purpose audio processing implementation 740. In this example, the audio input samples can be conceived of as-being applied to the input of contiguous single delay line 742. The output tap of the delay line is applied through a gain 744 to the audio output buffer within signal processor data memory 404. Samples from another tap on delay line 742 are passed through a summer 746 and returned to the delay line directly (over path 748) and also through a coefficient block 750, another summer 752 and a low pass filter 754. A further tap 756 from delay line 742 is connected to the other input of summer 752 and also to the other input of summer 746 (this time through a further coefficient block 758). This generalized implementation 740 allows a particular effect to be constructed by attaching an arbitrary number of effect primitives to single delay line 742. The parameters for each primitive in the effect are passed through via the commands discussed above. Each primitive consists of an all-pass with a variable length tap followed by a DC normalize (unity gain at DC) single poll low-pass filter 754 followed by an output gain 744 specifying how much of this primitive's output is to be contributed to the final effect output. The value of each of the parameters for a primitive specifies the function of that primitive as a whole within the effect. Note that in Figure 17, the feedback coefficient 758 can be used to construct an "all-pass inside a comb" reverb (in response to the a_reverb command discussed above).

[0218] The general nature of implementation 740 does not mean that all functions are implemented. Only those functions which are driven by legitimate parameters actually generate audio command operations by signal processor 400. This gives video game programmers a great degree of flexibility in defining an effect that is appropriate in terms of both sonic quality and efficiency.

COPROCESSOR DISPLAY PROCESSOR 500

[0219] Display processor 500 in this example rasterizes triangles and rectangles and produces high quality pixels that are textured, anti-aliased and z-buffered. Figure 18 shows the overall processes performed by display processor 500. Display processor 500 receives graphics display commands that, for example, specify the vertices, color, texture, surface normal and other characteristics of graphics primitives to be rendered. In this example, display processor 500 can render lines, triangles, and rectangles. Typically, display processor 500 will receive the specifications for the primitives it is to render from signal processor 400, although it is also possible for main processor 100 to specify these commands directly to the display processor.

[0220] The first operation display processor 500 performs on an incoming primitive is to rasterize the primitive, i.e., to generate pixels that cover the interior of the primitive (Figure 18, block 550). Rasterize block 550 generates various attributes (e.g., screen location, depth, RGBA color information, texture coordinates and other parameters, and a coverage value) for each pixel within the primitive. Rasterize block 550 outputs the texture coordinates and parameters to a texture block 552. Texture block 552 accesses texture information stored within texture memory 502, and applies

("maps") a texel (texture element) of a specified texture within the texture memory onto each pixel outputted by rasterized block 550. A color convert block 554 and a chroma keying block 556 further process the pixel value to provide a texture color to a color combined block 558.

[0221] Meanwhile, rasterize block 550 provides a primitive color (e.g., as a result of shading) for the same pixel to color combine block 558. Color combined block 558 combines these two colors to result in a single pixel color. This single pixel color output may have fog applied to it by block 560 (e.g., to create the effect of a smoke filled room, or the less extreme, natural effect of reducing color brilliance as an object moves further away from the viewer). The resulting pixel color value is then blended by a block 562 with a pixel value framebuffer 118 stores for the same screen coordinate location. An additional anti-alias/z-buffer operation 564 performs hidden surface removal (i.e., so closer opaque objects obscure objects further away), anti-aliasing (to remove jaggedness of primitive edges being approximated by a series of pixels), and cause the new pixel value to be written back into framebuffer 118.

[0222] The operations shown in Figure 18 are performed for each pixel within each primitive to be rendered. Many primitives may define a single complex scene, and each primitive may contain hundreds or thousands of pixels. Thus, display processor 500 must process millions of pixels for each image to be displayed on color television set 58.

[0223] Typically, framebuffer 118 is "double buffered" -- meaning that it is sized to contain two complete television screen images. Display processor 500 fills one screen worth of framebuffer information while video interface 210 reads from the other half of the framebuffer 118. At the end of the video frame, the video interface 210 and display processor 500 trade places, with the video interface reading from the new image representation just completed by display processor 500 and the display processor rewriting the other half of the framebuffer. This double buffering does not give display processor 500 any more time to complete an image; it must still finish the image in nominally one video frame time (i.e., during the video frame time just prior to the frame time during which the new image is to be displayed).

Pipelining:

[0224] Because high speed operation is very important in rendering pixels, display processor 500 has been designed to operate as a "pipeline." Referring again to Figure 18 "pipelining" means that the various steps shown in Figure 18 can be performed in parallel for different pixels. For example, rasterize block 550 can provide a first pixel value to texture block 552, and then begin working on a next pixel value while the texture block is still working on the first pixel value. Similarly, rasterize block 550 may be many pixels ahead of the pixel that blend block 562 is working on.

[0225] In this example, display processor 500 has two different pipeline modes: one-cycle mode, and two-cycle mode. In one-cycle mode, one pixel is processed for each cycle time period of display processor 500. A one-cycle mode operation is shown in Figure 19A. Note that the operations shown in Figure 19A are themselves pipelined (i.e., the blend operation 562 operates on a different pixel than the rasterize operation 550 is currently rasterizing), but the overall operation sequence processes one pixel per cycle.

[0226] Figure 19B shows the two-cycle pipeline mode operation of display processor 500 in this example. In the Figure 19B example, some of the operations shown in Figure 18 are performed twice for each pixel. For example, the texture and color convert/filtering operations 552, 554 shown in Figure 18 are repeated for each pixel; the color combine operation 558 is performed twice (once for the texture color output of one texture operation, and once for the texture color output of the other texture operation). Similarly, blend operation 562 shown in Figure 18 is performed twice for each pixel.

[0227] Even though these various operations are performed twice, display processor 500 in this example does not contain duplicate hardware to perform the duplicated operations concurrently (duplicating such hardware would have increased cost and complexity). Therefore, in this example, signal processor 500 duplicates an operation on a pixel by processing it with a particular circuit (e.g., a texture unit, a color combiner or a blender), and then using the same circuit again to perform the same type of operation again for the same pixel. This repetition slows down the pipeline by a factor of two (each pixel must "remain" at each stop in the pipeline for two cycles instead of one), but allows more complicated processing. For example, because the two-cycle-per-pixel mode can map two textures onto the same pixel, it is possible to do "trilinear" ("mipmapping") texture mapping. In addition, since in this example, display processor 500 uses the same blender hardware to perform both the fog operation 560 and the blend operation 562 (but cannot both blend and fog simultaneously), it is generally necessary to operate in the two-cycle-per-pixel mode to provide useful fog effects.

[0228] The following tables summarize the operations performed by the various blocks shown in Figures 19A and 19B during the one-cycle and two-cycle modes:

Display Processor Pipeline Block Functionality in One-Cycle Mode	
Block	Functionality

(continued)

Display Processor Pipeline Block Functionality in One-Cycle Mode	
Rasterize 550	Generates pixel and its attribute covered by the interior of the primitive.
Texture 552	Generates 4 texels nearest to this pixel in a texture map.
Filter Texture 554	Bilinear filters 4 texels into 1 texel, OR performs step 1 of YUV-to-RGB conversion.
Combine 558	Combines various colors into a single color, OR performs step2 of YUV-to-RGB conversion.
Blend 562	Blends the pixel with framebuffer memory pixel, OR fogs the pixel for writing to framebuffer.
Framebuffer 563	Fetches and writes pixels (color and z) from and to the framebuffer memory.

Display Processor Pipeline Block Functionality in Two-Cycle Mode	
Block	Functionality
Rasterize 550	Generates pixel and its attribute covered by the interior of the primitive.
Texture 552a	Generates 4 texels nearest to this pixel in texture map. This can be level X of a mipmap.
Texture 552b	Generates 4 texels nearest to this pixel in a texture map. This can be level X+1 of a mipmap.
Filter Texture 554a	Bilinear; filters 4 texels into 1 texel.
Filter Texture 554b	Bilinear; filters 4 texels into 1 texel.
Combine 558a	Combines various colors into a single color, OR linear interpolates the 2 bilinear filtered texels from 2 adjacent levels of a mipmap, OR performs step 2 of YUV-to-RGB conversion.
Combine 558b	Combines various colors into a single color, OR chroma keying.
Blend 562a	Combines fog color with resultant CC1 color.
Blend 562b	Blends the pipeline pixels with framebuffer memory pixels.
Framebuffer 563a	Read/modify/write color memory; and
Framebuffer 563b	Read/modify/write Z memory.

Fill and Copy Operations:

[0229] Display processor 500 also has a "fill" mode and a "copy" mode, each of which process four pixels per cycle. The fill mode is used to fill an area of framebuffer 118 with identical pixel values (e.g., for high performance clearing of the framebuffer or an area of it). The copy mode is used for high-performance image-to-image copying (e.g., from display processor texture memory 502 into a specified area of framebuffer 118). The copy mode provides a bit "blit" operation in addition to providing high performance copying in the other direction (i.e., from the framebuffer into the texture memory).

[0230] The pipeline operations shown in Figures 19A and 19B are largely unused during the fill and copy modes, because in this example, the operations cannot keep up with the pixel fill or copy rate. However, in this example, an "alpha compare" operation (part of blend operation 562) is active in the copy mode to allow display processor 500 to "blit" an image into framebuffer 118 and conditionally remove image pixels with the word alpha=0 (e.g., transparent pixels).

[0231] The display processor's mode of operation is selected by sending the display processor 500 a "set other mode" command specifying a "cycle type" parameter. See Appendix A. In the one-cycle-per-pixel or two-cycle-per-pixel pipeline modes, additional display processor 500 commands are available to insure that pipeline synchronization is maintained (e.g., so that the pipeline is emptied of one primitive before the parameters of another primitive take effect). See "Sync Pipe" command set forth in Appendix A.

EXAMPLE DISPLAY PROCESSOR 500 ARCHITECTURE

[0232] Figure 20 shows an example architecture of display processor 500. In this example, display processor 500 includes a command unit 514 with associated RAM 516 and DMA controller 518; an "edge walker"/rasterizer 504; a

RGBAZ pixel stepper 520; a color combiner/level interpreter 508, a blender/fogger 510, a ditherer 522, a coverage evaluator 524, a depth (z) comparator 526, a memory interface 512 and a texture unit 506. In this case, texture unit 506 includes, in addition to texture memory 502, texture steppers 528, a texture coordinate unit 530 and a texture filter unit 532.

[0233] Command unit 514 and DMA controller 518 connect to coprocessor main internal bus 214, and also connect to the signal processor 400 via a private "x" bus 218. Memory interface 512 is a special memory interface for use by display processor 500 primarily to access to the color framebuffer 118a and the z buffer 118b stored within main memory 300 (thus, display processor 500 has access to main memory 300 via memory interface 512 and also via coprocessor internal bus 214).

DMA Controller:

[0234] DMA controller 518 receives DMA commands from signal processor 400 or main processor 100 over bus 214. DMA controller 518 has a number of read/write registers shown in Figures 21A-21C that allow signal processor 400 and/or main processor 100 to specify a start and end address in SP data memory 404 or main memory 300 from which to read a string of graphics display commands (Figure 21A shows a start address register 518A, and Figure 21B shows an end address register 518B). DMA controller 518 reads data over main coprocessor bus 214 if registers 518a, 518b specify a main memory 300 address, and it reads data from the signal processor's data memory 404 over private "x bus" 214 if the registers 518a, 518b specify a data memory 404 address. DMA controller 518 also includes a further register (register 518C shown in Figure 21C) that contains the current address DMA controller 518 is reading from. In this example, DMA controller 518 is uni-directional -- that is, it can only write from bus 214 into RAM 516. Thus, DMA controller 518 is used in this example for reading from signal processor 400 or main memory 300. In this example, display processor 500 obtains data for its texture memory 502 by passing texture load commands to command unit 514 and using memory interface 512 to perform those commands.

Command Unit:

[0235] Command unit 514 retains much of the current state information pertaining to display processor 500 (e.g., mode and other selections specified by "set commands"), and outputs attributes and command control signals to specify and determine the operation of the rest of display processor 500. Command unit 514 includes some additional registers that may be accessed by main processor 100 (or signal processor 400) via coprocessor bus 214. These additional registers, which are mapped into the address space-of main processor 100, permit the main processor to control and monitor display processor 500.

[0236] For example, command unit 514 includes a status/command register 534 shown in Figure 21D that acts as a status register when read by main processor 100 and acts as a command register when the main processor writes to it. When reading this register 534, main processor 100 can determine whether display processor 500 is occupied performing a DMA operation reading from signal processor data memory 404 (field 536(1); whether the display processor is stalled waiting for access to main memory 300 (field 536(2); whether the display processor pipeline is being flushed (field 536(3); whether the display processor graphics dock is started (field 536(4); whether texture memory 502 is busy (field 536(5); whether the display processor pipeline is busy (field 536(6); whether command unit 514 is busy (field 536(7); whether the command buffer RAM 516 is ready to accept new inputs (field 536(8); whether DMA controller 518 is busy (field 536(9); and whether the start and end addresses and registers 518a and 518b respectively valid (fields 536(10), 536(11). When writing to this same register 534, main processor 100 (or signal processor 400) can clear an X-bus DMA operation from the signal processor 400 (field 538 (1); begin an X-bus DMA operation from signal processor data memory 404 (field 538(2); start or stop the display process (fields 538(3), 538(4); start or stop a pipeline flushing operation (fields 538 (5), 538(6); clear a texture memory address counter 540 shown in Figure 21H (field 538 (7); clear a pipeline busy counter 542 shown in Figure 21F (field 538(8); clear a command counter 544 used to index command buffer RAM 516 (field 538(9) (the counter 544 is shown in Figure 21G); and clear a dock counter 546 (see Figure 21E) used to count clock cycles (field 538(10).

[0237] As mentioned above, the clock count, buffer count, pipeline count and texture memory count can all be read directly from registers 540-546 (see Figures 21E-21H). In addition, main processor 100 or signal processor 400 can read and control the BIST operation pertaining to texture memory 502 (see BIST status/control register 548 shown in Figure 211), and can also enable in control testing of memory interface 512 by manipulating mem span test registers 549(a), 549(b) and 549(c) shown in Figure 21J.

[0238] Referring back to Figure 20, once one or more commands have been loaded into command unit buffer ram 518 and display processor 500 has been started, command unit 514 begins reading and processing each command sequentially. The repertoire of commands display processor 500 understands are set forth in Appendix A. Hardware (e.g., logic, gate arrays and the like) within display processor 500 directly interpret the graphics display commands

within RAM 516. In this example, display processor 500 has no ability to branch or jump in traversing this list of commands. Rather, display processor 500 in this example is a sequential state machine that accepts each new command as an input in strict sequence and alters its states and outputs in response to the command.

[0239] Display processor 500 halts if its command buffer RAM 516 is empty (i.e., it has processed all of the commands in the buffer, which buffer acts as a FIFO). Main processor 100 or signal processor 400 can determine if display processor 500 has halted by reading display processor status register 534 and may, if desired, pass the display processor a command that stalls the display processor temporarily (see "Sync Full" command in Appendix A).

Edgewalker and Steppers:

[0240] Edgewalker 504 shown in Figure 20 performs the rasterize process 550 shown in Figure 18. In this example, edgewalker 504 receives the edge coefficients, shade coefficients, texture coefficients and z buffer coefficients specified in a "triangle command" (see Appendix A specifying a particular primitive open line, triangle or rectangle), and outputs "span" values from which the following attributes for each pixel enclosed within the primitive can be derived:

- screen x, y location
- z depth for z buffer purposes
- RGBA color information
- s/w, t, w, 1/w texture coordinates, level-of-detail for texture index, perspective correction, and mipmapping (these are commonly referred to s, t, w, 1)
- coverage value (pixels on the edge of a primitive have partial coverage values, whereas pixels within the interior of a primitive are full).

[0241] Edgewalker 504 sends the parameters for a line of pixels across the primitive (a "span") to the pipeline hardware downstream for other computations. In particular, texture steppers 528 and RGBAZ steppers 520 receive the "span" information specified by edgewalker 504, and step sequentially along each pixel in the horizontal line (in the view plane coordinate system) of the "span" to derive the individual texture coordinates and RGBAZ values for each individual pixel in the span.

[0242] The RGBAZ stepper 520 may also perform a "scissoring" operation on triangle primitives (this does not work for rectangles in this example) to efficiently eliminate portions of triangle primitives extending outside of a view plane scissoring rectangle. Scissoring is commonly used to eliminate running performance-intensive clipping operations on signal processor 400. Scissoring is similar in concept to clipping, but whereas clipping is performed in the 3-D coordinate system, scissoring is performed in the 2-D coordinate system of the viewing plane. Scissoring by steppers 520, 528 is invoked by sending display processor 500 a "set scissor" command (see Appendix A).

[0243] As mentioned above, steppers 520 produces color and alpha information for each pixel within the "span" defined by edgewalker 504. Similarly, texture steppers 528 produces texture coordinate values (s, t, w) for each pixel within the span. Steppers 520, 528 operate in a synchronized fashion so that texture unit 506 outputs a mapped texture value for a pixel to color combiner 58 at the same time that the RGBAZ steppers 520 output a color value for the same pixel based on primitive color, shading, lighting, etc.

Texture Unit:

[0244] Texture unit 506 in this example takes the texture coordinates s, t, w and level-of-detail values for a pixel (as mentioned above, texture steppers 528 derive these values for each individual pixel based upon "span" information provided by edgewalker 504), and fetches appropriate texture information from onboard texture memory 502 for mapping onto the pixel. In this example, the four nearest texels to the screen pixel are fetched from texture memory 502, and these four texel values are used for mapping purposes. Video game program 108 can manipulate texture states such as texture image types and formats, how and where to load texture images, and texture sampling attributes.

[0245] Texture coordinate unit 530 computes appropriate texture coordinates for mapping texture stored within texture memory 502 onto the primitive being rendered. Since the 2-dimensional textures stored in texture memory 502 are square or rectangular images that must be mapped onto triangles of various sizes, the texture coordinate in 530 must select appropriate texels within the texture to map onto pixels in the primitive to avoid distorting the texture. See OpenGL Programming Guide at 278.

[0246] Texture coordinate unit 530 computes a mapping between the inputted pixel texture coordinates and four texels within the appropriate texture stored in texture memory 502. Texture coordinate unit 530 then addresses the texture memory 502 appropriately to retrieve these four texels. The four texel values are passed to the texture filter unit 532. Texture filter 532 takes the four texels retrieved from texture memory 502 and produces a simple bilinear-filtered texel. Texture filter 532 in this example can perform three. types of filter operations: point sampling, box filtering,

and bilinear interpolation. Point sampling selects the nearest texel to the screen pixel. In the special case where the screen pixel is always the center of four texels, the box filter can be used. In the case of the typical 3-D, arbitrarily rotated polygon, bilinear filtering is generally the best choice available. For hardware cost reduction, display processor texture filter unit 532 does not implement a true bilinear filter. Instead, it linearly interpolates the three nearest texels to produce the result pixels. This has a natural triangulation bias which is not noticeable in normal texture images but may be noticed in regular pattem images. This artifact can be eliminated by prefiltering the texture image with a wider filter. The type of filtering performed by texture filter unit 532 is set using parameters in the "set modes" display command (see Appendix A).

TEXTURE MEMORY 502

[0247] Display processor 500 treats texture memory 502 as a general-purpose texture memory. In this example, texture memory 502 is divided into four simultaneously accessible banks, giving output of four texels per clock cycle. Video game program 58 can load varying-sized textures with different formats anywhere in the texture memory 502. Texture coordinate unit 530 maintains eight texture tile descriptors that describe the location of texture images within texture memory 502, the format of each texture, and its sampling parameters. This allows display processor 500 to access as many as eight different texture tiles at a time (more than eight texture tiles can be loaded into the texture memory, but only eight tiles are accessible at any time).

[0248] Figure 22 shows an example of the texture tile descriptors and their relationship to texture tiles stored in texture memory 502. In this particular example shown in Figure 22, eight different texture tiles 802 are stored within texture memory 502. Each texture tile 802 has an associated texture tile descriptor block 804 (as discussed above, display processor 500 maintains up to eight descriptors 804 corresponding to eight texture tiles stored within texture memory 502). The texture descriptors contain information specified by a "set tile" command (see appendix A). For example, these texture tile descriptors specify the image data format (RGBA, YUV, color index mode, etc.), the size of each pixel/texel color element (four, eight, sixteen, thirty-two bits), the size of the tile line in 64-bit words, the starting address of the tile in texture memory 502, a palette number for 4-bit color indexed texels, clamp and mirror enables for each of the S and T directions, masks for wrapping/mirroring in each of S and T directions, level of detail shifts for each of S and T addresses. These descriptors 804 are used by texture coordinate unit 530 to calculate addresses of texels within the texture memory 502.

Texture Coordinate Unit:

[0249] Figure 23 shows a more detailed example of the processing performed by texture coordinate unit 530. Figure 23 shows the various tile descriptors 804 being applied as inputs to texture coordinate unit 530. Figure 23 also shows that texture coordinate unit 530 receives the primitive tile/level/texture coordinates for the current pixel from texture steppers 528. Texture coordinate unit 530 additionally receives mode control signals from command unit 514 based, for example, on the "set other mode" and "set texture image' commands (see Appendix A). Based on all of this input information, texture coordinate unit 530 calculates which tile descriptor 804 to use for this primitive, and converts the inputted texture image coordinates to tile-relative coordinates which the texture coordinate unit wraps, mirrors and/or clamps as specified by the tile descriptor 804. Texture coordinate unit 530 then generates an offset into texture memory 502 based on these tile coordinates. The texture coordinate unit 530 in this example can address 2 x 2 regions of texels in one or two cycle mode, or 4 x 1 regions in copy mode. Texture coordinate unit 530 also generates S/T/L fraction values that are used to bi-linearly or tri-linearly interpolate the texels.

[0250] Figure 24 is a detailed diagram of texture coordinate unit 530 and texture memory unit 502. As shown in Figure 24, the incoming s, t, w texture coordinates are inputted into a perspective correction block 566 which provides a perspective correction based on w when perspective correction is enabled. The perspective-corrected s, t values are then provided to a level-of-detail or precision shift block 568 which shifts the texture coordinates after perspective divide (e.g., for MIP mapping and possibly for precision reasons). A block 570 then converts the shifted texture coordinates to tile coordinates, providing fractional values to the texture filter unit 532. These tile coordinate values are then clamped, wrapped and/or mirrored by block 572 based on the current texture mode parameters of display processor 500. Meanwhile, the perspective-corrected texture coordinates provided by perspective correction block 566 are also provided to a level of detail block 574 which, when level of detail calculations are enabled, calculates a tile descriptor index into a tile descriptor memory 576 and also calculates a level of detail fractional value for interpolation by the color combiner 508. The tile descriptors 804 are stored in tile descriptor memory 576, and are retrieved and outputted to a memory conversion block 578 which conversion block also receives the adjusted texture coordinate values of block 572. Address conversion block 578 converts the adjusted texture coordinate values into texture memory unit addresses based on current tile size, format and other parameters as specified by the tile descriptor 804. Address conversion block 578 outputs the texel address to texture memory unit 502. The texture memory unit 502 also receives additional

parameters which arc used, for example, if the texture is color indexed. Texture memory unit 502 outputs four texel values to texture filter unit 532 for filtering as discussed above.

Texture Memory Loading:

[0251] Texture memory unit 502 includes a four kilobyte random access memory onboard coprocessor 200. Because texturing requires a large amount of random accesses with consistent access time, it is impractical to texture directly from main memory 300 in this example. The approach taken is to cache up to four kilobytes of an image in on-chip, high-speed texture memory 502. All primitives can be textured using the contents of texture memory 502.

[0252] In order to use texture memory 502, video game program 108 must load a texture tile into the texture memory and then load the associated descriptor 804 into tile descriptor 576. The "load tile" command (see Appendix A) is used to load a tile into texture memory 502, and a "set tile" and "set tile size" command are used to load corresponding tile descriptor blocks 804 into tile descriptor memory 576. In addition, a "Load Tlut" command (see Appendix A) can be used to load a color lookup table into texture memory 502 for use by color indexed textures.

[0253] Physically, texture memory 502 is organized in four banks, each comprising 256 16-bit wide words, each bank having a low half and a high half. This organization can be used to store 4-bit textures (twenty texels per row), 8-bit textures (ten texels per row), 16-bit textures (six texels per row), 16-bit YUV textures (twelve texels per row), and 32-bit textures (six texels per row). In addition, texture unit 506 in this example supports a color-indexed texture mode in which the high half of texture memory 502 is used to store a color lookup table and the low half of the texture memory is used to store 4-bit or 8-bit color indexed textures. This organization is shown in Figure 25. In this Figure 25 example, a color indexed texture tile 580 is stored in a low half 502(L) of texture memory 502, and a corresponding color lookup table 582 is stored in the upper half 502(H) of the texture memory.

[0254] Figure 26 shows a more detailed depiction of a particular texture memory color indexed mode, in which the color lookup table 582 is divided into four palette banks 584 or tables, each having, for example, sixteen entries, each entry being 16-bits wide. The color lookup table may represent color in 16-bit RGBA format, or in 16-bit IA format. Since four texels are addressed simultaneously, there are four (usually identical) lookup tables 484 stored in the upper half of texture memory 502. As mentioned above, these lookup tables are loaded using the "load Tlut" command shown in Appendix A.

[0255] Display processor 500 supports another color-indexed texture mode in which each texel in the lower half of texture memory 502 comprises eight bits-and therefore can directly access any one of the 256 locations in the upper half 502(H) of texture memory 502. Thus, 8-bit color-indexed textures do not use the palette number of the tile, since they address the whole 256-element lookup table directly. It is not necessary to use the entire upper half of texture memory 502 for a lookup table when using 8-bit color-indexed textures. For example; if less than eight of the bits of the 8-bit color-indexed texture tile is being used for color lookup, only a portion of color memory upper half 502(H) is required to store the lookup table—and the remainder of the upper half of the texture memory 502 might thus be used for storing a non-color-indexed texture such as a 4-bit I texture (see Figure 25). Similarly, even when color-indexed texture 580 is stored in the lower half 502(L) of texture memory 502, it is possible to also store non-color-indexed textures in the lower half as well. Thus, color-indexed textures and non-color-indexed textures can be co-resident in texture memory 502.

[0256] The following texture formats and sizes are supported by texture memory 502 and texture coordinate unit 530:

Texture Format and Sizes				
Type	4-bit	8-bit	16-bit	32-bit
RGBA			X	X
YUV			X	
Color Index	X	X		
Intensity Alpha (IA)	X	X	X	
Intensity (I)	X	X		

[0257] In this example, texture unit 506 will, unless explicitly told otherwise, change a tile descriptor 804 or a texture tile 802 immediately upon loading—even if it is still being used for texture mapping of a previous primitive. Texture loads after primitive rendering should be preceded by a "sync load" command and tile descriptor attribute changes should be preceded by a "sync tile" command to ensure that the texture tile and tile descriptor state of texture unit 506 does not change before the last primitive is completely finished processing (see Appendix A for example formats and functions of these commands).

[0258] As mentioned above in connection with the signal processor 400, two special commands ("texture rectangle" and "texture rectangle flip") can be used to map a texture onto a rectangle primitive (see Appendix A). It is possible to use the "texture rectangle" command to copy an image from texture memory 502 into frame buffer 118, for example. See Appendix A.

COLOR COMBINER

[0259] Referring once again to Figure 20, color combiner 508 combines texels outputted by texture unit 506 with stepped RGBA pixel values outputted by RGBAZ steppers 520. Color combiner 508 can take two color values from many sources and linearly interpolate between them. The color combiner 508 performs the equation:

$$newcolor = (A\text{-}B) * C + D$$

Here, A, B, C and D can come from many different sources (note that if D = B, then color combiner 508 performs simple linear interpolation).

[0260] Figure 27 shows possible input selection of a general purpose linear interpolator color combiner 508 for RGB and Alpha color combination in this example. As can be seen in Figure 27, only some of the inputs in the lefthand column come from texture unit 506 or RBGAZ steppers 520. The rest of the inputs are derived from color combiner 508 internal state that can be programmed by sending commands to display processor 500. As discussed above, the "combined color" and "combined Alpha" values provided to color combiner 508 are obtained from the RGBAZ steppers 520, and the texel color and texture Alpha are obtained from texture unit 506 (two texel colors and corresponding Alpha values are shown since in two-cycle-per-pixel mode two texels will be provided by texture unit 506 for purposes of mipmapping for example). Additionally, the level of detail fractional input is obtained from Figure 24 block 574, and the primitive level of detail value along with the primitive color and primitive Alpha value may be obtained from a "set primitive color" command sent to display processor 500 (see Appendix A) (the primitive color value/alpha/level of detail fraction value can be used to set a constant polygon face color). Similarly, a shade color and associated Alpha value may be obtained from a "shade coefficient" command (see Appendix A), and an environment color and associated Alpha value may be obtained from a "set environment color" command (see Appendix A) (the environment color/alpha value described above can be used to represent the ambient color of the environment). Two kinds of "set key" commands (one for green/blue, the other for red) are used for green/blue color keying and red color keying respectively—these supplying the appropriate key:center and key:scale inputs to color combiner 508 (see Appendix A). Both the primitive and environment values are programmable and thus can be used as general linear interpolation sources.

[0261] Convert K4 and K5 Inputs to color combiner 508 are specified in this example by the "set convert" command (see Appendix A) that adjust red color coordinates after conversion of texel values from YUV to RGB format (the remainder of the conversion process responsive to this set convert command being performed within texture filter unit 532).

[0262] Figure 28 shows a portion of color combiner 508 used for combining the alpha values shown as inputs in Figure 27. For both the RGB color combine in alpha color combine operations performed by color combiner 508, there are two modes, one for each of the two possible pipeline modes one cycle-per-pixel, and two cycles-per-pixel). In the two-cycle mode, color combiner 508 can perform two linear interpolation arithmetic computations. Typically, the second cycle is used to perform texture and shading color modulation (i.e., the operations color combiner 508 are typically used for exclusively in the one-cycle mode), and the first cycle can be used for another linear interpolation calculation (e.g., level of detail interpolation between two bi-linear filtered texels from two mipmap tiles). Color combiner 508 also performs the "alpha fix-up" operation shown in Figure 29 in this example (see "set key GB" command in Appendix A).

Blender

[0263] As discussed above, blender 510 takes the combined pixel value provided by color combiner 508 can blends them against the frame buffer 118 pixels. Transparency is accomplished by blending against the frame buffer color pixels. Polygon edge antialias is performed, in part, by blender 510 using conditional color blending based on depth (z) range. The blender 510 can also perform fog operations in two-cycle mode.

[0264] Blender 510 can perform different conditional color-blending and z buffer updating, and therefore can handle all of the various types of surfaces shown in Figure 30 (i.e., opaque surfaces, decal surfaces, transparent surfaces, and interpenetrating surfaces).

[0265] An important feature of blender 510 is its participation in the antialias process. Blender 510 conditionally blends or writes pixels into frame buffer 118A based on depth range (see Figure 33 which shows example z buffer formats including a "dz" depth-range field). See U.S. Patent Application Serial No. 08/562,283 of Akeley et al, entitled

"System and Method For Merging Pixel Fragments Based On Depth Range Values", filed on November 22, 1996.

[0266] In this example, video interface 210 applies a spatial filter at frame buffer read-out time to account for surrounding background colors to produce antialias silhouette edges. The antialiasing scheme requires ordered rendering sorted by surface or line types. Here is the rendering order and surface/line types for z buffer antialiasing mode:

1. All opaque surfaces are rendered.
2. All opaque decal surfaces are rendered.
3. All opaque interpenetrating surfaces are rendered.
4. All of the translucent surface and lines are rendered last.

These can be rendered in any order, but proper depth order gives proper transparency.

[0267] The mode blender 510 is controlled, in part by the groups of coefficients specified in the triangle command defining the primitive (see Appendix A). Thus, a primitive can be rendered in a z buffered mode or non-z buffered mode as specified by the triangle command. In addition, the "set other modes" command (see Appendix A) specifies blend mode words for cycle 0 and cycle 1 in addition to specifying "blend masks" and enabling/disabling antialiasing.

[0268] Blender 510 has two internal color registers: fog color and blend color. These values are programmable using the "set fog color" and "set blend color" commands, respectively (see Appendix A). These values can be used for geometry with constant fog or transparency.

[0269] Blender 510 can compare the incoming pixel alpha value with a programmable alpha source to conditionally update frame buffer 118A. This feature can allow complex, outlined, billboard type objects, for example. Besides thresholding against a value, blender 510 in this example can also compare against a dithered value to give a randomized particle effect. See "set other modes" command (Appendix A). Blender 510 can also perform fog operations, either in 1-cycle or 2-cycle mode, Blender 510 uses the stepped z value as a fog coefficient for fog and pipeline color blending.

[0270] Figure 31 shows an example of the overall operations performed by blender 510 in this example. In this particular example, blender 510 can be operated in a mode in which a coverage value produced by coverage evaluator 524 can be used to specify the amount of blending. Coverage evaluator 524 compares the coverage value of the current pixel (provided by edge walker 504) to stored coverage value within frame buffer 118A. As shown in Figure 32 (a depiction of the format of the color information stored for each pixel within color frame buffer 118A), the color of a pixel is represented by 5-bits each of red, green, and blue data and by a 3-bit "coverage" value. This "coverage" value can be used as-is, or multiplied by an alpha value for use as pixel alpha and/or coverage (see "set other modes" command in Appendix A). The "coverage" value nominally specifies how much of a pixel is covered by a particular surface. Thus, the coverage value outputted by edge walker 504 will be 1 for pixels lying entirely within the interior of a primitive, and some value less than 1 for pixels on the edge of the primitive. In this example, blender 510 uses the coverage value for antialiasing. At the time blender 510 blends a primitive edge, it does not know whether the primitive edge is internal to an object formed from multiple primitives or whether the edge is at the outer edge of a represented object. To solve this problem in this example, final blending of opaque edge values is postponed until display time, when the video interface 210 reads out frame buffer 118A for display purposes. Video interface 210 uses this coverage value to interpolate between the pixel color and the colors of neighboring pixels in the frame buffer 118A. In order to accomplish this antialiasing at display time, blender 510 must maintain the coverage value for each pixel within frame buffer 118a, thereby allowing video interface 210 to later determine whether a particular pixel is a silhouette edge or an internal edge of a multi-polygon object.

Memory Interface 512 and Z Buffering

[0271] Memory interface 512 provides an interface between display processor 500 and main memory 300. Memory interface 512 is primarily used during normal display processor 500 operations to access the color frame buffer 118a and the Z buffer 118b, Color frame buffer 118a stores a color value for each pixel on color television screen 60. The pixel format is shown in Figure 32. Z buffer 118b stores a depth value and a depth range value for each color pixel value stored in color frame buffer 118a. An example format for z buffer values is shown in Figure 33. The Z buffer 118b is used primarily by blender 510 to determine whether a newly rendered primitive is in front of or behind a previously rendered primitive (thereby providing hidden surface removal). The "DZ" depth range value shown in Figure 33 may be used to help ascertain whether adjacent texels art part of the same object surface.

[0272] Memory interface 512 can write to main memory 300, read from main memory, or read, modify and write (RMW) locations in the main memory. For RMW operations, memory interface 512, in this example, pre-fetches a row of pixels from frame buffer 118a as soon as edge walker 504 determines the x, y coordinates of the span. Memory interface 512 includes an internal "span buffer" 512a used to store this span or row of pixels. Memory interface 512 provides the appropriate pre-fetched pixel value from span buffer 510a to blender 510 at the appropriate time-thus minimizing the number of accesses to main memory 300. Span buffer 512a is also used to temporarily store blended

(modified) pixel values so that display processor 500 need not access main memory 300 each time a new pixel value is blended. In general, memory interface 512 writes the entire span worth of pixels into main memory 300 as a block all at once.

[0273] Memory interface 512 has enough on-chip RAM to hold several span buffers. This can cause problems, however, if two spans in sequence happen to overlap the same screen area. A parameter "atomic space" in the "Set Other Modes" command (see Appendix A) forces memory interface 512 to write one primitive to frame buffer 118a before starting the next primitive--thereby avoiding this potential problem by adding no cycles after the last span of a primitive is rendered.

[0274] Depth comparator 526 operates in conjunction with z buffer 118b to remove hidden surfaces and to insure the transparent values arc blended properly. Depth comparator 526 compares the z or depth value of the current pixel with the z value currently residing in z buffer 118a for that screen location. At the beginning of the rendering of a new frame, all locations in z buffer 118b are preferably initialized to maximum distance from the viewer (thus, any object will be open "in front of" this initialized value). Generally, each time display processor 500 (is to blend a new pixel into frame buffer 118a), depth comparator 526 compares the depth of the current pixel with the depth residing in that location of z buffer 118b. If the old z buffer value indicates that the previously written pixel is "closer" to the viewer than is the new pixel, the new pixel is discarded (at least for opaque values) and is not written into the frame buffer--thus accomplishing hidden surface removal. If the new pixel is "closer" to the old pixel as indicated by depth comparator 526, then the new pixel value (at least for opaque pixels) may replace the old pixel value in frame buffer 118a--and the corresponding value in z buffer 118b is similarly updated with the z location of the new pixel (see Figure 33A). Transparency blending may be accomplished by blending without updating the z buffer value--but nevertheless reading it first and not blending if the transparent pixel is "behind" an opaque pixel.

Video Interface 210

[0275] Video interface 210 reads the data out of frame buffer 118 and generates the composite, S video RGB video output signals. In this example, video interface 210 also performs anti-aliasing operations, and may also perform filtering to remove truncation caused by the introduction of dithering noise.

[0276] Video interface 210 in this example works in either NTSC or PAL mode, and can display 15-bit or a 24-bit color pixels with or without filtering at both high and low resolutions. The video interface 210 can also scale up a smaller image to fill the screen. The video interface 210 provides 28 different video modes plus additional special features.

[0277] Video interface 210 reads color frame buffer 118a in synchronization with the electron beam scanning the color television screen 60, and provides RGB values for each pixel in digital form to video DAC 144 for conversion into analog video levels in this example. Video interface 210 performs a blending function for opacity values based on coverage (thereby providing an antialiasing function), and also performs a back-filtering operation to remove some of the noise introduced by screen-based dithering.

[0278] Figure 34 is a block diagram of the architecture of video interface 210. In this example, video interface 210 includes the DMA controller 900, a buffer 902, control logic 904, anti-aliasing filters 906a, 906b, error correction blocks 908a, 908b, vertical interpolator (filter) 910, horizontal interpolator (filter) 912, "random" function generator 914, gamma block 916, and bus driver 918.

[0279] DMA controller 900 is connected coprocessor bus 214. DMA controller 900 reads color frame buffer 118a beginning at an "origin" address in the main memory specified by main process 100 (see Figure 35B). DMA controller 900 sequentially reads the pixel color and coverage values (see Figure 32) from frame buffer 118a in synchronism with the line scanning operations of television 58. The pixel values read by DMA controller 900 are processed by the remainder of video interface 210 and are outputted to video DAC 144 for conversion into an analog composite video signal NTSC or PAL format in this example.

[0280] DMA controller 900 in this example provides the color/coverage values it has read from main memory frame buffer 118a, to a RAM buffet 902 for temporary storage. In this example, buffer 902 does not store the pixel color values corresponding to an entire line of television video. Instead, buffer 902 stores a plurality of blocks of pixel data, each block corresponding to a portion of a line of video. Buffer 902 provides "double buffering," i.e., it has sufficient buffers to make some line portions available to filters 906 while other buffers are being written by DMA controller 900.

[0281] In this example, DMA controller 900 accesses, and stores into buffers 902, several of the pixel data corresponding to several horizontally-aligned portions of the video lines to be displayed on television screen 60. Looking at Figure 34A, frame buffer 118a is shown -- for purposes of illustration -- as being organized in a row/column order corresponding to pixels on the television screen (it will be understood that the frame buffer as stored in main memory 300 may actually be stored as a long sequential list of pixel color/coverage values). In this example, DMA controller 900 reads out a block of pixel values corresponding to a particular segment of the current line n of video to be displayed (top shaded block in Figure 34A frame buffer 118a), and also reads out the pixel values corresponding to a horizontally-aligned (on the television screen) line segment of a "next" video line n+1 (i.e., the part of the pixel data representing

the part of the "next" line just beneath the line n). In this particular example, also reads a further block of pixel values from the frame buffer corresponding to the horizontally-aligned line segment of video line n+2.

[0282] Each of these blocks of pixel values is stored in buffer 902. Filters 906a, 906b perform a filtering/anti-abasing operation based on coverage value to interpolate the current line's pixel values with neighboring pixel values (i.e., pixel values that are adjacent with respect to the displayed position on color television screen 60). The anti-aliasing filtering operations performed by filters 906a, 906b are as described in U.S. Patent Application Serial No. 08/539,956 of Van Hook et al, entitled "Antialiasing of Silhouette Edges", filed on October 6, 1996. Briefly, a three-scan-line high neighborhood is color weighted by coverage value in a blending process performed by filter 906. This filtering operation results in smoother, less jagged lines at surface edges by using the pixel coverage value retained in frame buffer 118a (which coverage value indicates what percentage of the pixel is covered by a polygon) to adjust the contribution of that pixel value relative to the contributions of neighboring pixel values in a blending process to produce the current pixel value. "Divot" error correction blocks 908a, 908b correct the outputs of anti-alias filters 906a, 906b for slight artifacts introduced by the anti-aliasing process. In particular, for any pixels on or adjacent to a silhouette edge, the error correction blocks 908 take the median of three adjacent pixels as the color to be displayed in place of the center pixel. This error correction can be enabled or disabled under software control (see Figure 35A), and a video game programmer may wish to disable the error correction since it interacts poorly with decal line rendering modes.

[0283] Anti-aliasing filters 906a, 906b operate in parallel in this example to produce pixel data blocks corresponding to horizontally aligned portions of two successive lines (line n, line n+1) of the image represented by frame buffer 118a. These pixel values are provided to vertical interpolator 910, which performs a linear interpolation between the two image lines to produce an image portion of a single scan line (see Figure 34A). Interpolator 910 interpolates between successive scan lines in order to reduce flicker in interlaced displays. For example, interpolator 910 can add in a contribution from a previous or next successive horizontally-aligned scan line portion to make transitions between successive video scan lines less noticeable -- thereby reducing flicker.

[0284] Additionally, interpolator 910 in this example can perform a vertical scaling function that allows the number of lines displayed on television screen 60 to be different from the number of lines represented by the frame buffer 118a pixel information. In this example, filter 906 scales in the vertical dimension by resampling the pixel data for successive lines of image represented by frame buffer 118a -- thereby allowing television screen 60 to have a different number of lines. This scaling operation (which also accommodates offsetting) is controlled by the values within the video interface Y scale register (see Figure 35N). The ability to scale the television image relative to the digital image size of frame buffer 118a provides additional flexibility. For example, the scaling ability makes it possible for signal processor 400 and display processor 500 to generate a smaller digital image representation in frame buffer 118 -- and yet allow that smaller image to fill the entire television screen 60. Since a smaller frame buffer 118 requires less time to rasterize (i.e., display processor 500 needs to handle fewer spans and fewer pixels per span for a given polygon) and less memory to store, the scaling ability can provide increased performance -- albeit at the cost of a lower resolution image.

[0285] The output of vertical filter 910 in this example is a block of pixel data representing the pixel values for a portion of the video line to be displayed. As shown in Figure 34A, this block of pixel values is provided to horizontal interpolator 912. Horizontal interpolator 912 provides a linear interpolation between neighboring pixel values in order to resample the pixels based on a horizontal scaling factor stored in the X scale register (see Figure 35M). Horizontal interpolator 112 thus provides a horizontal scaling ability, e.g., to convert a smaller number of frame buffer values into a larger number of screen pixels along a horizontal line.

[0286] The output of horizontal interpolator 912 is provided to a Gamma correction circuit 916 that converts linear RGB intensity into non-linear intensity values suitable for composite video generation for the gamma non-linearity of TV monitors. This amounts to taking a square root of the linear color space. The TV monitor effectively raises these color values to a power of 2.2 or 2.4. A "random" function block 914 introduces additional bits of resolution to each of the R, G and B color values in order to "de-dither" (i.e., to compensate for the bit truncation performed by display processor dithering block 522). As shown in Figure 32, one example frame buffer 118 color pixel format in this example provides only five bits of resolution of each R, G and B to conserve storage space within main memory 300. Display processor dithering block 522 may truncate 8-bit RGB color values provided by blender 510 to provide the compressed representation shown in Figure 32. Block 914 can reverse this truncation process to decompress the RGB values to provide 256 different display color levels for each R, G and B. See U.S. Application Serial No. 08/561,584, entitled "Restoration Filter For Truncated Pixels" of Carrol Philip Gossett, filed on November 21, 1996 in the name of Gossett, entitled "Restoration Filter For Truncated Pixels," (Atty Dkt. No. SKGF1452.1690000). This dither filter operation can be turned on and off under software control (see Figure 35A).

Example Video Interface Registers

[0287] There are sixteen control registers for the video interface 210 which control all its functions including sync generation, video rescaling, and anti-aliasing. Figures 35A-35P show the various registers within video interface 210

the can be accessed by main processor 100.

[0288] Figure 35a shows the video interface control register 952. Main processor 100 can write the following values into this register 952 to control the operation of video interface 210:

- Type field 952a specifies pixel data size as blank (no data, no sync), the format shown in Figure 32 (5bits each of RGB and a 3-bit coverage value), or 8/8/8/8 (32-bit color value and 8 bits of coverage);
- Gamma dither enable field 952b turns on and off the addition of some random noise to the lcast significant bits of the video out before the final quantization to 7 bits to eliminate Mach banding artifacts;
- Gamma enable field 952c turns on and off gamma correction;
- Divot enable field 952d turns on and off the divot error correction discussed above;
- video bus clock enable field 952e turns an internal clock on or off;
- Interlace field 952f turns interlacing on and off;
- Test mode field 952g;
- anti-alias mode on/off field 952h;
- diagnostic field 952i;
- pixel advance field 952j; and
- dither filter enable field 952k.

[0289] Figure 35B shows the video interface origin register 954 used to specify the beginning main memory 300 address of frame buffer 118a for read out. In this example, main processor 100 needs to explicitly set this register 954 each time video interface 210 is to read from a new area in main memory 300 (e.g., to read the other half of double buffered frame buffer 118).

[0290] Figure 35c shows the video interface line width register 956, which can be set to specify the number of pixels in each horizontal line. Figure 35d shows the video interface vertical interrupt register 958, which main processor 100 can set with a particular vertical line number so that coprocessor 200 will interrupt the main processor once per frame at the specified vertical line or half line. Figure 35e shows the video interface current line register 960, which specifies the current vertical line when read from by the main processor 100 and clears the vertical line interrupt when written to by the main processor.

[0291] The registers 962-972 shown in Figures 35G-35L are used by main processor to specify detailed composite video timing parameters. For example:

- Figure 35F shows the vertical interface timing register 962 which main processor 100 can write to to specify horizontal sync pulse width, color burst width, vertical sync pulse width, and color burst start timing.

- Figure 35G shows the video interface vertical sync register 964 that main processor 100 may write to specify the number of vertical half-lines per field.

- Figure 35H shows the video interface horizontal sync register 965 which main processor 100 can write to specify the total duration of a line and a horizontal "leap pattern" for PAL.

- Figure 35I shows the video interface h sync leap register 966 specifying two alternate h sync leap parameters for PAL.

- The video interface horizontal video register and vertical video register 968, 970 shown in Figures 35j, 35k, respectively, are used to specify horizontal and vertical video start and end times relative to hsync and vsync.

- The vertical interfaced vertical burst register 972 shown in Figure 35L specifies color burst start and end timing.

[0292] The timing parameters programmable into registers 962-972 can be used to provide compatibility with different kinds of television sets 58. For example, most television sets 58 in the United States use a composite video format known as NTSC, whereas most European television sets use a composite video format known as PAL. These formats differ in terms of their detailed timing parameters (e.g., vertical blanking integral width and location within the signal pattern, horizontal synchronization pulse width, color burst signal pulse width, etc.). Because registers 962-972 control these composite video timing parameters and are programmable by software executing on main processor 100, a programmer of video game 108 can make her program NTSC compatible, PAL compatible, or both (as selected by a user) by including appropriate instructions within the video game program that write appropriate values to registers 962-972. Thus, in this example, coprocessor 200 is compatible NTSC-standard television sets 58, PAL standard compatible television sets -- and even with video formats other than these within a range as specified by the contents of

registers 962-972.

[0293] Vertical interface x and y scale registers 974, 976 (see Figure 35m, 35n, respectively) specify x and y scale up and subpixel offset parameters for horizontal and vertical scaling, as discussed above. Figures 350 and 35p show video interface test data and address registers 978, 980 for diagnostic purposes.

Memory Controller/Interface 212

[0294] As explained above, coprocessor memory interface 212 interfaces main memory 300 with coprocessor internal bus 214. In this example, main memory 300 is accessed over a 9-bit wide bus, and one of the tasks memory interface 212 is responsible for is to buffer successive 9-bit words so they can be more conveniently handled by coprocessor 200. Figure 36 is an example diagram showing the overall architecture of memory controller/interface 212

[0295] In this example, memory interface/controller 212 includes a pair of registers/buffers 1000, 1002, a control block 1004, and a RAM controller block 212b. RAM controller block 212b comprise RAM control circuits designed and specified by Rambus Inc. for controlling main memory 300. Registers 1000, 1002 are used to latch outgoing and incoming data, respectively. Control block 1004 controls the operation of memory interface 212.

Example Memory Controller/Interface Registers

[0296] Figures 37A-37H show example control registers used by main processor 100 to control memory interface 212. Figure 37A shows a read/write mode register specifying operating mode and whether transmit or receive is active (1052). Figure 37B shows a configuration register 1054 that specifies current control input and current control enable. Figure 37C represents a current mode register 1056 that is write only, with any writes to this register updating the current control register. Figure 37D shows a select register 1058 used to select receive or transmit. Figure 37E shows a latency register 1060 used to specify DMA latency/overlap. Figure 37F shows a refresh register 1062 that specifies clean and dirty refresh delay, indicates the current refresh bank, indicates whether refresh is enabled, indicates whether refresh is optimized, and includes a field specifying refresh multi-bank device. Figure 37G shows an error register which in a read mode indicates NACK, ACK and over-range errors, and when written to by main processor 100 clears all error bits. Figure 37H shows a bank status register 1066 which, when read from indicates valid and dirty bits of the current bank, and when written to clears valid and sets dirty bits of the current bank.

CPU INTERFACE

[0297] Figure 38 shows a block diagram of coprocessor CPU interface 202 in this example. CPU interface 202 comprises a FIFO buffer 1102 and a control block 1104. FIFO buffer 1102 provides bidirectional buffering between the CPU SysAD multiplexed address/data bus 102a and the coprocessor multiplexed address/data bus 214D. Control block 1104 receives addresses asserted by the main processor 100 and places them onto the coprocessor address bus 214C. Control block 1104 also receives interrupt signals from the other parts of coprocessor 200, and receives command control signals from the main processor 100 via SysCMD bus 102b.

Example CPU Interface Registers

[0298] Figures 39A-39D show the registers contained within CPU interface 303 in this example. Figure 39 shows a CPU interface status/control register 1152 that controls coprocessor 200 when main processor 100 writes to the register and indicates overall coprocessor status when the main processor reads from the register. Main processor 100 can write to register 1152 to specify initialization code length, set or clear initialization mode, set or dear internal coprocessor bus test mode, clear display processor 400 interrupt, and set or clear main memory register mode. When main processor 100 reads from this register 1152, it can determine initialization code length, initialization mode, internal coprocessor bus test mode, and whether the coprocessor is operating in the main memory register mode.

[0299] Figure 39b shows a version register 1154 that main processor 100 can read from to determine version information pertaining to various components within coprocessor 200.

[0300] Figure 39c shows an interrupt register 1156 that main processor 100 can read from to determine the source of an interrupt it has received from coprocessor 200. In this example, a single line connects between coprocessor 200 and main processor 100 is used for interrupt purposes. Upon receiving a coprocessor interrupt, main processor 100 can read interrupt register (which contains an interrupt vector) to ascertain what component within coprocessor 200 (i.e., signal processor 400, serial interface 204, audio interface 208, video interface 210, parallel interface 206, or display processor 500) cause the interrupt. Figures 39d shows an interrupt mask register 1158 which main processor 100 can write to to set or clear an interrupt mask for any of the interrupts specified in interrupt register 1156, and may read to determine interrupts are masked and which are not.

AUDIO INTERFACE

[0301] Figure 40 shows an overall block diagram architecture of audio interface 208 in this example. Audio interface 208 includes DMA logic 1200, a state machine/controller 1202, an audio clock generator 1204, audio data buffers 1206 and a serializer 1208. In this example, DMA logic 1200 fetches digital audio sample data from audio buffer 114 within main memory 300. DMA logic 1200 writes this audio sample data, 8 bytes at a time, into audio data buffers 1206. There are multiple audio data buffers 1206 arranged in a FIFO so that DMA logic 1200 can be prefetching some audio sample data while serializer 1208 serializes other, previously fetched-and-buffered audio sample data. Thus, buffers 1206 store enough data to supply serializer 1208 between block reads by DMA logic 1200. Since the output rate of serializer 1208 is relatively slow (e.g., on the order of 4 bytes at 50kHz, a single 64-bit buffer 1206b can store enough digitized audio samples to last a relatively long time in terms of real time audio output

[0302] As discussed above, serializer converts the parallel contents of audio buffers 1206 into serial format, and places the resulting serial digital audio data stream onto bus 209 for communication to audio DAC 140. Digital audio bus 209 in this example includes a single serial data line 209a multiplexed between left channel data and right channel data. In this example, serializer 1208 outputs a 16-bit long word for each stereo channel, alternating between the channels. The output bit rate of serializer 1208 is specified by audio dock generator 1204. Audio clock generator 1204 produces an audio clock output on 209b to synchronize audio DAC 140 to the serializer 1208 output bit rate, and produces an audio L/R dock on line 209c specifying whether the current serializer output 1208 is for the left or right stereo channel.

[0303] Figure 40 shows a number of registers and counters used to control audio interface 208. DMA controllers 1200 receives a starting main memory address from an address register 1210. Main processor 100 writes to this address register 1210 (see Figure 41A) to point audio interface 208 to the locations in main memory 300 providing the audio buffer 114 for the current audio to be played. A counter 1212 increments this address for each fetch by DMA controller 1200—thereby sequencing the DMA controller through the entire audio buffer 114. Main process 100 writes the length of audio buffer 114 into a transfer length register 1214 (see Figure 41B). An additional counter 1216 associated with length register 1214 sequences state machine 1202 through an appropriate number of control states corresponding to the length of audio buffer 114. State machine 1202 generates control signals that synchronize the operations of the other parts of audio interface 208 relative to one another. In this example, main processor 100 can enable audio interface 208 to begin fetching data from the main memory 300 by writing to a DMA enable register location 1217 (not shown in Figure 40; see Figure 41C). Main processor 100 may also determine the state of audio interface 200 by reading an audio interface status register 1218 (not shown in Figure 40; see Figure 41D). In this example, state machine 1202 generates a main processor interrupt when it reaches the end of audio buffer 114 as specified by length register 1214, and the main processor 100 can clear this interrupt by writing to the status register 1218 location (see Figure 41D).

[0304] In this example, main processor 100 may also control the rate of the clocking signals generated by audio clock generator 1204. Main processor 100 can program these rates by writing to audio rate registers 1218, 1220 (see Figures 41E, 41F). A counter 1222 may provide a programmable dividing function based on the rate values main processor 100 as written into audio rate registers 1218, 1220.

SERIAL INTERFACE

[0305] Figure 42 shows an overall high level block diagram of serial interface 204 in this example.

[0306] In this example, serial interface 204 moves blocks of data between coprocessor 200 and serial peripheral interface 138. Serial interface 204 can either read a 64-byte data block from serial peripheral interface 138 and transfer it to a specified location in main memory 300 or alternatively, it can read a 64-byte block of data stored in the main memory and transfer it serially to the serial peripheral interface. In this example, serial interface 204 comprises primarily direct memory access logic 1300, control logic 1302, and a parallel/serial converter 1304. Parallel/serial converter 1304 in this example comprises a shift register that converts serial data sent by serial peripheral interface 138 over a read data/acknowledge bus 205a into parallel data for application to latch 1308. The contents of latch 1308 is then applied to coprocessor data bus 214d for writing into main memory 300. Alternatively, in a parallel-to-serial conversion mode, shift register 1304 receives parallel data from the coprocessor data bus 214d via a latch 1310 and converts that data into serial for transmission to serial peripheral interface 138 via a command and write data bus 205b.

[0307] Main processor 100 specifies the address within main memory 300 that serial interface 204 is to read from or write to, by writing this address into an address register 1312 (see Figure 43A). Address register 1312 contents specify the main memory address to be loaded in DMA address counter 1314. Part of the contents of address register 1312 may also be used to specify "address" information within serial peripheral interface 138. Such serial peripheral interface "address" information is loaded into a latch 1316, the contents of which are provided to shift register 1304 for transmission to the serial peripheral interface. This serial peripheral interface "address" information may be used, for

example, to specify a location within the serial peripheral interface 138 (i.e., a boot ROM location 158, a RAM buffer or a status register).

[0308] In this example, serial interface 204 has the ability to place the shift register 1304 parallel output onto the coprocessor address bus 214c via register 1308, a multiplexer 1318, and a latch 1320.

[0309] As shown in Figures 43B, 43C, main processor 100 in this example specifies the direction of serial transfer by writing to a location 1322 or 1324. A write to location 1322 causes serial interface 204 to read a 64-byte data block from the serial peripheral interface 138 and write It to the main memory 300 location specified by address register 1312. A write by main processor 100 to register location 1324 causes serial interface 204 to read a 64-byte block of data from the main memory 300 location specified by address register 1312, and to write the data in serial form to the serial peripheral interface 138.

[0310] Figure 43D shows the serial interface status register 1326. Main processor 100 can read status register 1326 to determine the status of serial interface 204 (e.g., whether the serial interface is busy with a DMA or I/O operation (fields 1328 (1) 1328 (2), respectively); whether there has been a DMA error (field 1328 (3); or whether the serial interface has caused a main processor interrupt (field 1328(4)). Serial interface 204 may generate a main processor interrupt each time it has completed a data transfer to/from serial peripheral interface 138. Main processor 100 can clear the serial interface interrupt by writing to register 1326.

PARALIEL PERIPHERAL INTERFACE

[0311] Figure 44 shows an example block diagram of parallel peripheral interface 206. In this example, parallel interface 206 transfers blocks of data between main memory 300 and storage device 54. Although storage device 54 described above includes only a read-only memory 76 connected to parallel bus 104, system 50 can accommodate different configurations of peripherals for connection to connector 154. For example, two different types of peripheral devices (e.g., a ROM and a RAM) may be connected to peripheral connector 154. Peripheral interface 206 is designed to support communications between two different types of peripheral devices connected to the same parallel bus 104 without requiring any time-consuming reconfiguration between writes.

[0312] Some such peripheral devices may be read-only (e.g., ROM 76), other such peripheral devices may be read/write (e.g., a random access memory or a modem), and still other such peripheral devices could be write only. Peripheral interface 206 supports bi-directional, parallel transfer over parallel bus 104 between connector 154 and main memory 300.

[0313] Parallel peripheral interface 206 in this example includes a DAM controller 1400, a control/register block 1402, and a register file 1404. Register file 1404 buffers blocks of data being transferred by peripheral interface 206 between a peripheral device connected to connector 154 and a block of storage locations within main memory 300. In this example, register file 1404 comprises a small RAM that stores 16 64-bit words. Register file 1404 operates as a FIFO, and is addressed by control/register block 1402. The output of register file 1404 is multiplexed into 16-bit portions by multiplexer 1406. These 16-bit-wide values are latched by a latch 1408 for application to the peripheral device connected to connector 154 via a multiplexed address/data bus 104ad. Data read from the peripheral device via the multiplexed address/data bus 104ad is temporarily stored in a latch 1410 before being applied (via a multiplexer 1412 that also positions the 16-bit read value within an appropriate quarter of a 64-bit word) into register file 1404. Multiplexer 1412 also receives data from coprocessor data bus 214d via latch 1414, and can route this received data into register file 1404 for storage. The register file 1404 output can also be coupled to coprocessor data bus 214d via latch 1416. In this example, the register file 1404 output may also be coupled to the coprocessor address bus 214c via a multiplexer 1418 and a latch 1420.

[0314] Main processor 100 controls the parameters of a DAM transfer performed by peripheral interface 206 by writing parameters into control/register block 1402. For example, main processor 100 can write a starting main memory address into a DRAM address register 1422 (see Figure 45A) and can write a starting address space of a peripheral device connected to connector 154 by writing a peripheral bus address starting address into the peripheral bus register 1424 (see Figure 45B). In this example, main processor 100 specifies the length and direction of transfer by writing to one of registers 1426, 1428 shown in Figures 45C, 45D, respectively, A write to read length register 1426 shown in Figure 45C controls the peripheral interface 206 to transfer in one direction, whereas writing a length value into register 1428 shown in Figure 45D causes the peripheral interface to transfer in the opposite direction. In this example, the main processor 100 can read the status of peripheral interface 206 by reading from a status register location 1430(R) (See Figure 45B). This status register 1430(R) contains fields 1432 indicating DMA transfer in progress (field 1432 (1)), I/O operation in process (field 1432 (#)), an error condition (field 1432(3)). By writing to the same register 1430 (W) location, main processor 100 can clear an interrupt peripheral interface 206 generates when it has completed a requested transfer. Writing to status register location 1430(W) also allows main processor 100 to both clear and interrupt and abort a transfer in progress (see Figure 45A field 1434 (1)).

[0315] Figures 45F, 45G, 45H, 451 show additional registers main processor 100 can write to in order to control

timing and other parameters of the peripheral interface bus 104. These registers permit main processor 100 to configure the bus 104 for particular types of peripheral devices--all under control of software within game program 108. In this example, peripheral interface 44 supports duplicate sets of registers 1436, 1438, 1440 and 1442 shown in Figures 45F-45I- allowing different peripheral bus 104 protocols to be used for different peripheral devices connected simultaneously to the bus without requiring the main processor 100 to re-write the configuration registers each time it request access to a different device. In this example, one set of configuration registers 1436, 1438, 1440 and 1442 are used to configure the bus 104 protocol whenever the peripheral interface 206 accesses a "region 1" address space within the 16-bit peripheral address space, in the other set of register parameters are used whenever the peripheral interface accesses a "region 2" address space within the peripheral bus address range (see Figure 5D memory map). The configurations specified by these two sets of registers are invoked simply by main processor 100 writing to the appropriate region.

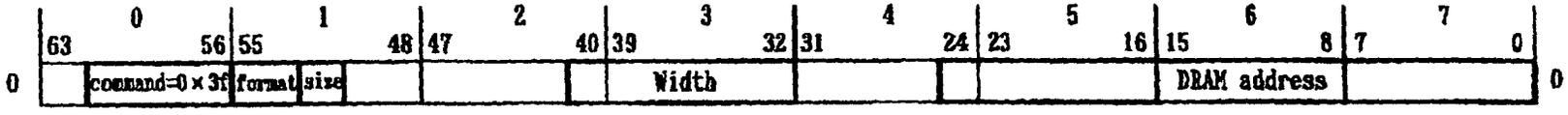
[0316] The various ones of control registers shown in Figures 45A-45I may, in this example, be located within the control/register block 1402 of Figure 44. The configuration values stored in registers 1436, 1438, 1442 are used in this example to control the timing of the access control signals control/register block 1402 produces on bus control line 1404C. A latch 1434 is used to temporarily latch addresses on the co-processor address bus 214C for application to control/register block 1402 (e.g., to select between the various registers). Control/register block 1402 in this example includes appropriate counters and the like to automatically increment DMA addresses.

APPENDIX A: Display Processor 500 Graphic Display Command Example Formats and associated functions of this example.

RDP Command Set

[0317]

Set Color Image



Set_Color_Image Command Format

Field	Word	Bits	Description
Command	0	61-56	Command identifier
format	0	55-53	Image data format,0=rgba,1=yuv,2=Color Indx,3=IA,4=I
size	0	52-51	Size of pixel/texel color element,0=4b,1=8b,2=16b,3=32b
Width	0	41-32	Width of image in pixels, image width=width+1
DRAM adrs	0	25-0	Base address(top left corner) of image in DRAM, in bytes

Legal Color Image Types/Sizes

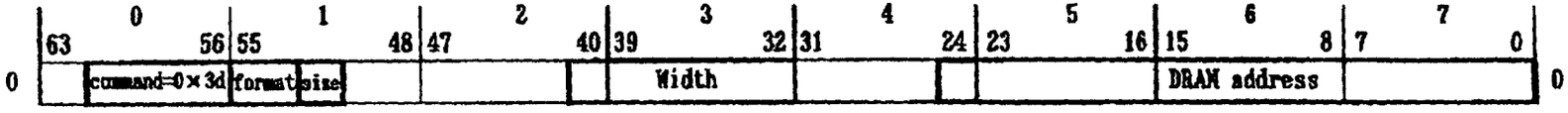
Type	8b	16b	32b
RGBA		✔	✔
YUV			
Clr Indx	✔		

Set_Color_Image Usage Notes:

Read/Modify/Write of 32b color image with depth buffer must be done in two cycle mode.

EP 0 778 536 B1

Set Texture Image



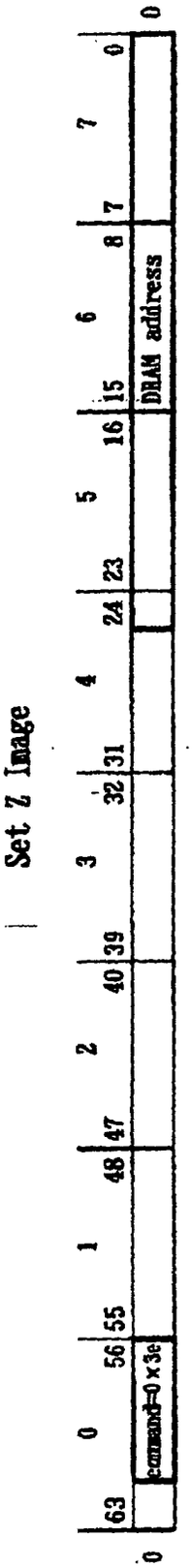
Set_Texture_Image Command Format

Field	Word	Bits	Description
command	0	61-56	Command identifier
format	0	55-53	Image data format, 0=rgba, 1=yuv, 2=Color Indx, 3=IA, 4=I
size	0	52-51	Size of pixel/texel color element, 0=4b, 1=8b, 2=16b, 3=32b
Width	0	41-32	Width of image in pixels, image width=width+1
DRAM adrs	0	25-0	Base address(top left corner) of image in DRAM, in bytes

Legal Texture Image Types/Sizes

Type	4b	8b	16b	32b
RGBA			✔	✔
YUV			✔	
Clr Indx	✔	✔		
IA	✔	✔	✔	
I	✔	✔		

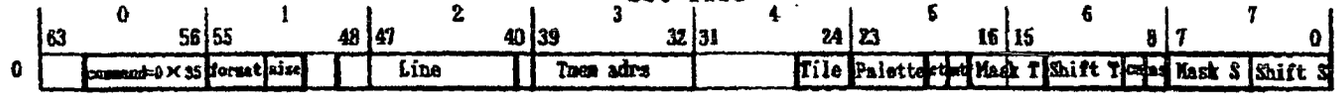
Set Z Image



Set_Mask_Image Command Format

Field	Word	Bits	Description
command	0	61-56	Command identifier
DRAM adrs	0	25-0	Base address(top left corner) of image in DRAM, in bytes

Set Tile



Set_Tile Command Format

Field	Word	Bits	Description
command	0	61-56	Command identifier
format	0	55-53	Image data format, 0=rgba, 1=yuv, 2=Color Indx, 3=IA, 4=I
size	0	52-51	Size of pixel/texel color element, 0=4b, 1=8b, 2=16b, 3=32b, 4=Other
Line	0	49-41	Size of tile line in 64b words, max of 4KB
Tmem Adrs	0	40-32	Starting Tmem address for this tile in words(64b), 4KB range
tile	0	26-24	Tile descriptor index
Palette	0	23-20	Palette number for 4b Color Indexed texels. This number is used as the MS 4b of an 8b index.
ct	0	19	clamp enable for T direction
mt	0	18	mirror enable for T direction
Mask T	0	17-14	Mask for wrapping/mirroring in T direction. If this field is zero then clamp, otherwise pass(mask)LSBs of T adress.
Shift T	0	13-10	Level of Detail shift for T addresses
cs	0	9	clamp enable bit for S direction
ms	0	8	mirror enable bit for S diredtion
Mask S	0	7-4	Mask for wrapping/mirroring in S direction. If this field is zero then clamp, otherwise pass(mask)LSBs of S adress.
Shift S	0	3-0	Level of Detail shift for S address

Set_Tile Usage Notes:

For YUV textures, Tile Line(number of Tmem words per tile line)is Line=(Width+7)>>3(8b texels)because although the image texels are 16bit, the Tmem Y texels are 8 bit and the Tmem UV texels are 16 bit at 1/2 the width in S.

YUV mask and mask/mirror are undefined

No mirroring for 32b RGBA Images.

Wrap on all but YUV images.

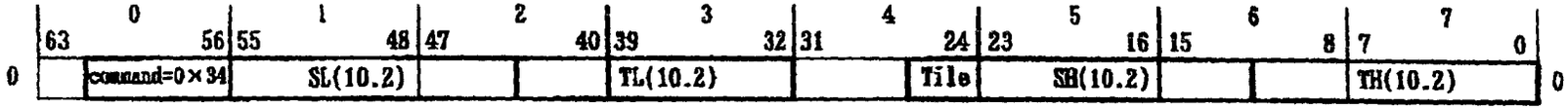
YUV texture images are stored as interleaved 8bit YUYV as 16bits per texel. Software must specify texture coordinates and tile coordinates and sizes which are even numbers in S(lsb=0),so that a pair UV values are avaiable. Software must specify a Tile Base address in the low half of Tmem, and load a tile which fits in the low half(where 2048<Base+W*H) in 8b Y's. Software can't have CI TLUT's and YUV textures coexisting in Tmem.

32b RGBA,YUV Set_Tile type should indicate 16b texels.

Mask=1 means pass bit 0 so minimum mask width is 2 texels

EP 0 778 536 B1

Load Tile



Load_Tile

Field	Word	Bits	Description
command	0	61-56	Command identifier
SL	0	55-44	Low S coordinate of tile in image
TL	0	43-32	Low T coordinate of tile in image
Tile	0	26-24	Tile descriptor index
SH	0	23-12	High S coordinate of tile in image
TH	0	11-0	High T coordinate of tile in image

Load_Tile Usage Notes:

4b textures should be loaded as bytes(they must be byte aligned) using the Set_Image Texture *type* field when loading more than 4k texels. This means the Load_Block parameters will have 8b texel units. The Set_Tile type field can be used to set the proper 4b type. The Set_Tile_Sz can be used to set the proper SL,SH,TL,TH values after the Load_Block. In general, textures can be loaded as one type and used as another type by proper manipulation of the Set_Image_Texture and Set_Tile *format size* fields. Normally, during a load, the tile texel size and image pixel size should match.

YUV SL must be even and SH must be odd, which means the width is even, to provide a UV value at each texel. For bilinear interpolation, tile SH should be adjusted to tile load SH-1, so that clamping occurs at a UV value.

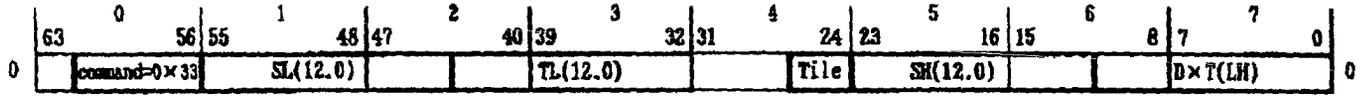
For YUV and 32b RGBA images, software must specify a Tile Base address in the low half of Tmem, and load a tile which fits in the low half (where 2048<Base+W*H in 8b Y's).

For YUV and 32b RGBA images, can't have color index TLUT's in Tmem.

Fractional coordinates for L and H terms must usually be the same. Useful for subpixel coordinate offset in multi-tile operations.

EP 0 778 536 B1

Load Block



Load Block

Field	Word	Bits	Description
Command	0	61-56	Command identifier. Load Block loads a Tmem tile with a single memory "span" from SL, TL to SH,TL. During the tile load, the T coordinate is incremented by DxT every 8 Tmem bytes, in order to perform odd-line swapping and line strides.
SL	0	55-44	Low S coordinate of tile in image
TL	0	43-32	Low T coordinate of tile in image, the two MSBs of this number should be zero (i.e., a 10-bit number)
Tile	0	26-24	Tile descriptor index
SH	0	23-12	High S coordinate of tile in image
DxT	0	11-0	unsigned T increment value

Load_Block Usage Notes:

A ceiling function must be performed on the LH DxT field of Load_Block. That is , if any number has non-zero bits to the right of the 11-bit fraction, then a ceiling operation should be performed on the the number. For example, a 12 texel(16b/texel)wide texture would have a DxT of 1/3. The 11b fraction would be 1/3*2048 or 682 2/3. The ceiling is 683.

The texture image width must be multiples of 8 bytes. For example, a 4bit texel texture must have an image width of n*16.

Each Load_Block command should be followed by Set Tile_Ss command to set the actual tile SH,TH values.

4b textures should be loaded as bytes(they must be byte aligned)using the Set_Image_Texture *type* field when loading more than 4k texels. This means the Load_Block parameters will have 8b texel units. The Set_Tile *type* field can be used to set the proper 4b type after the Load_Block. The Set_Tile_Size can be used to set the proper SL,SH,TL,TH values after the Load_Block. In general,textures can be loaded as one type and used as another type by proper manipulation of the Set_Image_Texture and Set_Tile *format,size* fields.

Normally, during a load, the tile texel size and image pixel size should match.

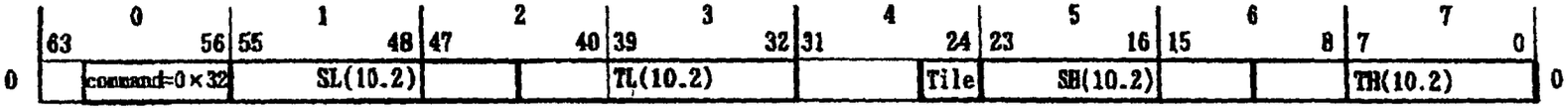
YUV texture images are stored as intereleaved 8bit YUYV as 16 bits per texel. Software must specify texture coordinates and tile coordinates and sizes which are even numbers in S(lsb==0), so that a pair UV values are avaiable. Software must specify a Tile Base address in the low half of Tmem, and load a tile which fits in the low half(where 2048<Base+W*H)in 8b Y's. Software can't have CI TLUT's and YUV textures coexisting in Tmem.

In Load Block, the Tile line is the number of words to skip for each T. That is, zero for a contiguous tile.

When Load_Block is used to load multiple tiles of different widths(such as a mipmap pyramid), the image data in memory must be ordered for Tmem interleaved access. This means that for odd lines(t&1), the two longs(32b)in each double(64b)must be swapped. Since Load_Block lines must consist of an integral number of double words, this is not effected by width. Note that Load_Tile performs this interleave during load, and Load_Block can perform this interleave by computing a T coordinate from DxT in the command. For memory image data which is interleaved, DxT should be zero.

EP 0 778 536 B1

Set Tile Size



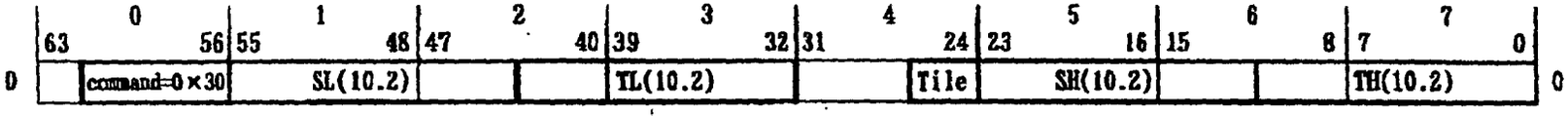
Set_Tile_Size

Field	Word	Bits	Description
command	0	61-56	command identifier
SL	0	55-44	Low S coordinate of tile in image
TL	0	43-32	Low T coordinate of tile in image
Tile	0	26-24	Tile descriptor index
SH	0	23-12	High S coordinate of tile in image
TH	0	11-0	High T coordinate of tile in image

Set_Tile_Size Usage Notes:

For YUV textures SL must be even and SH must be odd(this means the width is an even number of texels). Hardware will clamp to SH-1, that is, the last even texel, in order to have a valid UV at the last texel. If linear interpolation is performed, the max S texture coordinate must be odd, and SH is the max S+2 (an odd SH), in order to supply a valid UV at the max S+1.

Load Tlut



Load_Tlut

Field	Word	Bits	Description
command	0	61-56	command identifier, this command is used to initiate a load from DRAM of a Indexed Texture Lookup Table(TLUT). This table dereferences color indexed texels before texture filtering.
SL	0	55-44	low index into table(0-255), fractional bits should be zero
TL	0	43-32	normally zero
Tile	0	26-24	Tile descriptor index
SH	0	23-12	high index into table(0-255), fractional bits should be zero
TH	0	11-0	normally zero

Load_Tlut Usage Notes:

Use the set_image_texture to define the dram adress, with a 16b type. Use set_tile to define the Tmem address.
Reference that tile in load_tlut. The Texture Coordinate unit should get a D×S of 4 due to quadrication of table when loading.

Tmem Address must be in high half(msb=1)of Tmem.

Tmem address in 64b words.

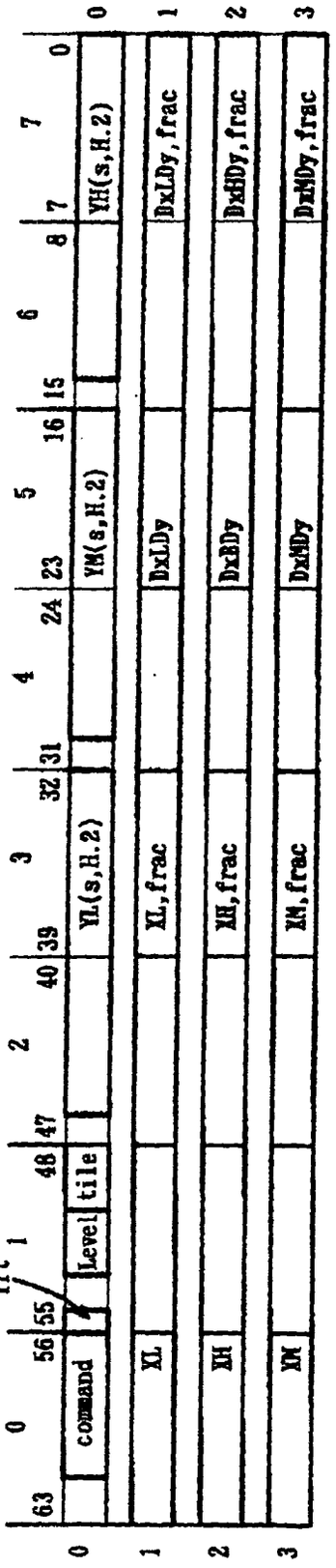
EP 0 778 536 B1

Triangle command of various types are formed by concatonating groups of coefficients as shown below. The order for concatonation is from left to right in the table. The formats for each group of coefficients are shown on the following pages.

Triangle Commands

Command	Edge	Shade	Texture	Zbuffer
Non-Shaded Triangle, 0×08	✓			
Shade Triangle, 0×0c	✓	✓		
Texture Triangle, 0×0a	✓		✓	
Shade, Texture Triangle, 0×0e	✓	✓	✓	
Non-Shaded, ZBuff Triangle, 0×09	✓			✓
Shade, ZBuff Triangle, 0×0d	✓	✓		✓
Texture, ZBuff Triangle, 0×0b	✓		✓	✓
Shade, Texture, ZBuff Triangle, 0×0f	✓	✓	✓	✓

Edge Coefficients



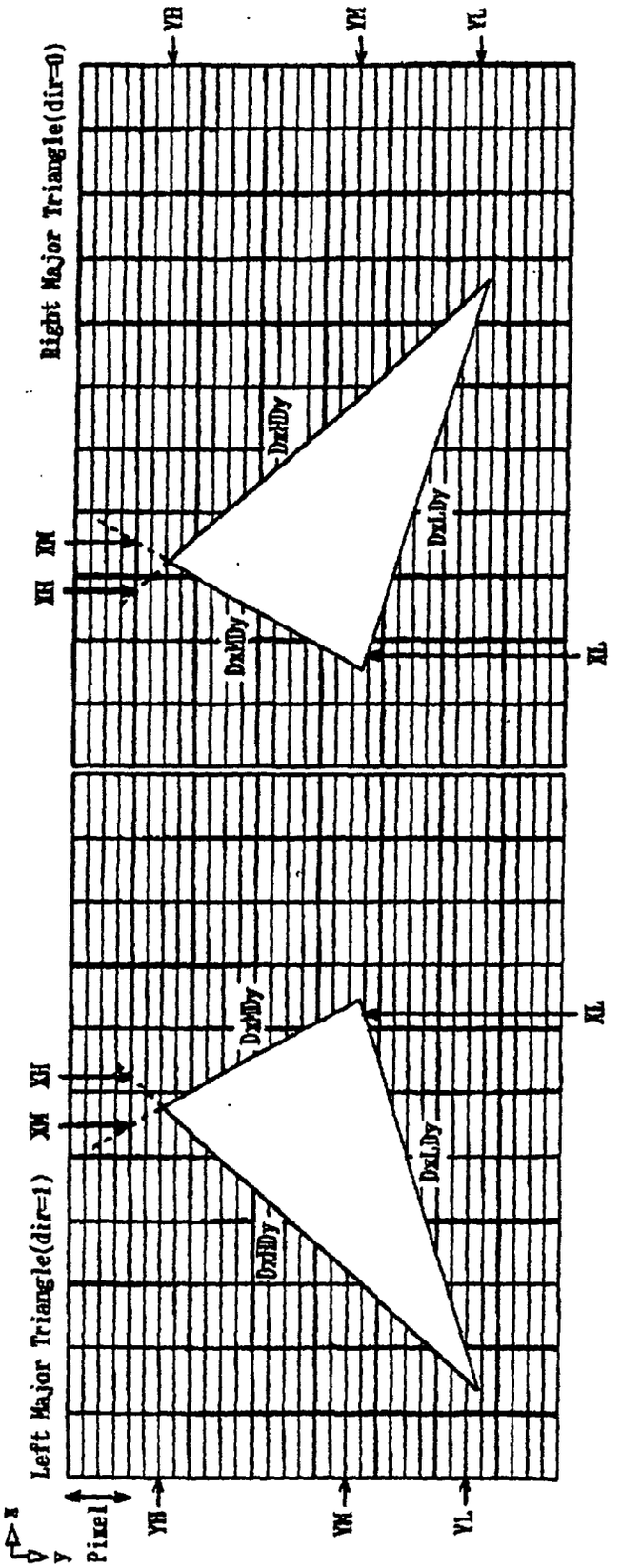
Edge Coefficients

Field	Word	Bits	Description
command	0	61-56	Command identfier
lft	0	55	Left major flag, 1=left major, 0=right major
Level	0	53-51	number of mip-maps minus one
file	0	50-48	Tile descriptor index. Used to reference texture for this primitive.
YL	0	45-32	Y coordinate of low minor edge
YM	0	29-16	Y coordinate of mid minor edge
YH	0	13-0	Y coordinate of major edge
XL	1	63-48	X coordinate of low edge, integer
XL,frac	1	47-32	X coordinate of low edge, fraction
DxLDy	1	31-16	Inverse slope of low edge, integer
DxLDy,frac	1	15-0	Inverse slope of low edge, fraction

Edge Coefficients

Field	Word	Bits	Description
XH	2	63-48	X coordinate of major edge, integer
XH, frac	2	47-32	X coordinate of major edge, fraction
DxHDy	2	31-16	Inverse slope of major edge, integer
DxHDy,frac	2	15-0	Inverse slope of major edge, fraction
XM	3	63-48	X coordinate of middle edge, integer
XM,frac	3	47-32	X coordinate of middle edge, fraction
DxMDy	3	31-16	Inverse slope of middle edge, integer
DxMDy,frac	3	15-0	Inverse slope of middle edge, fraction

The edge coefficients are calculated at specific points on the view screen. The diagram below shows where each term is located. In general, Y terms are calculated to at least subpixel(1/4 pixel) resolution. The XM and XH are calculated where the H and M edges intersect the previous scan line. XL is calculated where the L edge intersects the next subpixel *at or below the mid vertex.*



Shade Coefficients

	63	0	56	55	1	48	47	2	40	39	3	32	31	4	24	23	5	16	15	6	8	7	7	0	
4	Red						Green						Blue						Alpha						4
5	DrDx						DgDx						DbDx						DaDx						5
6	Red Fraction						Green Fraction						Blue Fraction						Alpha Fraction						6
7	DrDx, fraction						DgDx, fraction						DbDx, fraction						DaDx, fraction						7
8	DrDe						DgDe						DbDe						DaDe						8
9	DrDy						DgDy						DbDy						DaDy						9
10	DrDe, fraction						DgDe, fraction						DbDe, fraction						DaDe, fraction						10
11	DrDy, fraction						DgDy, fraction						DbDy, fraction						DaDy, fraction						11

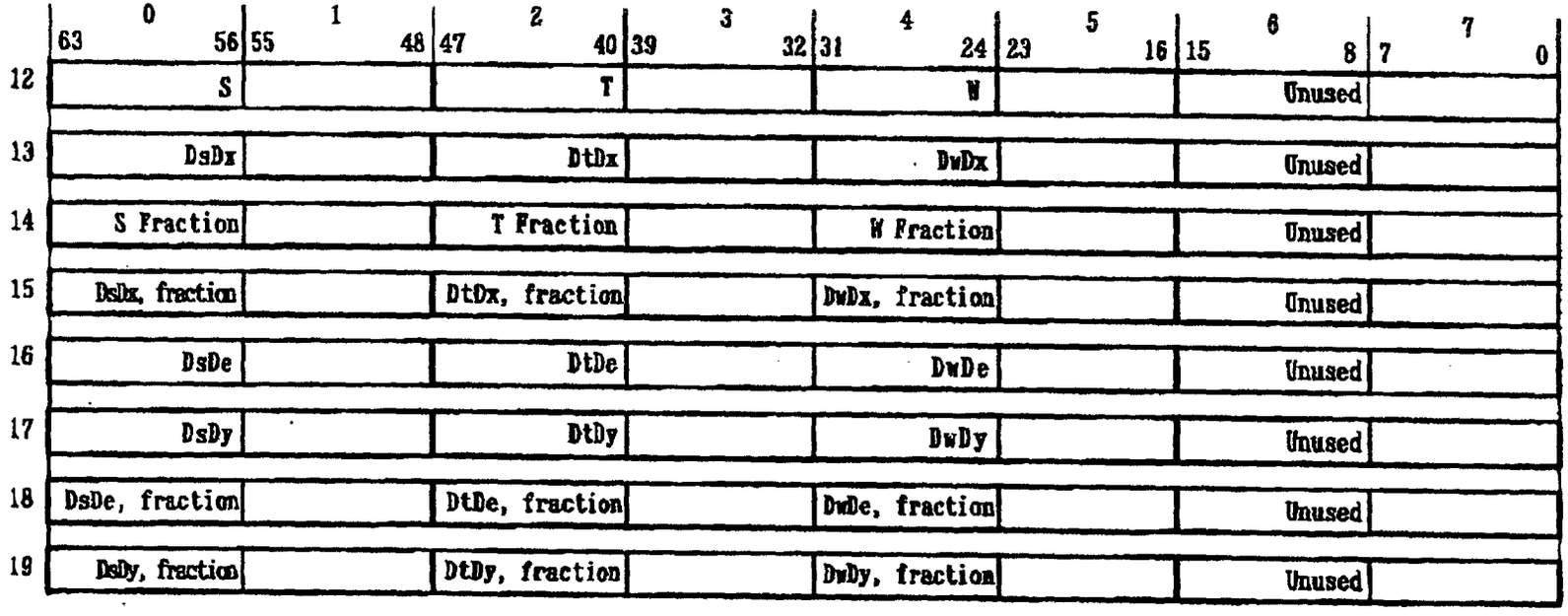
Shade Coefficients

Field	Word	Bits	Description
Red	4	63-48	Red color component, integer
Green	4	47-32	Green color component, integer
Blue	4	31-16	Blue color component, integer
Alpha	4	15-0	Alpha color component, integer
DrDx	5	63-48	Change in red per change in X coordinate, integer
DgDx	5	47-32	Change in green per change in X coordinate, integer

Shade Coefficients

Field	Word	Bits	Description
DbDx	5	31-16	Change in blue per change in X coordinate, integer
DaDx	5	15-0	Change in alpha per change in X coordinate, integer
Red, frac	6	63-48	Red color component, fraction
Green, frac	6	47-32	Green color component, fraction
Blue, frac	6	31-16	Blue color component, fraction
Alpha, frac	6	15-0	Alpha color component, fraction
DrDx, frac	7	63-48	Change in red per change in X coordinate, fractoin
DgDx, frac	7	47-32	Change in green per change in X coordinate, fraction
DbDx, frac	7	31-16	Change in blue per change in X coordintate, fraction
DaDx, frac	7	15-0	Change in alpha per change in X coordinate, fraction
DrDe	8	63-48	Change in red along the edge, integer
DgDe	8	47-32	Change in green along the edge, integer
DbDe	8	31-16	Change in blue along the edge, integer
DaDe	8	15-0	Change in alpha along the edge, integer
DrDy	9	63-48	Change in red per change in Y coordinate, integer
DgDy	9	47-32	Change in green per change in Y coordinate, integer
DbDy	9	31-16	Change in blue per change in Y coordinate, integer
DaDy	9	15-0	Change in alpha per change in Y coordinate, integer
DrDe	10	63-48	Change in red along the edge, fraction
DgDe	10	47-32	Change in green along the edge, fraction
DbDe	10	31-16	Change in blue along the edge, fraction
DaDe	10	15-0	Change in alpha along the edge, fraction
DrDy	11	63-48	Change in red per change in Y coordinate, fraction
DgDy	11	47-32	Change in green per change in Y coordinate, fraction
DbDy	11	31-16	Change in blue per change in Y coordinate, fraction
DaDy	11	15-0	Change in alpha per change in Y coordinate, fraction

Texture Coefficients



Texture Coefficients

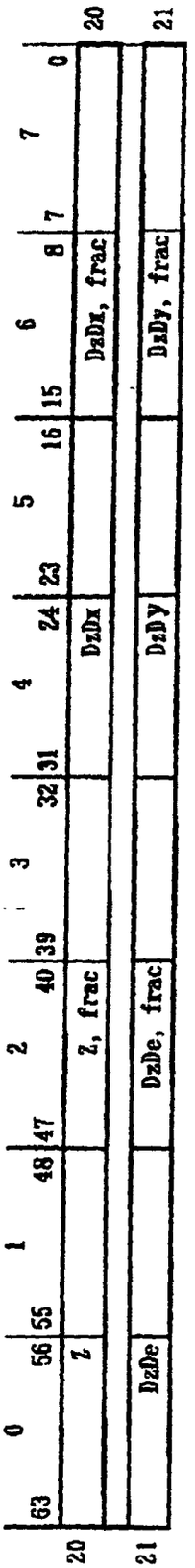
Field	Word	Bits	Description
S	12	63-48	S texture coordinate, integer
T	12	47-32	T texture coordinate, integer
W	12	31-16	Normalized inverse depth, integer
DsDx	13	63-48	Change in S per change in X coordinate, integer
DtDx	13	47-32	Change in T per change in X coordinate, integer
DwDx	13	31-16	Change in W per change in X coordinate, integer

EP 0 778 536 B1

Texture Coefficients

Field	Word	Bits	Description
S, frac	14	63-48	S texture coordinate, fraction
T, frac	14	47-32	T texture coordinate, fraction
W, frac	14	31-16	Normalized inverse depth, fraction
DsDx, frac	15	63-48	Change in S per change in X coordinate, fraction
DtDx, frac	15	47-32	Change in T per change in X coordinate, fraction
DwDx, frac	15	31-16	Change in W per change in X coordinate, fraction
DsDe	16	63-48	Change in S along the edge, integer
DtDe	16	47-32	Change in T along the edge, integer
DwDe	16	31-16	Change in W along the edge, integer
DsDy	17	63-48	Change in S per change in Y coordinate, integer
DtDy	17	47-32	Change in T per change in Y coordinate, integer
DwDy	17	31-16	Change in W per change in Y coordinate, integer
DsDe, frac	18	63-48	Change in S along the edge, fraction
DtDe, frac	18	47-32	Change in T along the edge, fraction
DwDe, frac	18	31-16	Change in W along the edge, fraction
DsDy, frac	19	63-48	Change in S per change in Y coordinate, fraction
DtDy, frac	19	47-32	Change in T per change in Y coordinate, fraction
DwDy, frac	19	31-16	Change in W per change in Y coordinate, fraction

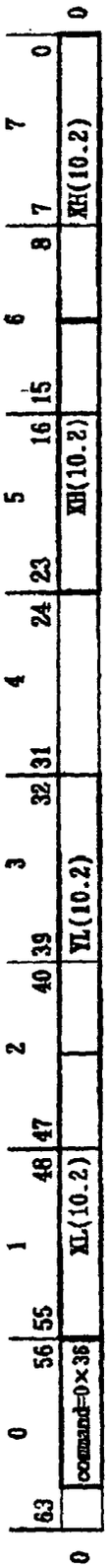
ZBuffer Coefficients



ZBuffer Coefficients

Field	Word	Bits	Description
Z	20	63-48	Inverse Depth, integer
Z, frac	20	47-32	Inverse Depth, fraction
DzDx	20	31-16	Change in Z per change in X coordinate, integer
DzDx, frac	20	15-0	Change in Z per change in X coordinate, fraction
DzDe	21	63-48	Change in Z along major edge, integer
DzDe, frac	21	47-32	Change in Z along major edge, fraction
DzDy	21	31-16	Change in Z per change in Y coordinate, integer
DzDy, frac	23	15-0	Change in Z per change in Y coordinate, fraction

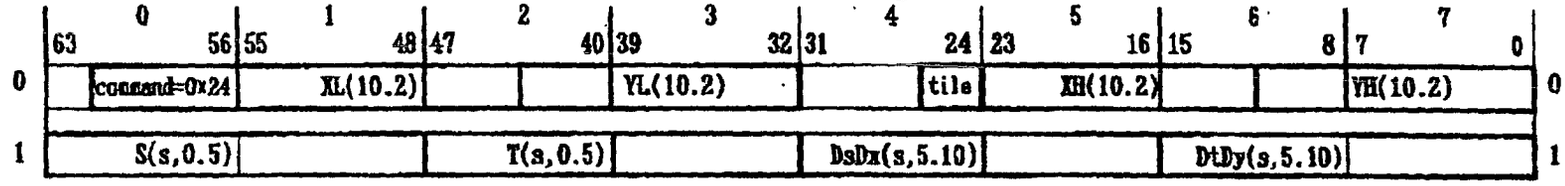
Fill Rectangle



Fill_Rectangle

Field	Word	Bits	Description
command	0	61-56	Command identifier
XL	0	55-44	X coordinate of bottom right corner of rectangle
YL	0	43-32	Y coordinate of bottom right corner of rectangle
XH	0	23-12	X coordinate of top left corner of rectangle
YH	0	11-0	Y coordinate of top left corner of rectangle

Texture Rectangle



Texture_Rectangle

Field	Word	Bits	Description
command	0	61-56	Command identifier
XH	0	55-44	X coordinate of top left corner of rectangle
YH	0	43-32	Y coordinate of top left corner of rectangle
tile	0	26-24	Title descriptor index
XL	0	23-12	X coordinate of bottom right corner of rectangle
YL	0	11-0	Y coordinate of bottom right corner of rectangle
S	1	63-48	S texture coordinate at top left corner of rectangle
T	1	47-32	T texture coordinate at top left corner of rectangle
DsDx	1	31-16	Change in S per change in X coordinate
DtDy	1	15-0	Change in T per change in Y coordinate

Texture_Rectangle Usage Notes:

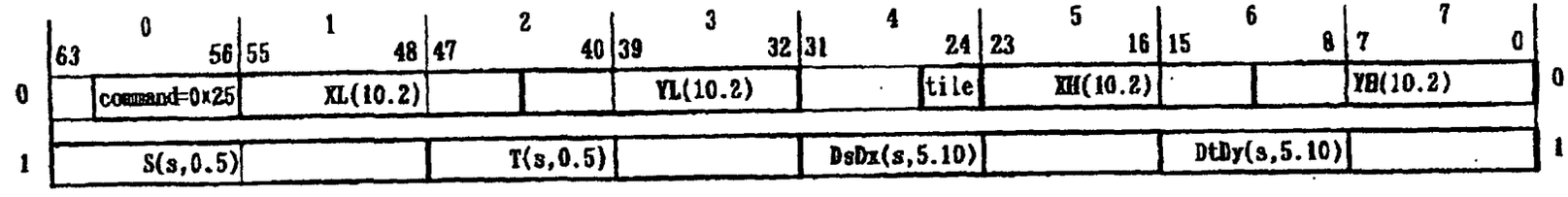
To copy an image, set cycle_type to "copy" in Set_Other_Modes, and set the combination of Set_Tile's "shift S" and DsDx to step by 4 texels. No texture filtering, color combining, or blending operations are available for copied texels. Write enables may be generated using threshold compares of the alpha channel of each copied texel.

Texture Rectangle and Copy mode:

No Z-buffer or anti-aliasing in copy mode.
4b, YUV, and 32b RGBA textures cannot be directly copied.
4b images are expanded to 8b images before copying.
4b and 8b images can be copied to an 8b color image only.
16b image can be copied to a 16b color image only.
Note that 4b, YUV, and 32b RGBA images can be copied by using 8b or 16b color images and correctly scaling image coordinates and width.

EP 0 778 536 B1

Texture Rectangle Flip

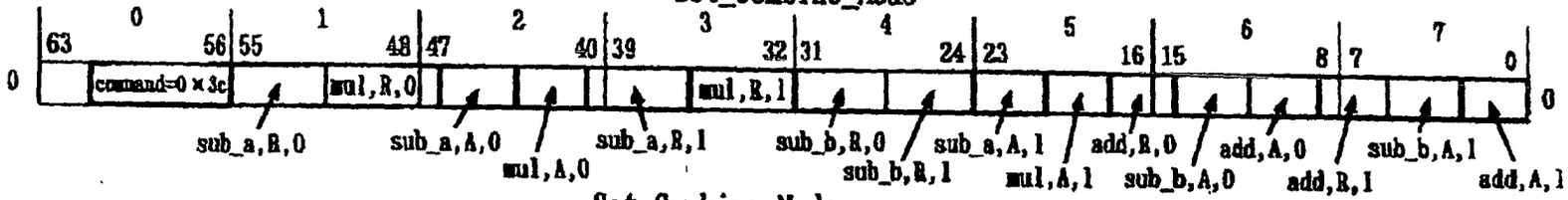


Texture_Rectangle Flip

Field	Word	Bits	Description
command	0	61-56	Command identifier, same as Texture Rectangle except hardware swaps S/T and DsDx/DtDy.
XH	0	55-44	X coordinate of top left corner of rectangle
YH	0	43-32	Y coordinate of top left corner of rectangle
tile	0	26-24	Tile descriptor index
XL	0	23-12	X coordinate of bottom right corner of rectangle
YL	0	11-0	Y coordinate of bottom right corner of rectangle
S	1	63-48	S texture coordinate at top left corner of rectangle
T	1	47-32	T texture coordinate at top left corner of rectangle
DsDx	1	31-16	Change in S per change in X coordinate
DtDy	1	15-0	Change in T per change in Y coordinate

EP 0 778 536 B1

Set_Combine_Mode



Set_Combine_Mode

Field	Word	Bits	Description
command	0	61-56	Command identifier
sub_a,R,0	0	55-52	sub_A input, RGB components, cycle 0
mul,R,0	0	51-47	multiply input, RGB components, cycle 0
sub_a,A,0	0	46-44	sub_A input, Alpha component,cycle 0
mul,A,0	0	43-41	multiply input, Alpha component, cycle 0
sub_a,R,1	0	40-37	sub_A input, RGB components, cycle 1
mul,R,1	0	36-32	multiply input, RGB components, cycle 1
sub_b,R,0	0	31-28	sub_B input, RGB components, cycle 0
sub_b,R,1	0	27-24	sub_B input, RGB components, cycle 1
sub_a,A,1	0	23-21	sub_A input, Alpha component, cycle 1
mul,A,1	0	20-18	multiply input, Alpha component, cycle 1
add R,0	0	17-15	adder input, RGB components, cycle 0
sub_b,A,0	0	14-12	sub_B input, Alpha component, cycle 0
add,A,0	0	11-9	adder input, Alpha components, cycle 0
add,R,1	0	8-6	adder input, RGB components, cycle 1
sub_b,A,1	0	5-3	sub_B input, Alpha components, cycle 1
add,A,1	0	2-0	adder input, Alpha components, cycle 1

Set_Combine_Mode Usage Notes:

The Color Combiner implements the equation:$(A-B)^{*}C+D$ on each color. RGB and Alpha channels have separate mux selects. In addition, there are separate mux selects for cycle 0 and cycle 1. If the RDP is configured for one cycle mode, set the cycle 0 and cycle 1 mux selects to the same value.

EP 0 778 536 B1

Set_Other_Modes

Set_Other_Modes

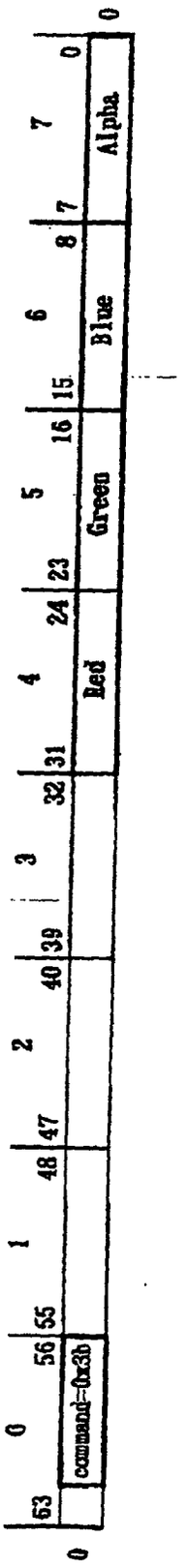
Field	Word	Bits	Description
command	0	61:56	Command identifier
k:atomic_prim	0	55	Force primitive to be written to frame buffer before read of following primitive.
j:Reserved	0	54	This mode bit is not currently used but may be in the future.
i:cycle_type	0	53:52	Display pipeline cycle control mode:0=1 cycle, 1=2 cycle, 2=Copy, 3=Fill
h:persp_tex_en	0	51	enable perspective correction on texture
g:detail_tex_en	0	50	enable detail texture
f:sharpen_tex_en	0	49	enable sharpened texture
e:tex_lod_en	0	48	enable texture Level of Detail (LOD)
d:en_tlut	0	47	enable lookup of texel values from TLUT. Meaningful if texture type is index, tile is in low Tmem, TLUT is in high Tmem, and color image is RGB.
c:Tlut type	0	46	Type of texels in table, 0=16b RGBA(5/5/5/1), 1=IA(8/8)
b:sample_type	0	45	determines how textures are sampled: 0=1×1(Point Sample), 1=2×2. Note that copy (point sample 4 horizontally adjacent texels) mode is indicated by cycle_type.
a:mid_texel	0	44	indicates texture filter should do a 2x2 half texel interpolation, primarily used for MPEG motion compensation processing.
Z:bi_lerp_0	0	43	1=bi_lerp, 0=color convert operation in texture filter. Used in cycle 0
Y:bi_lerp_1	0	42	1=bi_lerp, 0=color convert operation in texture filter. Used in cycle 1
X:convert_one	0	41	Color convert texel that was the output of the texture filter on cycle 0, used to qualify bi_lerp_1
W:key_en	0	40	Enables chroma keying

Set_Other_Modes

Field	Word	Bits	Description
V2:rgb_dither_sel	0	39:38	0=magic square matrix(preferred if filtered) 1=" standard" bayer matrix(preferred if not filtered) 2=noise(as before) 3=no dither
V1:alpha_dither_sel	0	37:36	0=pattern 1=pattern 2=noise 3=no dither
Reserved	0	35:32	Reserved for future use, default value is 0xF
V:b,m1a,0	0	31:30	Blend modeword, multiply 1a input select, cycle 0
U:b,m1a,1	0	29:28	Blend modeword, multiply 1a input select, cycle 1
T:b,m1b,0	0	27:26	Blend modeword, multiply 1b input select, cycle 0
S:b,m1b,1	0	25:24	Blend modeword, multiply 1b input select, cycle 1
R:b,m2a,0	0	23:22	Blend modeword, multiply 2a input select, cycle 0
Q:b,m2a,1	0	21:20	Blend modeword, multiply 2a input select, cycle 1
P:b,m2b,0	0	19:18	Blend modeword, multiply 2b input select, cycle 0
O:b,m2b,1	0	17:16	Blend modeword, multiply 2b input select, cycle 1
N:reserved	0	15	This mode bit is not currently used, but may be in the future
M:force_blend	0	14	force blend enable
L:alpha_cvg_select	0	13	use cvg(or cvg*alpha)for pixel alpha
K:cvg_times_alpha	0	12	use cvg times alpha for pixel alpha and coverage
J:z_mode[1:0]	0	11:10	0:opaque, 1:interpenetrating, 2:transparent, 3:decal
I:cvg_dest[1:0]	0	9:8	0:clamp(normal), 1:wrap(as assume full cvg), 2:zap(force to full cvg), 3:save(don't overwrite memory cvg).
H:color_on_cvg	0	7	only update color on coverage overflow(transparent surface)
G:image_read_en	0	6	enable color/cvg read/modify/write memory access
F:z_update_en	0	5	enable writing of Z if color write enabled
E:z_compare_en	0	4	conditional color write enable on depth comparison
D:antialias_en	0	3	if not force blend, allow blend enable-use cvg bits
C:z_source_sel	0	2	choose between Primitive Z and pixel Z
B:dither_alpha_en	0	1	use random noise in alpha compare, otherwise use blend alpha in alpha compare
A:alpha_compare_en	0	0	conditional color write on alpha compare

Set_Other_Modes Usage Notes:

YUV Copy mode is not supported.
Fill mode in "Cycle Type" means replicate the Fill Color(see Set_Fill_Color).
Initialize the Z-buffer by setting the color image to point to the Z-buffer(Set_Color_Image) and Filling with initial depth value.
Multi-tile mixed color index thru TLUT and other not supported, because TLUT effects all modes.



Set Env Color

Field	Word	Bits	Description
command	0	61-56	Command identifier
Red	0	31-24	Red Component
Green	0	23-16	Green Component
Blue	0	15-8	Blue Component
Alpha	0	7-0	Alpha Component

Set Prim Color

Set Env Color

Field	Word	Bits	Description
command	0	61-56	Command identifier
Prim Min Level	0	44-40	Minimum clamp for LOD fraction when in detail or sharpen texture modes, fixed point 0.5.
Prim Level Frac	0	39-32	Level of Detail fraction for primitive, used primarily in multi-tile operations for rectangle primitives, 0.8.
Green	0	23-16	Green Component
Blue	0	15-8	Blue Component
Alpha	0	7-0	Alpha Component

Set Blend Color

63	0	56	55	1	48	47	2	40	39	3	32	31	4	24	23	5	16	15	6	8	7	7	0	0
	command=0x39											Red			Green			Blue			Alpha			

Set . Blend Color

Field	Word	Bits	Description
command	0	61-56	Command identifier
Red	0	31-24	Red Component
Green	0	23-16	Green Component
Blue	0	15-8	Blue Component
Alpha	0	7-0	Alpha Component

Set Fog Color

Set Fog Color

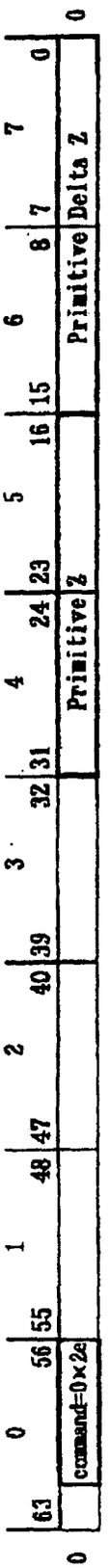
Field	Word	Bits	Description
command	0	61-56	Command identifier
Red	0	31-24	Red Component
Green	0	23-16	Green Component
Blue	0	15-8	Blue Component
Alfha	0	7-0	Alfha Component

Set Fill Color

Set Fill Color

Field	Word	Bits	Description
command	0	61-56	Command identifier
Packed Color	0	31-0	Packed Color, For example, if the Color image was set be 16b RGBA, then the fill color would be two horizontally adjacent 16b RGBA pixels.

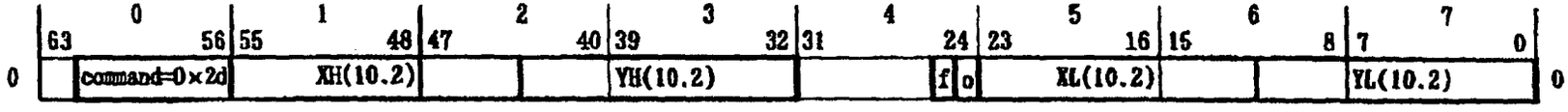
Set Prim Depth



Set Prim Depth

Field	Word	Bits	Description
command	0	61-56	Command identifier
Primitive 2	0	31-16	Primitive 2
Primitive Delta Z	0	15-0	Primitive Delta Z

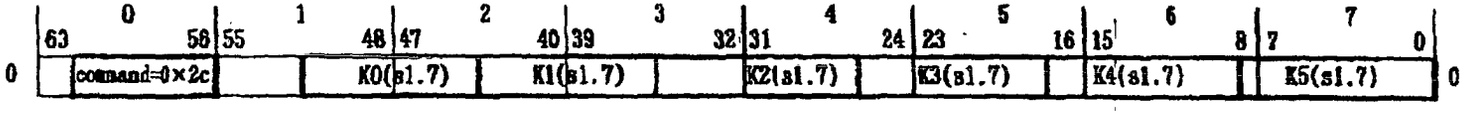
Set Scissor



Set Scissor

Field	Word	Bits	Description
command	0	61-56	Command identifier
XH	0	55-44	X coordinate of top left corner of scissor box in screen space.
YH	0	43-32	Y coordinate of top left corner of scissor box in screen space.
f	0	25	scissor field, enables scissoring of odd or even lines for interlaced displays
o	0	24	odd line:0=keep even line, 1=keep odd line, indicates whether all odd lines or all even lines should be skipped(for interlaced displays).
XL	0	23-12	X coordinate of bottom right corner of scissor box in screen space.
YL	0	11-0	Y coordinate of bottom right corner of scissor box in screen space.

Set Convert



Set Convert

Field	Word	Bits	Description
command	0	61-56	Command identifier, this command updates the coefficients for converting YUV pixels to RGB. Conceptually the equations are: R=C0*(Y-16)+C1*V G=C0*(Y-16)-C2*U-C3*V B=C0*(Y-16)+C4*U
K0	0	53-45	K0 term of YUV-RGB conversion matrix
K1	0	44-36	K1 term of YUV-RGB conversion matrix
K2	0	35-27	K2 term of YUV-RGB conversion matrix
K3	0	26-18	K3 term of YUV-RGB conversion matrix
K4	0	17-9	K4 term of YUV-RGB conversion matrix
K5	0	8-0	K5 term of YUV-RGB conversion matrix

Set_Convert Usage Notes:

In the hardware, the color conversion is done in two stages.
In the texture filter(TF), the following equation is performed:

R' =Y+K0*V
G' =Y+K1*U+K2*V
B' =Y+K3*U

In the color combiner, the following equations are performed:

R=(R' -K4)*K5+R'
G=(G' -K4)*K5+G'
B=(B' -K4)*K5+B'

Where(CI terms as defined in the table above):

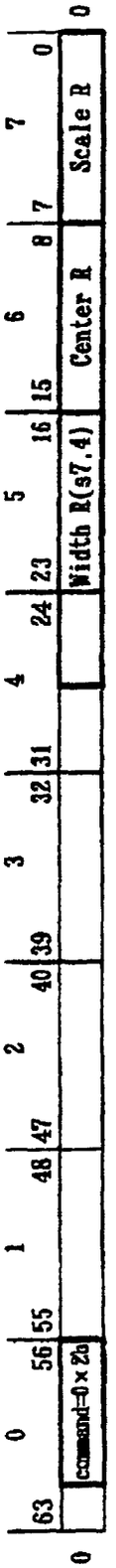
K0=C1/C0
K1=C2/C0
K2=C3/C0
K3=C4/C0
K4=16+16/(C0-1.0)
K5=C0-1.0

Typical Values for YUV to RGB conversion:

K0=175
K1=-43
K2=-89
K3=222
K4=114
K5=42

EP 0 778 536 B1

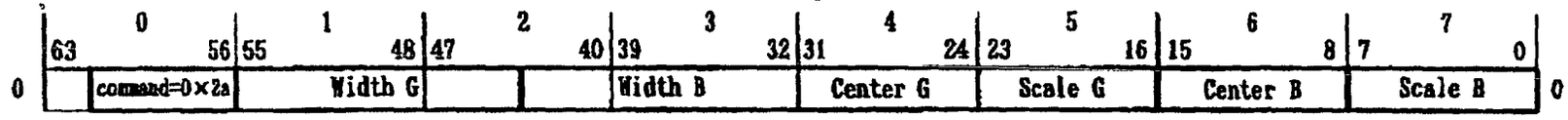
Set Key R



Set_ Key_ R

Field	Word	Bits	Description
command	0	61-56	Command identifier. This command set the coefficients used for Red keying. The equation used for keying is: KeyR=clamp(0.0,-abs(R-Center)*Scale)+Width,1.0). The Key Alpha is the minimum of the KeyR, KeyG, KeyB.
Width R	0	27-16	(Size of half the key window including the soft edge)*scale. If width>1.0, then keying is disabled for that channel.
Center R	0	15-8	Defines color or intensity at which key is active,0-255
Scale R	0	7-0	(1.0/(size of soft edge). For hard edge keying, set scale to 255.

Set Key GB



Set_ Key_ GB

Field	Word	Bits	Description
command	0	61-56	Command identifier, This command set the coefficients used for Green/Blue keying. Conceptually, the equation used for keying is: KeyG/B=clamp(0.0,-abs(G/B-Center)*Scale)+Width,1.0). The Key Alpha is the minimum of the KeyR, KeyG, KeyB.
Width G	0	55-44	(Size of half the key window including the soft edge)*scale. If width>1.0, then keying is disabled for that channel.
Width B	0	43-32	(Size of half the key window including the soft edge)*scale. If width>1.0, then keying is disabled for that channel.
Center G	0	31-24	Defines color or intensity at which key is active,0-255
Scale G	0	23-16	(1.0/(size of soft edge). For hard edge keying, set scale to 255.
Center B	0	15-8	Defines color or intensity at which key is active,0-255
Scale B	0	7-0	(1.0/(size of soft edge). For hard edge keying, set scale to 255.

Set_ Key_ XX Usage Notes:

In the hardware, the keying equation is performed in two stages.
In the Color Combiner(CC), the equation performed is:

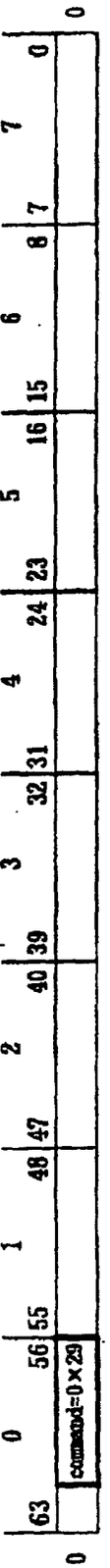
Key' =(pixel-Center)*Scale+0

In the Alpha Fixup unit(AF), the equation performed is:

Key=clamp(0,-abs(Key')+Width,1.0)
KeyAlpha=MIN(KeyR,KeyG,KeyB)

In two-cycle mode, the keying operation must be specified
in the second cycle(that is, the key alpha is not available as a combine operand).

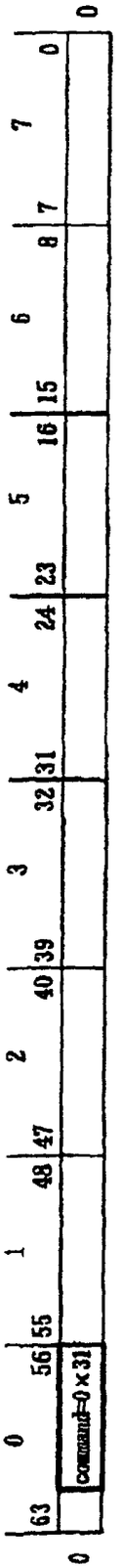
EP 0 778 536 B1



Sync Full

Field	Word	Bits	Description
command	0	61-56	Command identifier, This command stalls the RDP until the last dram buffer is read or written from any preceeding primitive. It is typically only needed if the memory data is to be reused, like switching display buffers, or writing a color_image to be used as a texture_image, or for consistent r/w access to an RDP w/r image from the cpu.

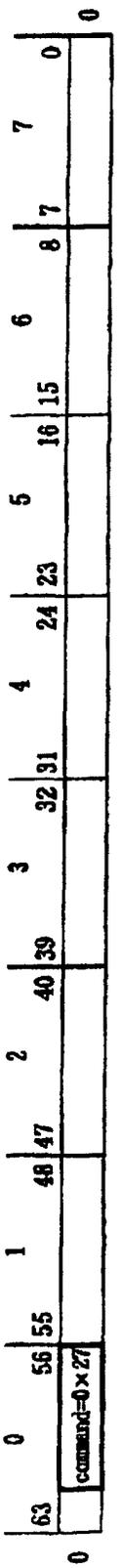
Sync Load



Sync Full

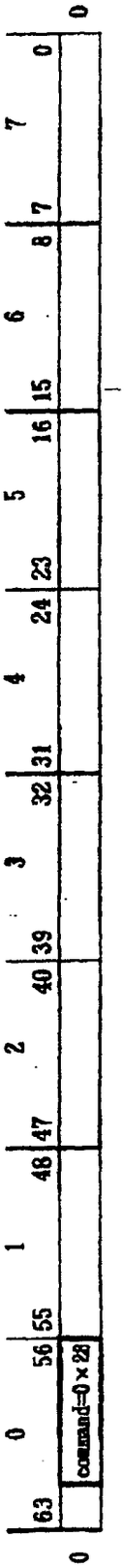
Field	Word	Bits	Description
command	0	61-56	Command identifier. This command stalls the execution of load commands(loadTLUT, load tile, load block) until preceeding primitives has completely finished. Usually preceed all load commands by sync load.

Sync Pipe



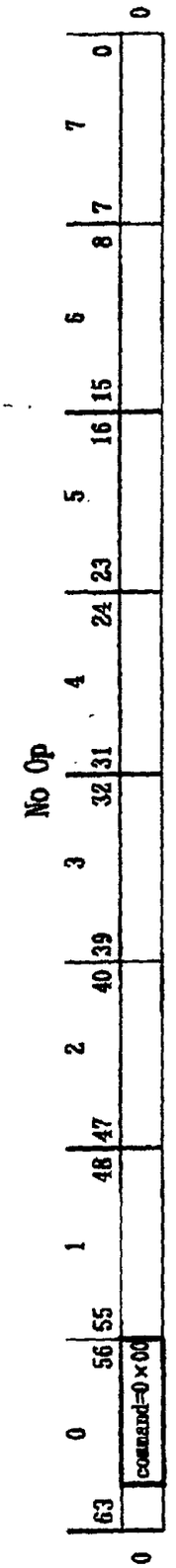
Sync Pipe

Field	Word	Bits	Description
command	0	61-56	Command identifier, Geuelal attributes(other than pria_color/depth) which are being read by up to two preceeding primitives should be peceeded by sync_pipe, which stalls until the most recent primitive is past the last usage of any attribute. Only one sync_pipe is needed before any number of attribute commands. Software can optimize sync_pipe usage if it knows what is being read, for example, a set_texture_image can follow tri's or rects in preparation for a load_tile without a sync_pipe, because tris or rects don't use the texture_image attribute. In general, the RDP is optimized for a number of primitives rendered with the same attribute setting. If attributes change per primitive, performance will degrade slightly.



Sync Tile

Field	Word	Bits	Description
command	0	61-56	Command identifier, Allors synchronization between commands that write to the same tile descriptor that an immediately previous command is reading.



No Op

Field	Word	Bits	Description
command	0	61-56	Command identifier. This command has no effect on rdp command execution but is useful for padding command buffers.

Claims

1. An interactive video game system (50) comprising:

at least one interactive user input device (56);
a main processor (100) coupled to the input device (56), the main processor (100) having an address space, the main processor (100) interactively selecting a point of view in response to inputs from the user input device (56);
a coprocessor (200) coupled to the main processor (100), the coprocessor (200) providing a predetermined graphics feature set for interactively generating image data in response to the selected point of view by projecting polygons representing a three dimensional world onto a two dimensional viewing plane, the coprocessor (200) including:

a signal processor (400) that is shared between at least graphics functions and audio processing functions, the signal processor (400) including a scalar unit (410) and a vector unit (420), the vector unit (420) capable of performing plural calculations in parallel, the signal processor (400) including a microcode store (402) that stores microcode, the signal processor (400) executing the microcode in the microcode store (402) to perform the graphics and audio processing functions;
a display processor (500) comprising display pipeline hardware that alternatively provides a one-pixel-per-cycle mode and a two-pixel-per-cycle mode to minimize hardware while providing a rich feature set including level-of-detail processing, the display pipeline hardware including a texture memory having first and second parts, the texture memory first part being capable of storing texture maps that are colour indexed and texture maps that are not colour indexed, the texture memory second part being capable of storing texture maps and/or colour lookup tables for the colour indexed texture maps,
a video interface (210),
an audio interface (208),
a serial interface (204), and
a parallel peripheral interface (206),
each of the signal processor (400), the display processor (500), the video interface (210), the audio interface (208), the serial interface (204) and the parallel peripheral interface (206) includes circuitry for accessing a main memory (300); the main memory (300) being coupled to the coprocessor (200) via a 9 bit wide bus, the main memory (300) providing a common address space for the coprocessor (200) and the main processor (100), the main memory (300) storing at least the following data structures:

instructions for execution by the main processor (100);
a colour frame buffer (118a);
a depth buffer (118b);
graphics microcode (156);
audio processing microcode (156);
at least one display list (110a);
at least one texture map (116); and
at least one audio output buffer (114);
a video signal generating circuit coupled to the coprocessor video interface (210), the video signal generating circuit generating a video signal for display on a colour television set;
a removable storage device (54) including a housing, a security chip (152), a read only memory (76) and at least one further memory device, the coprocessor (200) including an arrangement that maps the read only memory and the further memory device into the main processor (100) address space, the read only memory initially storing the graphics and audio processing microcode; and
a connector (154) that connects the coprocessor (200) to the removable storage device (54); and
a serial peripheral interface circuit (138) coupled to the coprocessor serial interface (204), the serial peripheral interface circuit (138) including a processor that performs serial interface functions and security functions and further includes a boot ROM that provides main processor (100) initial program load instructions, the serial interface circuit processor being coupled to the removable storage device security chip (152) through the connector (154).

2. An interactive video game system (50) as claimed in claim 1, wherein the main memory (300) providing a common address space is a random access memory.

3. An interactive video game system (50) as claimed in claim 1 or 2, wherein the main processor (100) is also coupled to address the main memory (300).

4. An interactive video game system (50) as claimed in claim 1, 2 or 3, wherein the main processor (100) stores

instructions in and executes instructions from the main memory (300) in real time response to inputs received from the at least one user input device (56), the main processor (100) storing at least display list graphics commands (110a) and play list audio commands (110b) in the main memory (300).

5. An interactive video game system (50) as claimed in any one of claims 1 to 4, wherein the signal processor (400) is coupled to address the main memory (300).

6. An interactive video game system as claimed in claim 4 or 5, wherein the signal processor (400) fetches and executes microcode stored in the main memory (300), the signal processor (400) reading the display list graphics commands (110a) and play list audio commands (110b) from the main memory (300), the signal processor (400) generating audio sample data in response to the play list audio commands (110b) and generating graphics display commands in response to the display list, the signal processor (400) storing the sample data in the audio output buffer (114) allocated within the main memory.

7. An interactive video game system (50) as claimed in any one of claims 1 to 6, wherein the display processor (500) is coupled to address the main memory (300).

8. An interactive video game system (50) as claimed in any one claims 3 to 7, wherein the display processor (500) generates image data based at least in part on at least one texture map and other graphics data stored in the main memory (300), the display processor (500) producing the image data in response to the graphics display commands (110a), the display processor (500) storing the image data in the colour image frame buffer (118a) within the main memory (300).

9. An interactive video game system (50) as claimed in any one of claims 1 to 8, wherein the video interface (210) is coupled to address the main memory (300).

10. An interactive video game system (50) as claimed in any one of claims 1 to 9, wherein the video interface (210) reads the colour image frame buffer (118a) in synchronism with a display raster scan (504).

11. An interactive video game system (50) as claimed in any one of claims 1 to 10, wherein the audio interface (208) is coupled to address the main memory (300).

12. An interactive video game system (50) as claimed in any one of claims 1 to 11, wherein the audio interface (208) reads the audio output buffer (114) in synchronism with real time sound generation.

13. An interactive video game system (50) as claimed in any one of claims 1 to 12, wherein the display processor (500) includes a rasterizer circuit (504), a texture unit (506) and associated texture memory (502), a colour combiner (508), a blender circuit (510) for, in use, blending the colour combiner output with contents of the frame buffer (118a), and a memory interface circuit (512) for, in use, being coupled to the random access memory.

14. An interactive video game system (50) as claimed in any one of claims 1 to 13, wherein the co-processor (200) further comprises a loading circuit for loading graphics processing instructions and audio processing instructions into the microcode store (402), wherein at least some of the graphics processing instructions and the audio processing instructions loaded in the microcode store (402) by said loading circuit originate from a portable replaceable memory device included in the system.

15. An interactive video game system (50) as claimed in any one of claims 1 to 14, wherein the graphics display commands generated by the signal processor (400) are provided to the display processor (500) via a private bus.

16. An interactive video game system (50) as claimed in any one of claims 1 to 15, wherein the vector processing unit (420) comprises a plurality of calculation units operating in parallel.

17. An interactive video game system (50) as claimed in any one of claims 1 to 16, wherein the vector processing unit (420) comprises a plurality of parallel fixed point product-sum calculation circuits.

18. An interactive video game system as claimed in any one of claims 1 to 17, wherein the vector processing unit (420) provides fixed point calculations based on floating point type instructions in said microcode store (402).

19. An interactive video game system (50) as claimed in any one of claims 1 to 18, wherein the vector processing unit (420) comprises a plurality of calculation units operating in parallel based on the same instruction from said microcode store (402).

20. A method of operating a graphics display system of the type including a main processor (100), a coprocessor (200) coupled to the main processor (100), a main random access memory (300) coupled to the coprocessor (200) and addressable by both the main processor (100) and the coprocessor (200) through a 9 bit wide bus, and a video signal generating arrangement that produces a video signal for display, the method including the following steps:

(a) storing main processor code in the main memory (300);
(b) executing, with the main processor (100), the main processor . code stored by the storing step, said executing step including storing coprocessor code, a task list, at least one texture map and a colour lockup table into the main memory (300);
(c) fetching the task list from main memory (300);
(d) processing the task list with the coprocessor (200) in accordance at least in part with the coprocessor code stored by step (b), the processing step including performing the following steps:

(1) loading the texture map and the colour lookup table from the main memory (300) into an on-chip texture memory;
(2) performing at least one 3D geometric transformation on a set of vertices using a scalar unit (410) and a vector unit (420) including performing multiple calculations in parallel with the vector unit (420);
(3) generating a triangle command based on the 3D geometric transformation;
(4) generating a pixel value in response to the triangle command;
(5) accessing the texture memory twice to provide colour indexed texels based on the triangle command;
(6) combining the texels with the generated pixel value to generate a combined pixel value;
(7) accessing pixel values in a frame buffer (118) stored in the main memory;
(8) blending the combined pixel value with at least one pixel value stored in the frame buffer (118);
(9) conditionally writing the combined pixel value into the frame buffer (118) based on a comparison using a depth buffer (118b) stored in the main memory (300);
(10) using said scalar (410) and vector (420) units to generate output audio samples including performing multiple calculations in parallel with the vector unit (420); and
(11) storing the output audio samples into the main memory (300);

(e) reading the frame buffer (118) in real time synchronism with colour television set line scanning and converting the frame buffer contents to a composite video signal; and
(f) reading the stored output audio samples in real time and converting the stored audio samples into stereo sound.

Patentansprüche

1. Interaktives Videospielsystem (50), umfassend
mindestens eine interaktive Benutzereingabevorrichtung (56);
einen Hauptprozessor (100), der mit der Eingabevorrichtung (56) verbunden ist, wobei der Hauptprozessor (100) einen Adressraum aufweist, wobei der Hauptprozessor (100) ansprechend auf Eingaben von der Benutzereingabevorrichtung (56) interaktiv einen Blickpunkt auswählt;
einen Koprozessor (200), der mit dem Hauptprozessor (100) verbunden ist, wobei der Koprozessor (200) eine vorbestimmte Grafikmerkmalsgruppe bereitstellt, um ansprechend auf den ausgewählten Blickpunkt interaktiv Bilddaten zu erzeugen, durch Projizieren von Polygonen, die eine dreidimensionale Welt darstellen, auf eine zweidimensionale Betrachtungsebene, wobei der Koprozessor (200) einschließt:

einen Signalprozessor (400), der mindestens zwischen Grafikfunktionen und Tonverarbeitungsfunktionen aufgeteilt wird, wobei der Signalprozessor (400) eine Skalareinheit (410) und eine Vektoreinheit (420) einschließt, wobei die Vektoreinheit (420) imstande ist, mehrfache Berechnungen parallel auszuführen, wobei der Signalprozessor (400) einen Mikrocodespeicher (402) einschließt, der Mikrocode speichert, wobei der Signalprozessor (400) den Mikrocode im Mikrocodespeicher (402) ablaufen lässt, um die Grafik- und Tonverarbeitungsfunktionen auszuführen;
einen Anzeigeprozessor (500), der Anzeigefließbandhardware umfasst, die zum Minimieren von Hardware

alternativ einen Ein-Pixel-pro-Zyklus-Modus und einen Zwei-Pixel-pro-Zyklus-Modus bereitstellt, während für eine reichhaltige Merkmalsgruppe einschließlich einer Detailmaßverarbeitung gesorgt ist, wobei die Anzeigefließbandhardware einen Texturspeicher einschließt, der einen ersten und zweiten Teil aufweist, wobei der erste Teil des Texturspeichers imstande ist, Texturabbilder, die mit Farbenindices versehen sind, und Texturabbilder, die nicht mit Farbenindices versehen sind, zu speichern, wobei der zweite Teil des Texturspeichers imstande ist, Texturabbilder und/oder Farbnachschlagtabellen für die mit Farbenindices versehenen Texturabbilder zu speichern,
eine Videoschnittstelle (210),
eine Tonschnittstelle (208),
eine serielle Schnittstelle (204), und
eine parallele Peripherieschnittstelle (206),

wobei der Signalprozessor (400), der Anzeigeprozessor (500), die Videoschnittstelle (210), die Tonschnittstelle (208), die serielle Schnittstelle (204) und die parallele Peripherieschnittstelle (206) jeweils eine Schaltungsanordnung zum Zugreifen auf einen Hauptspeicher (300) einschließen; wobei der Hauptspeicher (300) über einen 9 Bit breiten Bus mit dem Koprozessor (200) verbunden ist, wobei der Hauptspeicher (300) einen gemeinsamen Adressraum für den Koprozessor (200) und den Hauptprozessor (100) bereitstellt, wobei der Hauptspeicher (300) mindestens die folgenden Datenstrukturen speichert:

Instruktionen zur Ausführung durch den Hauptprozessor (100);
einen Farbbildpufferspeicher (118a);
einen Tiefenpufferspeicher (118b);
Grafikmikrocode (156);
Tonverarbeitungsmikrocode (156);
mindestens eine Anzeigeliste (110a);
mindestens ein Texturabbild (116); und
mindestens einen Tonausgabepufferspeicher (114);

eine Videosignalerzeugungsschaltung, die mit der Videoschnittstelle (210) des Koprozessors verbunden ist, wobei die Videosignalerzeugungsschaltung ein Videosignal zur Anzeige auf einem Farbfernsehgerät erzeugt;
eine abnehmbare Speichervorrichtung (54), die ein Gehäuse, einen Sicherheits-Chip (152), einen Nur-Lese-Speicher (76) und mindestens eine weitere Speichervorrichtung einschließt, wobei der Koprozessor (200) eine Anordnung einschließt, die den Nur-Lese-Speicher und die weitere Speichervorrichtung in den Adressraum des Hauptprozessors (100) abbildet, wobei der Nur-Lese-Speicher zu Beginn den Grafik- und Tonverarbeitungsmikrocode speichert; und
einen Steckverbinder (154), der den Koprozessor (200) mit der abnehmbaren Speichervorrichtung (54) verbindet; sowie
eine serielle Peripherieschnittstellenschaltung (138), die mit der seriellen Schnittstelle (204) des Koprozessors verbunden ist, wobei die serielle Peripherieschnittstellenschaltung (138) einen Prozessor einschließt, der serielle Schnittstellenfunktionen und Sicherheitsfunktionen ausführt und weiter einen Boot-ROM einschließt, der dem Hauptprozessor (100) Urladeprogrammanweisungen liefert, wobei der Prozessor der seriellen Schnittstellenschaltung durch den Steckverbinder (154) mit dem Sicherheits-Chip (152) der abnehmbaren Speichervorrichtung verbunden ist.

2. Interaktives Videospielsystem (50) nach Anspruch 1, bei dem der einen gemeinsamen Adressraum liefernde Hauptspeicher (300) ein Direktzugriffsspeicher ist.

3. Interaktives Videospielsystem (50) nach Anspruch 1 oder 2, bei dem der Hauptprozessor (100) auch angeschlossen ist, um den Hauptspeicher (300) zu adressieren.

4. Interaktives Videospielsystem (50) nach Anspruch 1, 2 oder 3, bei dem der Hauptprozessor (100) im Echtzeit-Ansprechen auf von der mindestens einen Benutzereingabevorrichtung (56) empfangene Eingaben Anweisungen im Hauptspeicher (300) speichert und Anweisungen aus dem Hauptspeicher (300) ausführt, wobei der Hauptprozessor (100) mindestens Anzeigelisten-Grafikbefehle (110h) und Abspiellisten-Tonbefehle (110) im Hauptspeicher (300) speichert.

5. Interaktives Videospielsystem (50) nach einem der Ansprüche 1 bis 4, bei dem der Signalprozessor (400) angeschlossen ist, um den Hauptspeicher (300) zu adressieren.

6. Interaktives Videospielsystem nach Anspruch 4 oder 5, bei dem der Signalprozessor (400) im Hauptspeicher (300) gespeicherten Mikrocode holt und ausführt, wobei der Signalprozessor (400) die Anzeigelisten-Grafikbefehle (110a) und die Abspiellisten-Tonbefehle (110b) aus dem Hauptspeicher (300) liest, wobei der Signalprozessor (400) ansprechend auf die Abspiellisten-Tonbefehle (110b) Tonmusterdaten erzeugt und ansprechend auf die Anzeigeliste Grafikanzeigebefehle erzeugt, wobei der Signalprozessor (400) die Musterdaten in dem im Hauptspeicher zugeteilten Tonausgabepufferspeicher (114) speichert.

7. Interaktives Videospielsystem (50) nach einem der Ansprüche 1 bis 6, bei dem der Anzeigeprozessor (500) angeschlossen ist, um den Hauptspeicher (300) zu adressieren.

8. Interaktives Videospielsystem (50) nach einem der Ansprüche 3 bis 7, bei dem der Anzeigeprozessor (500) basierend mindestens teilweise auf mindestens einem Texturabbild und anderen im Hauptspeicher (300) gespeicherten Grafikdaten Bilddaten erzeugt, wobei der Anzeigeprozessor (500) die Bilddaten ansprechend auf die Grafikanzeigebefehle (110a) erzeugt, wobei der Anzeigeprozessor (500) die Bilddaten im Farbbildbildpufferspeicher (118a) im Hauptspeicher (300) speichert.

9. Interaktives Videospielsystem (50) nach einem der Ansprüche 1 bis 8, bei dem die Videoschnittstelle (210) angeschlossen ist, um den Hauptspeicher (300) zu adressieren.

10. Interaktives Videospielsystem (50) nach einem der Ansprüche 1 bis 9, bei dem die Videoschnittstelle (210) den Farbbildbildpufferspeicher (118a) in Synchronisation mit einer Anzeigerasterung (504) liest.

11. Interaktives Videospielsystem (50) nach einem der Ansprüche 1 bis 10, bei dem die Tonschnittstelle (208) angeschlossen ist, um den Hauptspeicher (300) zu adressieren.

12. Interaktives Videospielsystem (50) nach einem der Ansprüche 1 bis 11, bei dem die Tonschnittstelle (208) den Tonausgabepufferspeicher (114) in Synchronisation mit Echtzeit-Klangerzeugung liest.

13. Interaktives Videospielsystem (50) nach einem der Ansprüche 1 bis 12, bei dem der Anzeigeprozessor (500) eine Rasterungsschaltung (504), eine Textureinheit (506) und einen zugehörigen Texturspeicher (502), einen Farbkombinierer (508), eine Mischerschaltung (510), um im Gebrauch die Ausgangssignale des Farbkombinierers mit Inhalt des Bildpufferspeichers (118a) zu mischen, und eine Speicherschnittstellenschaltung (512) einschließt, um im Gebrauch mit dem Direktzugriffsspeicher verbunden zu sein.

14. Interaktives Videospielsystem (50) nach einem der Ansprüche 1 bis 13, bei dem der Koprozessor (200) weiter eine Ladeschaltung zum Laden von Grafikverarbeitungsanweisungen und Tonverarbeitungsanweisungen in den Mikrocodespeicher (402) umfasst, wobei mindestens einige von den Grafikverarbeitungsanweisungen und den Tonverarbeitungsanweisungen, die von der Ladeschaltung in den Mikrocodespeicher (402) geladen werden, aus einer im System enthaltenen tragbaren austauschbaren Speichervorrichtung stammen.

15. Interaktives Videospielsystem (50) nach einem der Ansprüche 1 bis 14, bei dem die vom Signalprozessor (400) erzeugten Grafikanzeigebefehle über einen privaten Bus zum Anzeigeprozessor (500) geliefert werden.

16. Interaktives Videospielsystem nach einem der Ansprüche 1 bis 15, bei dem die Vektorverarbeitungseinheit (420) eine Mehrzahl von parallel arbeitenden Berechnungseinheiten umfasst.

17. Interaktives Videospielsystem (50) nach einem der Ansprüche 1 bis 16, bei dem die Vektorverarbeitungseinheit (420) eine Mehrzahl von parallelen Festkomma-Produkt-Summen-Berechnungsschaltungen umfasst.

18. Interaktives Videospielsystem nach einem der Ansprüche 1 bis 17, bei dem die Vektorverarbeitungseinheit (420) basierend auf Anweisungen vom Fließkomma-Typ im Mikrocodespeicher (402) Festkomma-Berechnungen liefert.

19. Interaktives Videospielsystem (50) nach einem der Ansprüche 1 bis 18, bei dem die Vektorverarbeitungseinheit (420) eine Mehrzahl von Berechnungseinheiten umfasst, die basierend auf derselben Anweisung aus dem Mikrocodespeicher (402) parallel arbeiten.

20. Verfahren zum Betreiben eines Grafikanzeigesystems von der Art enthaltend einen Hauptprozessor (100), einen Koprozessor (200), der mit dem Hauptprozessor (100) verbunden ist, einen Hauptspeicher (300) mit Direktzugriff,

der mit dem Koprozessor (200) verbunden ist und sowohl vom Hauptprozessor (100) und vom Koprozessor (200) durch einen 9 Bit breiten Bus adressierbar ist, und eine Videosignalerzeugungsanordnung, die ein Videosignal zum Anzeigen erzeugt, wobei das Verfahren die folgenden Schritte einschließt:

(a) Speichern von Hauptprozessorcode im Hauptspeicher (300) ;
(b) Ausführen, mit dem Hauptprozessor (100), des durch den Speicherungsschritt gespeicherten Hauptprozessorcodes, wobei der Ausführungsschritt ein Speichern von Koprozessorcode, einer Aufgabenliste, mindestens eines Texturabbilds und einer Farbnachschlagtabelle in den Hauptspeicher (300) einschließt;
(c) Holen der Aufgabenliste aus dem Hauptspeicher (300);
(d) Verarbeiten der Aufgabenliste mit dem Koprozessor (200) und zwar mindestens teilweise in Übereinstimmung mit dem durch den Schritt (b) gespeicherten Koprozessorcode, wobei der Verarbeitungsschritt ein Ausführen der folgenden Schritte einschließt:

(1) Laden des Texturabbilds und der Farbnachschlagtabelle aus dem Hauptspeicher (300) in einen chipintegrierten Texturspeicher;
(2) Ausführen von mindestens einer 3D-Geometrie-Transformation an einer Gruppe von Eckpunkten unter Verwendung einer Skalareinheit (410) und einer Vektoreinheit (420), einschließlich Ausführen von mehreren Berechnungen parallel mit der Vektoreinheit (420);
(3) Erzeugen eines Dreieckbefehls basierend auf der 3D-Geometrie-Transformation;
(4) Erzeugen eines Pixelwerts ansprechend auf den Dreieckbefehl;
(5) zweimaliges Zugreifen auf den Texturspeicher, um basierend auf dem Dreieckbefehl mit Farbindices versehene Texel bereitzustellen;
(6) Kombinieren der Texel mit dem erzeugten Pixelwert, um einen kombinierten Pixelwert zu erzeugen;
(7) Zugreifen auf Pixelwerte in einem im Hauptspeicher gespeicherten Bildpufferspeicher (118);
(8) Mischen des kombinierten Pixelwerts mit mindestens einem im Bildpufferspeicher (118) gespeicherten Pixelwert;
(9) bedingungsweise Schreiben des kombinierten Pixelwerts in den Bildpufferspeicher (118) basierend auf einem Vergleich unter Verwendung eines im Hauptspeicher (300) gespeicherten Tiefenpufferspeichers (118b);
(10) Verwenden der Skalareinheit (410) und der Vektoreinheit (420), um Ausgabetonmuster zu erzeugen, einschließlich Ausführen von mehreren Berechnungen parallel mit der Vektoreinheit (420); und
(11) Speichern der Ausgabetonmuster in den Hauptspeicher (300);

(e) Lesen des Bildpufferspeichers (118) in Echtzeit-Synchronisation mit einer Farbfernsehgerätzeilenabtastung und Umwandeln des Bildpufferspeicherinhalts in ein zusammengesetztes Videosignal; sowie
(f) Lesen der gespeicherten Ausgabetonmuster in Echtzeit und Umwandeln der gespeicherten Tonmuster in Stereoklang.

Revendications

1. Un système (50) de jeu vidéo interactif qui comprend:

au moins un dispositif d'entrée interactif (56) d'utilisateur;
un processeur principal (100) couplé au dispositif d'entrée (56), le processeur principal (100) incluant un espace d'adresses, le processeur principal (100) sélectionnant un point de vue de façon interactive en réponse à des entrées provenant du dispositif d'entrée (56) d'utilisateur;
un coprocesseur (200) couplé au processeur principal (100), le coprocesseur (200) réalisant un jeu prédéterminé de particularités de graphiques pour engendrer des données d'image de façon interactive en réponse au point de vue sélectionné en projetant, sur un plan de visualisation bidimensionnel, des polygones qui représentent un monde tridimensionnel, le coprocesseur (200) incluant:

un processeur (400) de signaux qui est partagé entre au moins des fonctions graphiques et des' fonctions de traitement vidéo, le processeur (400) de signaux incluant une unité scalaire (410) et une unité vectorielle (420), l'unité vectorielle (420) pouvant exécuter plusieurs calculs en parallèle, le processeur (400) de signaux incluant un enregistrement (402) de microcode qui enregistre un microcode, le processeur (400) de signaux exécutant le microcode contenu dans l'enregistrement (402) de microcode pour exécuter les fonctions graphiques et les fonctions de traitement vidéo;

un processeur (500) d'affichage qui comprend des éléments matériels de pipeline d'affichage qui réalisent en alternance un mode à un pixel par cycle et un mode à deux pixels par cycle pour minimiser les éléments matériels tout en réalisant un riche jeu de particularités qui inclut un traitement au niveau de détail, les éléments matériels de pipeline d'affichage incluant une mémoire de texture qui comprend une première et une deuxième fractions, la première fraction de la mémoire de texture pouvant enregistrer des applications de texture qui sont indexées en couleurs et des applications de texture qui ne sont pas indexées en couleurs, la deuxième fraction de mémoire de texture pouvant enregistrer des applications de texture et/ou des tables à consulter, appelées simplement tables dans ce qui suit, de couleurs pour les applications de texture indexées en couleurs,

une interface vidéo (210),
une interface audio (208),
une interface sérielle (204),
une interface parallèle (206) de périphérique,
le processeur (400) de signaux et le processeur (500) d'affichage, l'interface vidéo (210), l'interface audio (208), l'interface sérielle (204) et l'interface parallèle (206) de périphérique incluant chacun un circuit d'accès à une mémoire principale (300); la mémoire principale (300) étant couplée au coprocesseur (200) par l'intermédiaire d'un bus d'une largeur de 9 bits, la mémoire principale (300) réalisant un espace commun d'adresses pour le coprocesseur (200) et le processeur principal (100), la mémoire principale (300) enregistrant au moins les structures suivantes de données:

des instructions à exécuter par le processeur principal (100);
un tampon (118a) de trame de couleurs;
un tampon (118b) de profondeur;
un microcode (156) de graphiques;
un microcode (156) de traitement audio;
au moins une liste d'affichage (110a);
au moins une application (116) de texture; et
au moins un tampon de sortie audio (114);

un circuit générateur de signal vidéo couplé à l'interface vidéo (210) du coprocesseur, le circuit de génération de signal vidéo engendrant un signal vidéo à afficher sur un appareil de télévision en couleurs;
un dispositif d'enregistrement amovible (54) qui inclut un boîtier, une microplaquette (152) de sécurité, une mémoire morte (76) et au moins un autre dispositif de mémoire, le coprocesseur (200) incluant un agencement qui applique la mémoire morte et des dispositifs logiciels de mémoire dans l'espace d'adresses du processeur principal (100), la mémoire morte enregistrant initialement le microcode de graphiques et de traitement audio; et
un connecteur (154) qui connecte le coprocesseur (200) au dispositif d'enregistrement amovible (54); et
un circuit (138) d'interface sérielle de périphérique couplé à l'interface sérielle (204) du coprocesseur, le circuit (138) d'interface sérielle de périphérique incluant un processeur qui exécute des fonctions d'interface sérielle et des fonctions de sécurité et incluant en outre une mémoire morte d'amorçage qui envoie des instructions initiales de chargement de programme au processeur principal (100), le processeur du circuit d'interface sérielle étant couplé à la microplaquette de sécurité (152) du dispositif d'enregistrement amovible par l'intermédiaire du connecteur (154).

2. Un système (50) de jeu vidéo interactif selon la revendication 1, dans lequel la mémoire principale (300) qui réalise un espace commun d'adresses est une mémoire vive.

3. Un système (50) de jeu vidéo interactif selon la revendication 1 ou 2, dans lequel le processeur principal (100) est aussi couplé de manière à adresser la mémoire principale (300).

4. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications précédentes, dans lequel le processeur principal (100) enregistre des instructions dans la mémoire principale (300) et exécute des instructions provenant de la mémoire principale (300) en répondant en temps réel à des entrées reçues du dispositif d'entrée unique au moins (56) d'utilisateur, le processeur principal (100) enregistrant au moins, dans la mémoire principale (300), des commandes (110a) de graphiques de liste d'affichage et des commandes audio (110b) de liste de lecture.

5. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications précédentes, dans lequel le

processeur (400) de signaux est couplé de manière à adresser la mémoire principale (300).

6. Un système (50) de jeu vidéo interactif selon la revendication 4 ou 5, dans lequel le processeur (400) de signaux extrait et exécute un microcode enregistré dans la mémoire principale (300), le processeur (400) de signaux lisant dans la mémoire principale (300) les commandes de graphiques (110a) de liste d'affichage et les commandes audio (110b) de liste de lecture, le processeur (400) de signaux engendrant des données d'échantillons audio en réponse aux commandes audio (110b) de liste de lecture et engendrant des commandes d'affichage de graphiques en réponse à la liste d'affichage, le processeur (400) de signaux enregistrant les données échantillons dans le tampon de sortie audio (114) alloué à l'intérieur de la mémoire principale.

7. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications précédentes, dans lequel le processeur (500) d'affichage est couplé de manière à adresser la mémoire principale (300).

8. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications 3 à 7, dans lequel le processeur (500) d'affichage enregistre des données d'image sur la base au moins en partie d'au moins une application de texture et d'autres données graphiques enregistrées dans la mémoire principale (300), le processeur (500) d'affichage produisant les données d'image en réponse à des commandes (110a) d'affichage de graphique, le processeur (500) d'affichage enregistrant les données d'image dans le tampon (118a) de trame d'image en couleurs à l'intérieur de la mémoire principale (300).

9. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications précédentes, dans lequel l'interface vidéo (210) est couplée de manière à adresser la mémoire principale (300).

10. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications précédentes, dans lequel l'interface vidéo (210) lit le tampon (118a) de trame d'image en couleurs en synchronisme avec un balayage récurrent (504) d'affichage.

11. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications précédentes, dans lequel l'interface audio (208) est couplée de manière à adresser la mémoire principale (300).

12. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications précédentes, dans lequel l'interface audio (208) lit le tampon de sortie audio (114) en synchronisme avec une génération de sons en temps réel.

13. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications précédentes, dans lequel le processeur (500) d'affichage inclut un circuit (504) de balayage récurrent, une unité (506) de texture et une mémoire associée (502) de texture, un combinateur (508) de couleurs, un circuit de mélange (510) pour mélanger, en cours d'utilisation, la sortie du combinateur de couleurs et le contenu du tampon (118a) de trame, et un circuit d'interface (512) de mémoire à coupler, en cours d'utilisation, à la mémoire vive.

14. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications précédentes, dans lequel le coprocesseur (200) comprend en outre un circuit de chargement pour charger des instructions de traitement de graphiques et des instructions de traitement audio dans l'enregistrement (402) de microcode, dans lequel au moins certaines des instructions de traitement de graphiques et des instructions de traitement audio chargées dans l'enregistrement (402) de microcode par ledit circuit de chargement proviennent d'un dispositif de mémoire remplaçable portatif inclus dans le système.

15. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications précédentes, dans lequel les commandes d'affichage de graphiques engendrées par le processeur (400) de signaux sont envoyées au processeur (500) d'affichage par l'intermédiaire d'un bus privé.

16. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications précédentes, dans lequel l'unité (420) de traitement vectoriel comprend une série d'unités de calcul qui fonctionnent en parallèle.

17. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications précédentes, dans lequel l'unité (420) de traitement vectoriel comprend une série de circuits de calcul produit-somme à virgule fixe.

18. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications précédentes, dans lequel l'unité (420) de traitement vectoriel réalise des calculs à virgule fixe sur la base d'instructions du type à virgule flottante

contenues dans ledit enregistrement (402) de microcode.

19. Un système (50) de jeu vidéo interactif selon l'une quelconque des revendications précédentes, dans lequel l'unité (420) de traitement vectoriel comprend une série d'unités de calcul qui fonctionnent en parallèle sur la base de la même instruction provenant dudit enregistrement (402) de microcode.

20. Un procédé de mise en oeuvre d'un système d'affichage graphique du type qui inclut un processeur principal (100), un coprocesseur (200) couplé au processeur principal (100), une mémoire vive principale (300) couplée au coprocesseur (200) et adressable tant par le processeur principal (100) que par le coprocesseur (200) par l'intermédiaire d'un bus d'une largeur de 9 bits, et un agencement générateur de signal vidéo qui produit un signal vidéo à afficher, le procédé incluant les étapes consistant à:

(a) enregistrer un code de processeur principal dans la mémoire principale (300);
(b) exécuter, au moins du processeur principal (100), le code de processeur principal enregistré à l'étape d'enregistrement, ladite étape d'exécution incluant des étapes consistant à enregistrer dans la mémoire principale (300) un code de coprocesseur, une liste de tâches, au moins une application de texture et une table de couleurs;
(c) extraire de la mémoire principale (300) la liste de tâches;
(d) traiter la liste de tâches au moyen du coprocesseur (200) conformément au moins en partie au code de coprocesseur enregistré par l'étape (b), l'étape de traitement incluant l'exécution des étapes consistant à:

(1) charger de la mémoire principale (300) vers une mémoire de texture sur microplaquette l'application de texture et la table de couleurs;
(2) exécuter au moins une transformation géométrique tridimensionnelle sur un jeu de sommets en utilisant une unité scalaire (410) et une unité vectorielle (420), ce qui inclut l'exécution de multiples calculs en parallèle par l'unité vectorielle (420);
(3) engendrer une commande de triangle sur la base de la transformation géométrique tridimensionnelle;
(4) engendrer une valeur de pixel en réponse à la commande de triangle;
(5) accéder deux fois à la mémoire de texture pour réaliser des texels indexés en couleurs sur la base de la commande de triangle;
(6) combiner les texels avec la valeur de pixel engendrée pour engendrer une valeur combinée de pixel;
(7) accéder à des valeurs de pixels contenues dans le tampon (118a) de trame enregistré dans la mémoire principale;
(8) mélanger la valeur combinée de pixel avec au moins une valeur de pixel enregistrée dans le tampon (118a) de trame;
(9) fixer conditionnellement la valeur combinée de pixel dans le tampon (118a) de trame sur la base d'une comparaison qui utilise un tampon (118b) de profondeur enregistré dans la mémoire principale (300);
(10) utiliser lesdites unités scalaire (410) et vectorielle (420) pour engendrer des échantillons audio de sortie, ce qui inclut l'exécution de calculs multiples en parallèle par l'unité vectorielle (420); et
(11) enregistrer les échantillons audio de sortie dans la mémoire principale (300);

(e) lire le tampon (118a) de trame en synchronisme en temps réel avec un balayage de lignes d'un appareil de télévision en couleurs et convertir le contenu du tampon de trame en un signal vidéo composite ; et
(f) lire en temps réel les échantillons audio de sortie enregistrés et convertir en son stéréo les échantillons audio enregistrés.

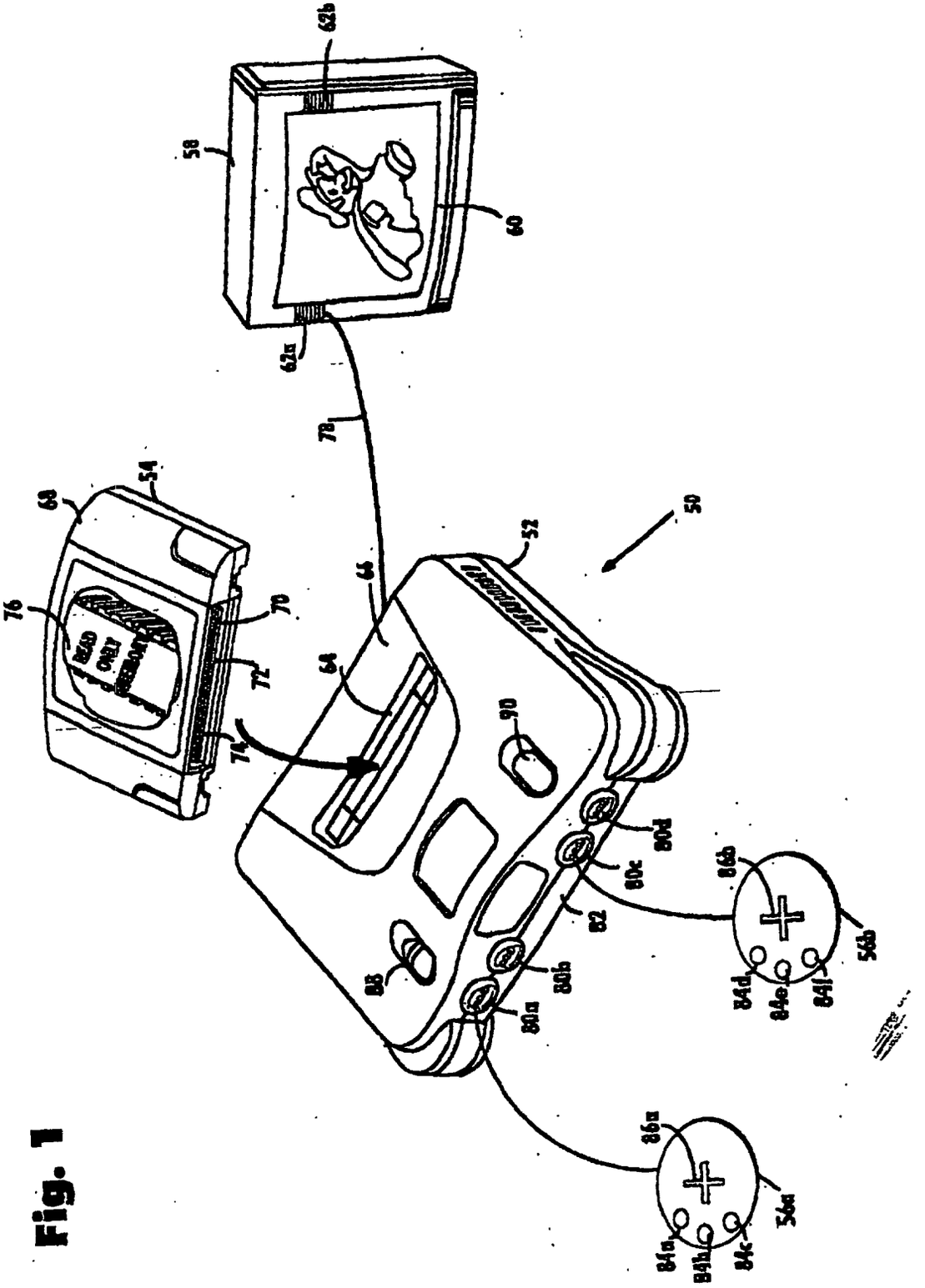

62b
58
60
62a
78
54
68
76
70
72
74
64
66
50
52
90
80d
80c
82
86b
54b
84d
84e
84f
88
80b
80a
86a
54a
84a
84b
84c

Fig. 1

Fig. 1A

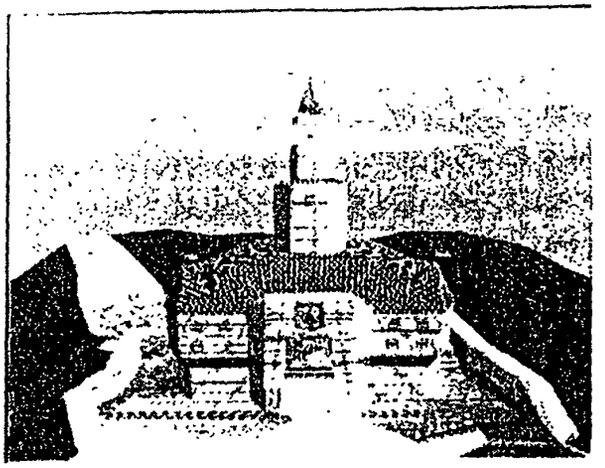


Fig. 1B

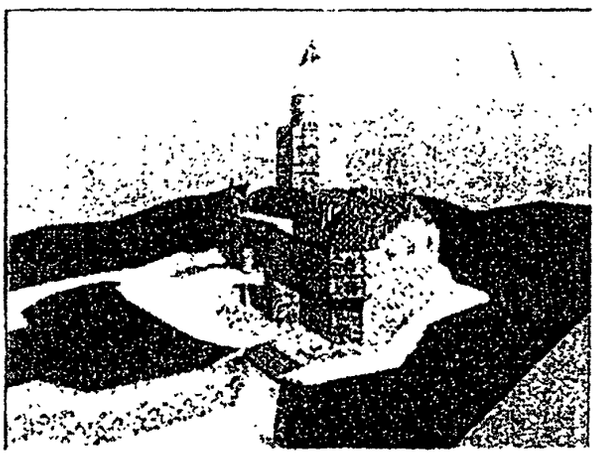


Fig. 1C

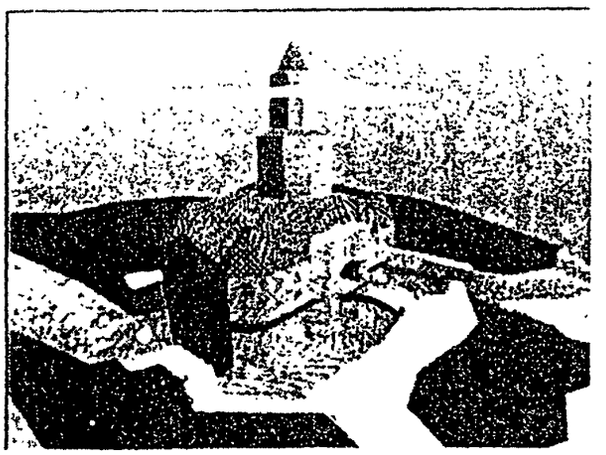


Fig. 1D

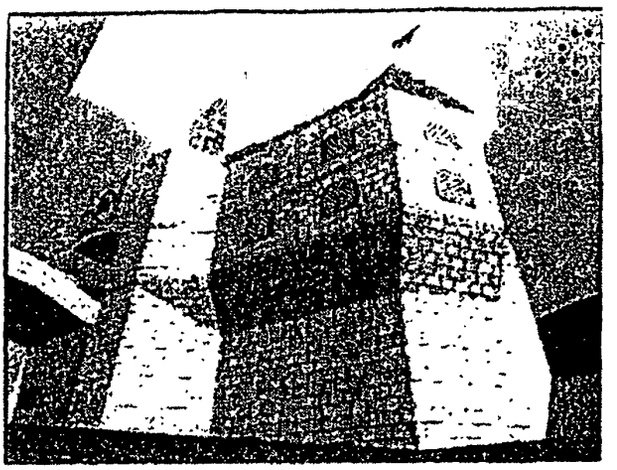


Fig. 1E

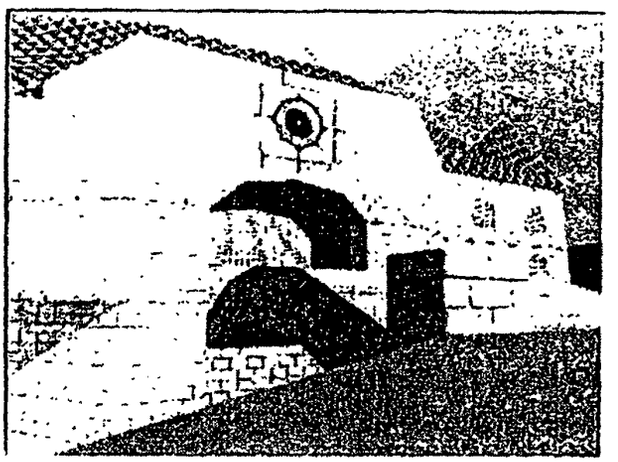
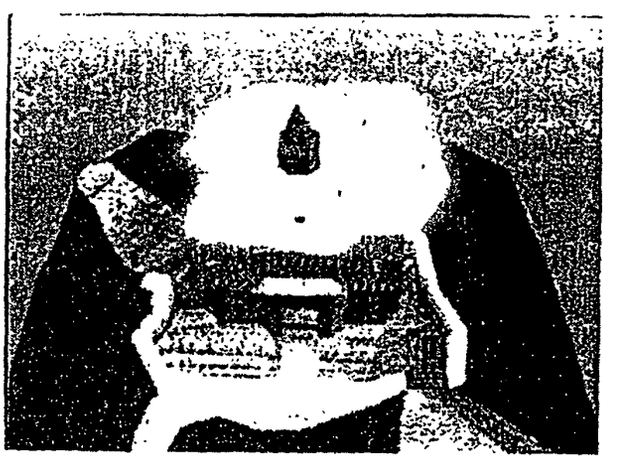


Fig. 1F



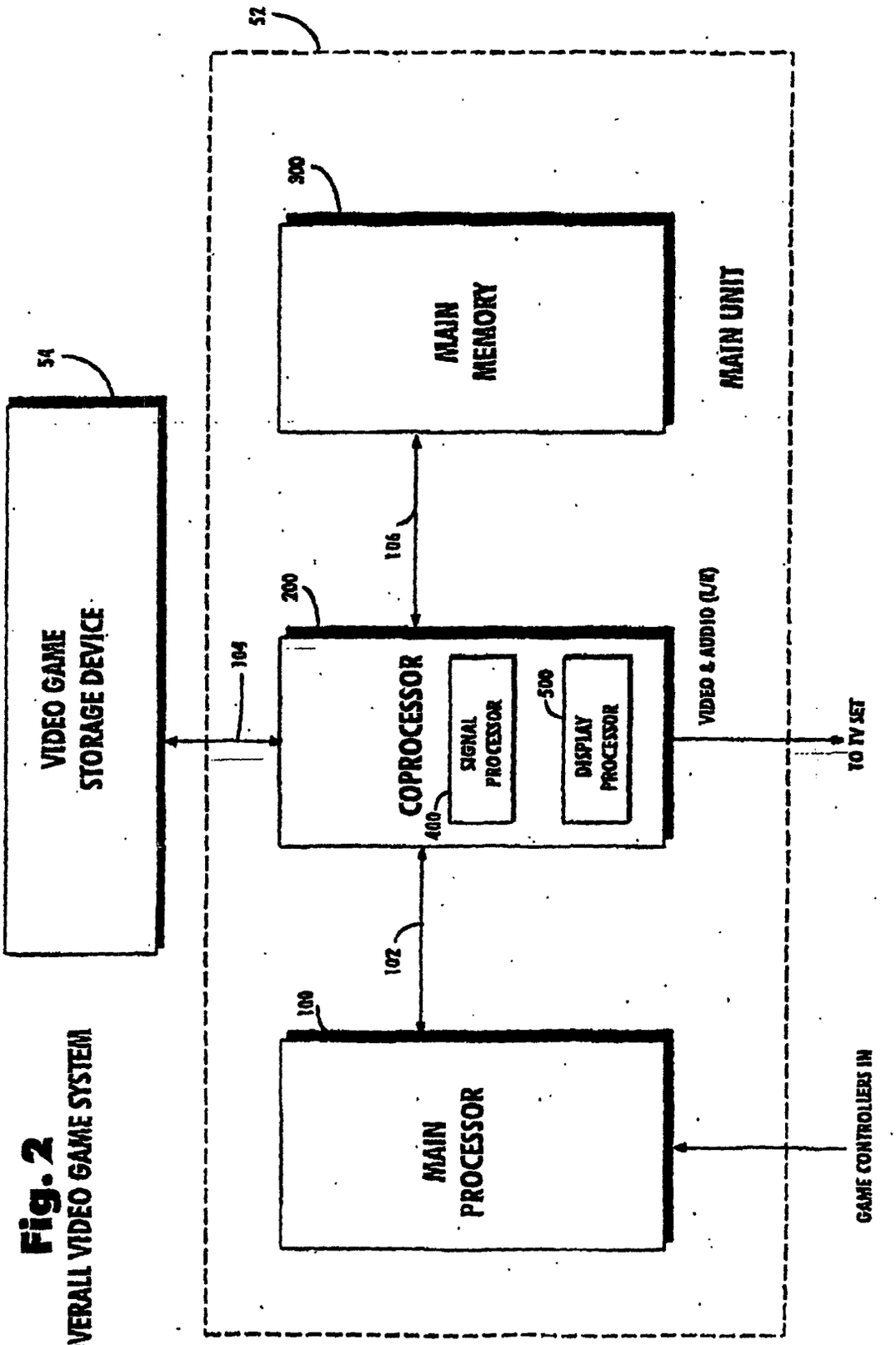


Fig. 2
OVERALL VIDEO GAME SYSTEM

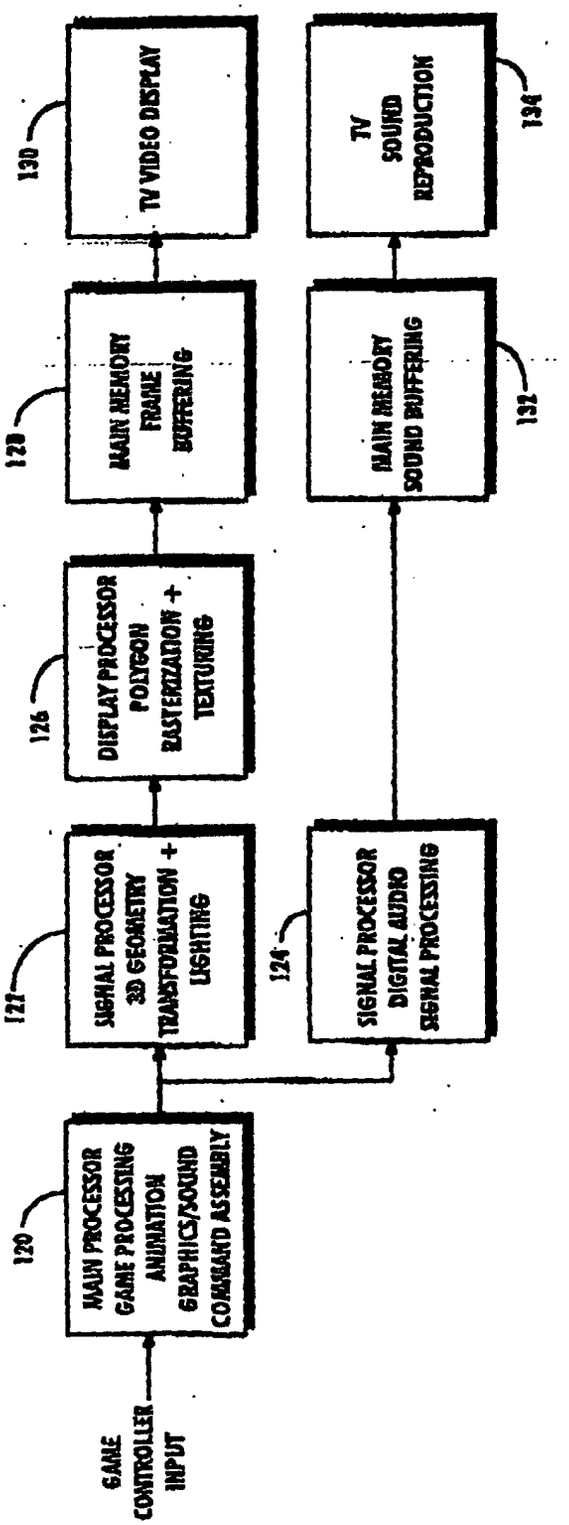


Fig. 3 OVERALL SYSTEM PROCESSES

Fig. 4
OVERALL SYSTEM OPERATION

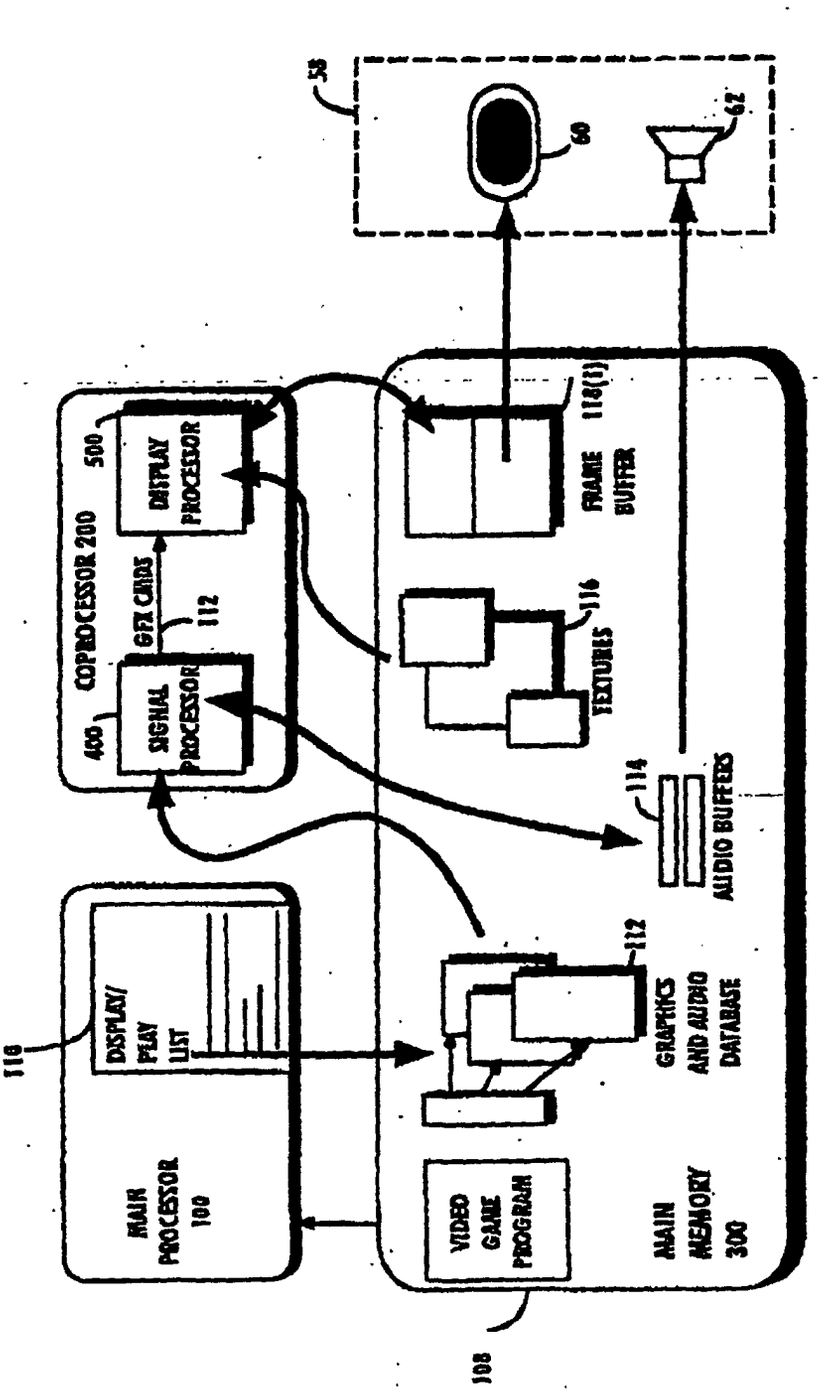
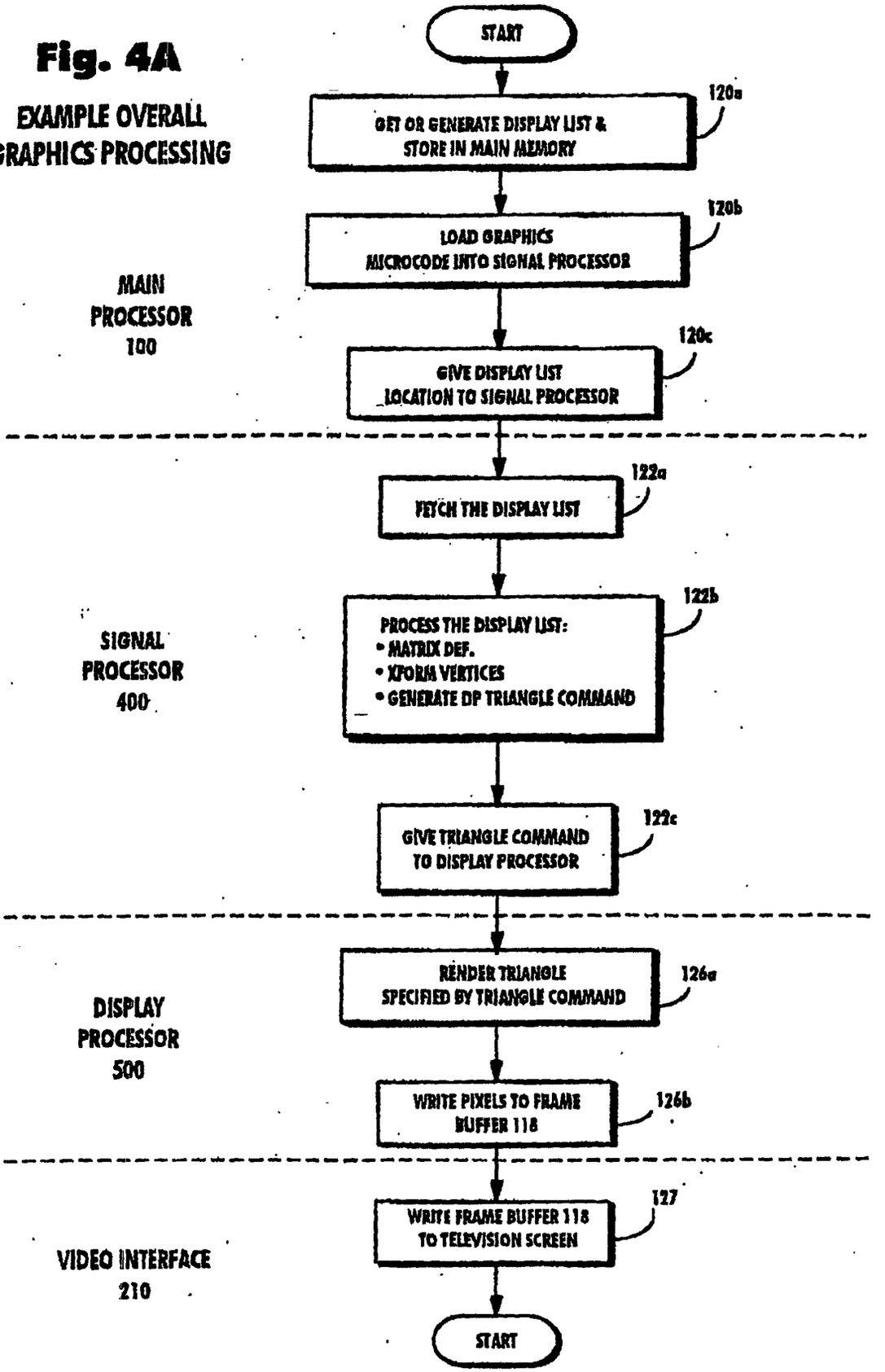


Fig. 4A

EXAMPLE OVERALL GRAPHICS PROCESSING



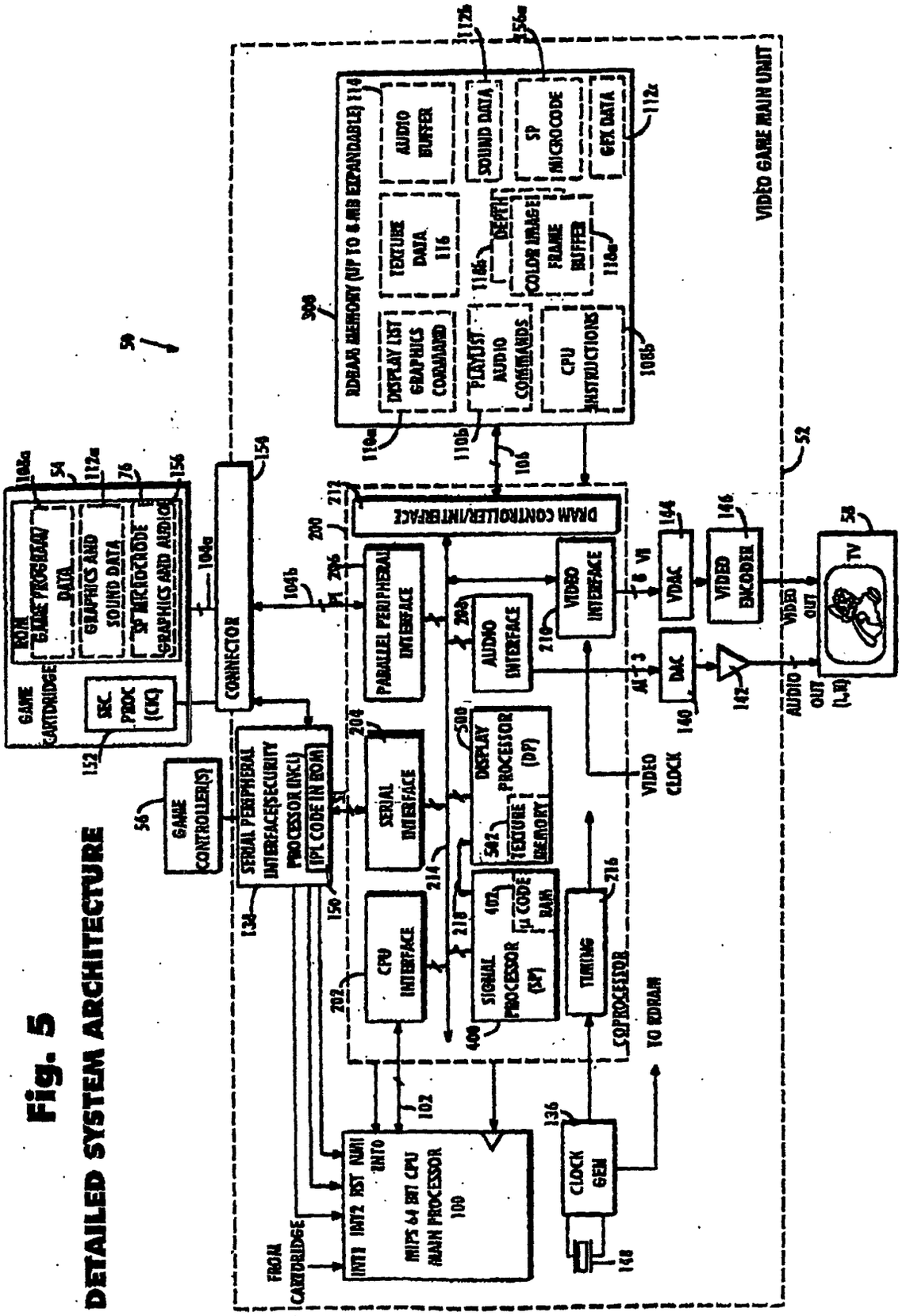
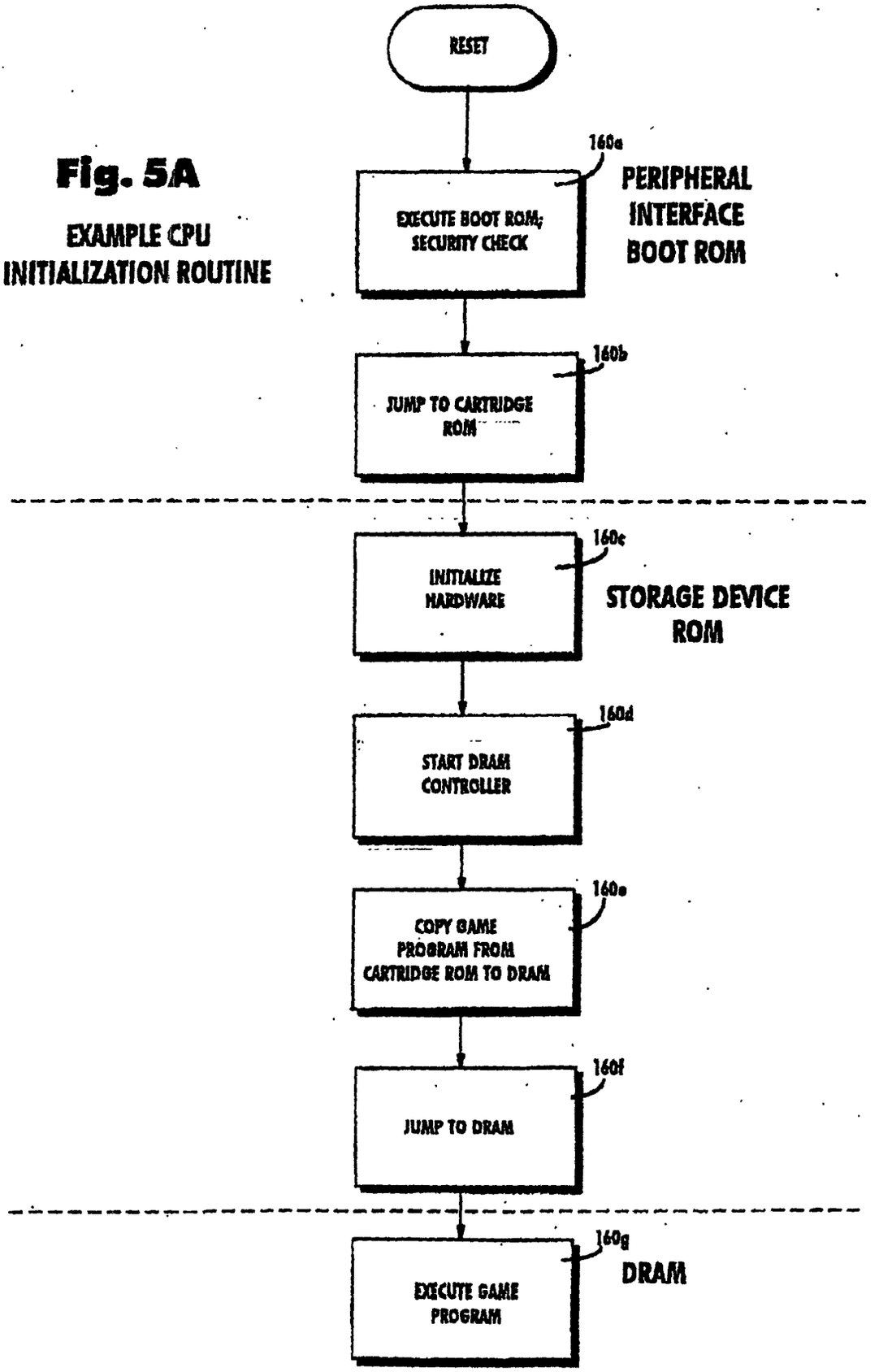


Fig. 5

DETAILED SYSTEM ARCHITECTURE

Fig. 5A

EXAMPLE CPU INITIALIZATION ROUTINE



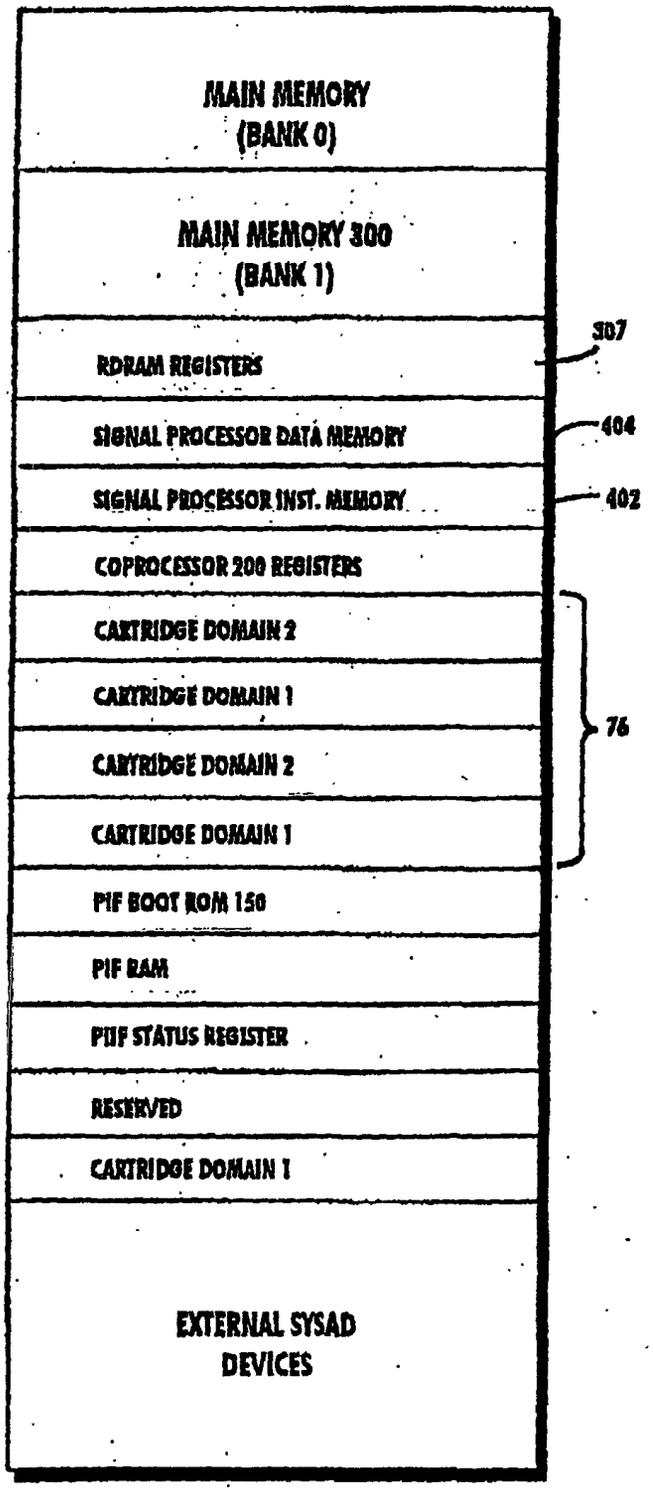
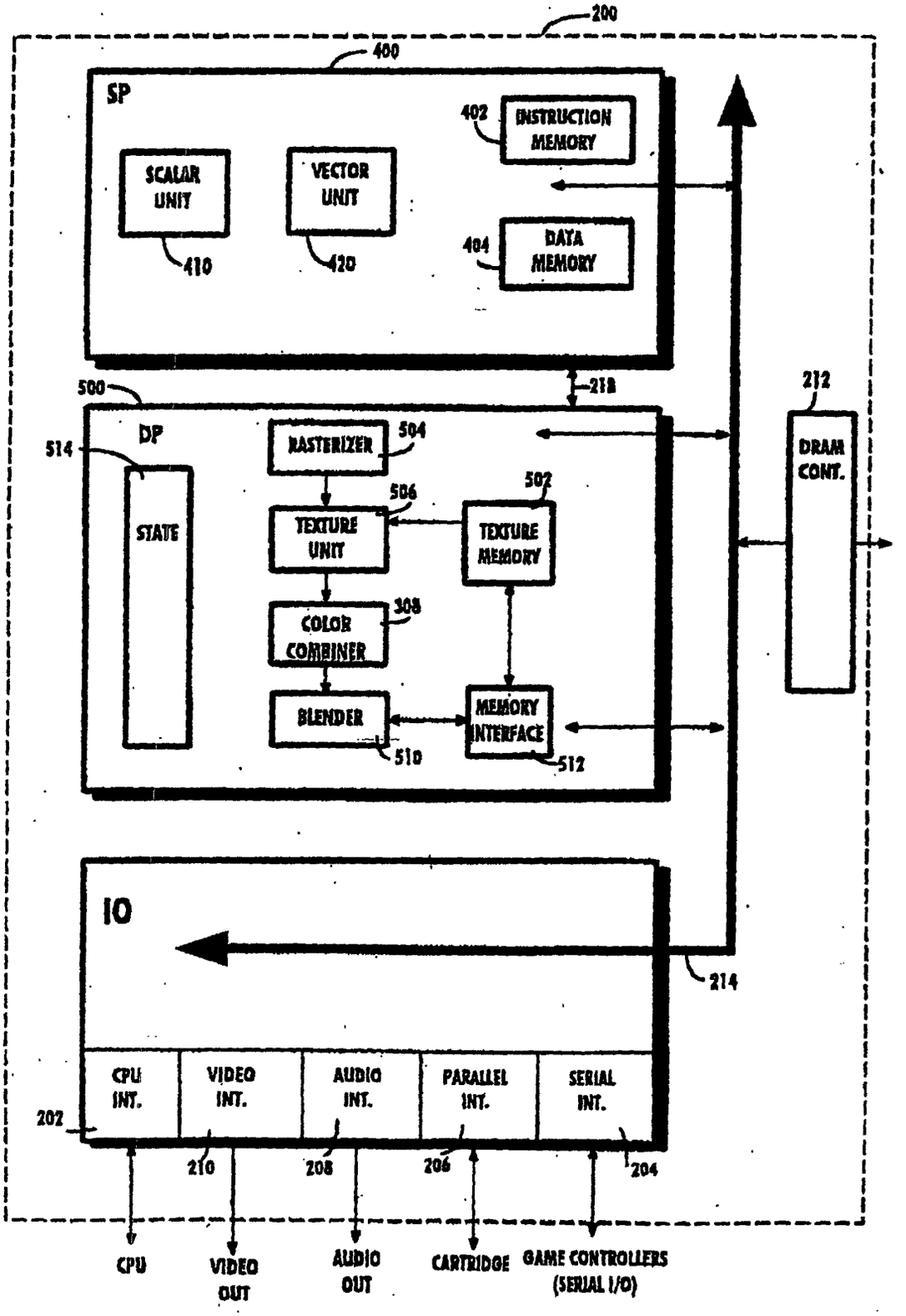


Fig. 5B EXAMPLE CPU MEMORY MAP

Fig. 6
COPROCESSOR



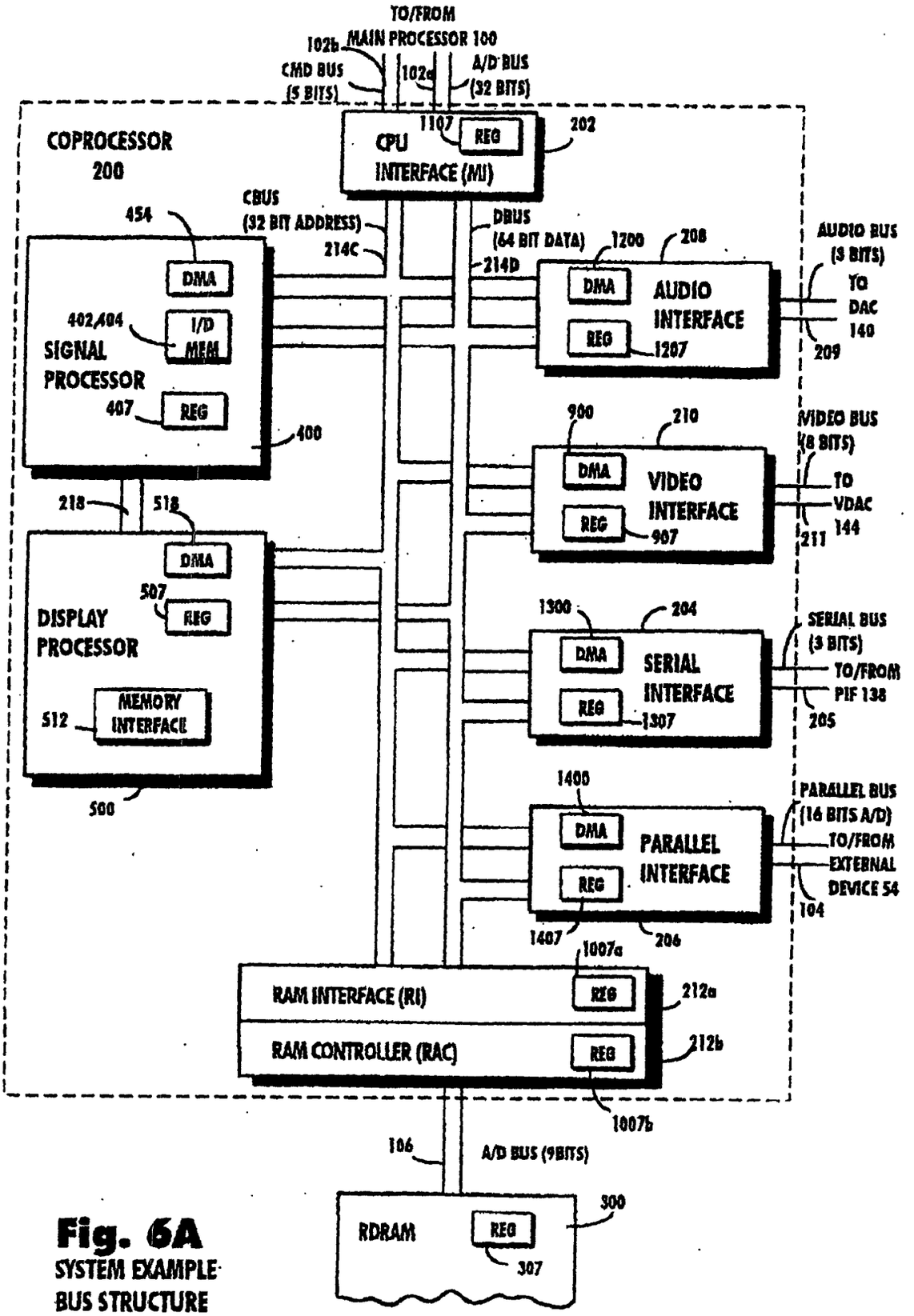


Fig. 6A
SYSTEM EXAMPLE BUS STRUCTURE

Fig. 7
SIGNAL PROCESSOR 400

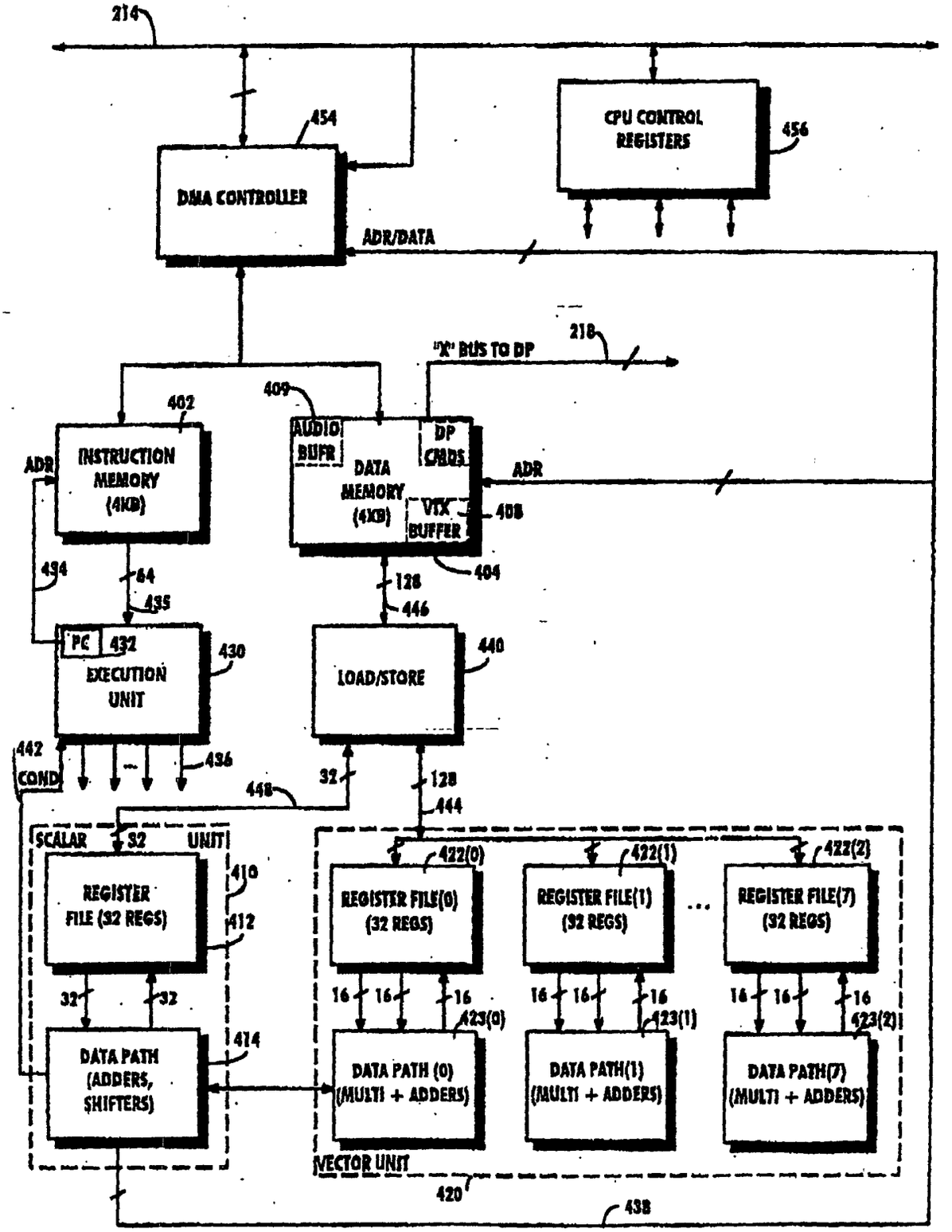


FIG. 7A

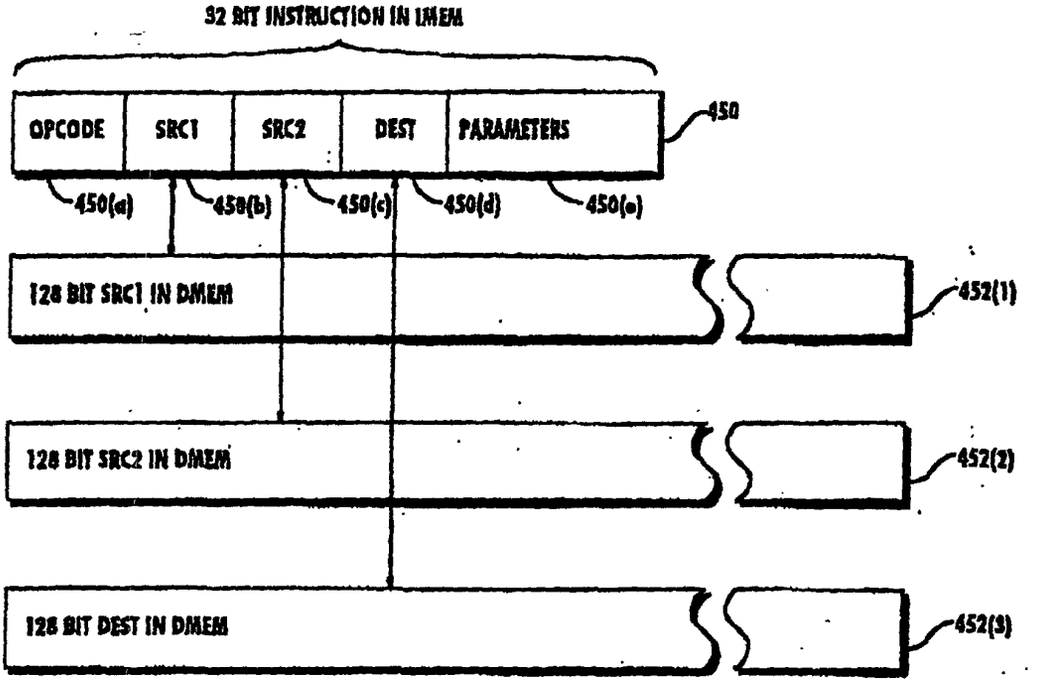


FIG. 7B

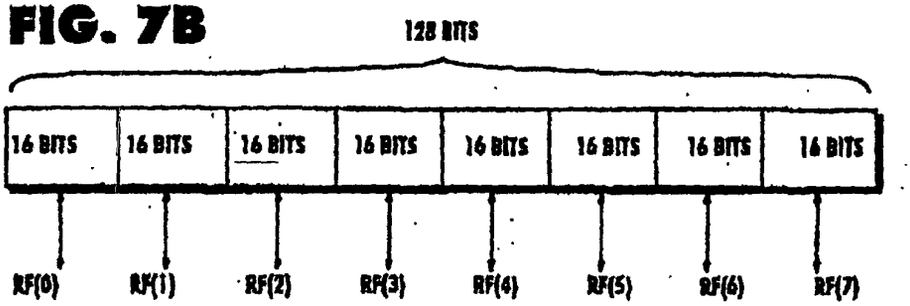


FIG. 7C

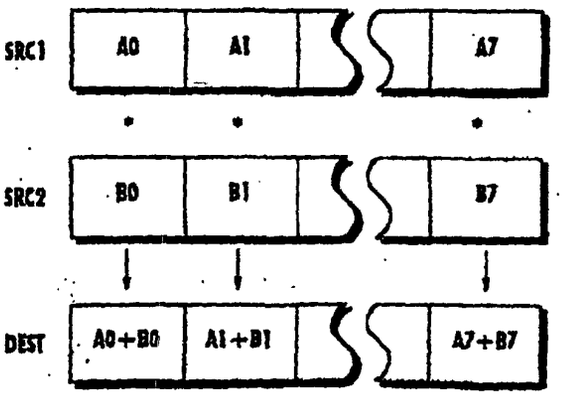


Fig. 7C SP DRAM DMA ADDRESS REGISTER (R/W)

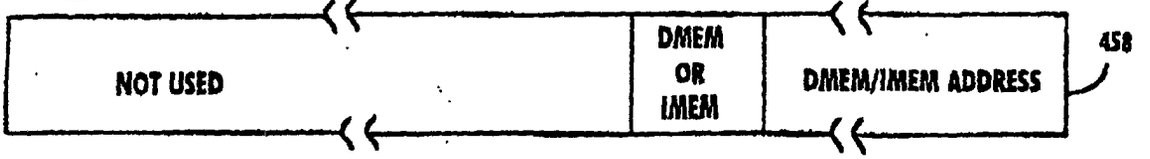


Fig. 7D SP MEMORY DMA ADDRESS (R/W)

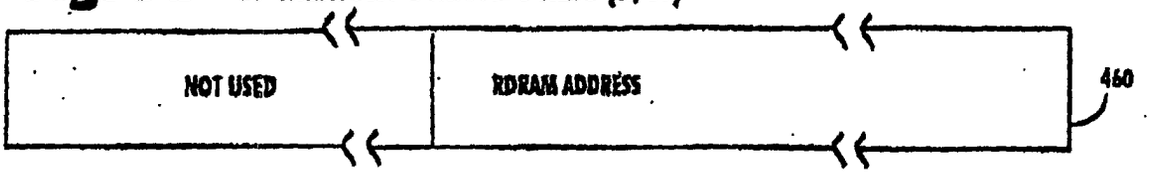


Fig. 7E SP READ DMA LENGTH (R/W)

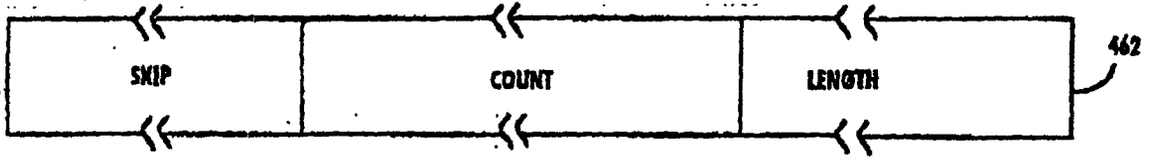


Fig. 7F SP WRITE DMA LENGTH (R/W)

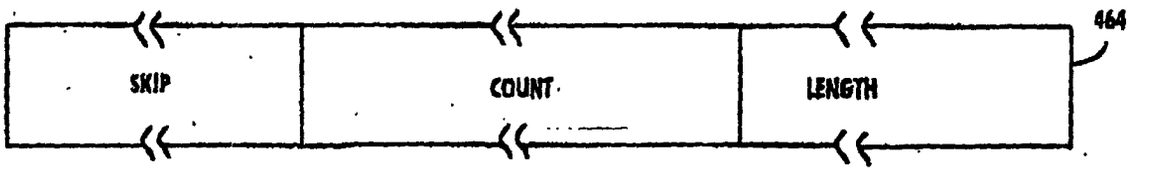


Fig. 7G SP DMA FULL REGISTER (R)

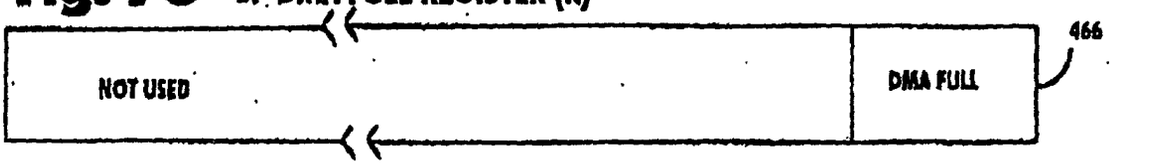
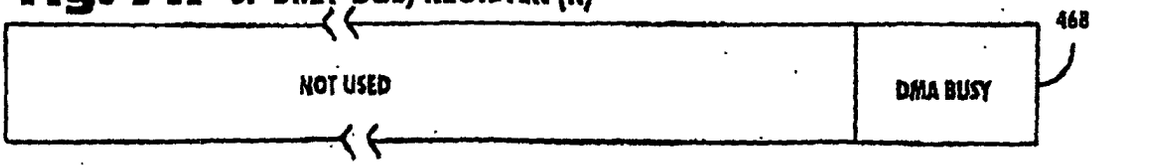


Fig. 7H SP DMA BUS, REGISTER (R)



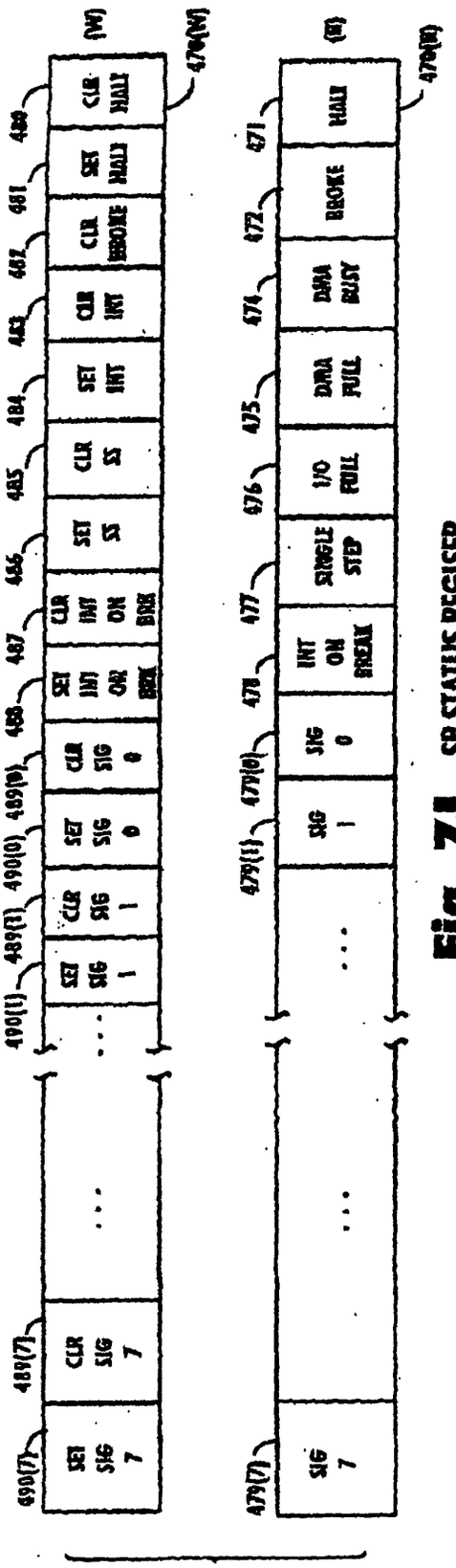


Fig. 7J SP STATUS REGISTER

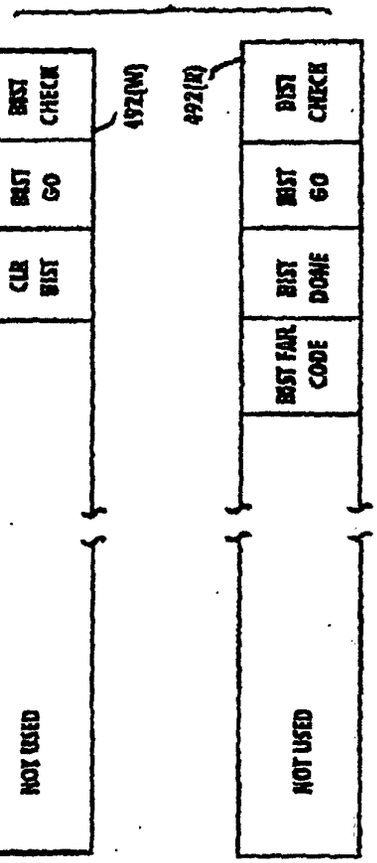


Fig. 7L SP IMEM BIST STATUS REGISTER

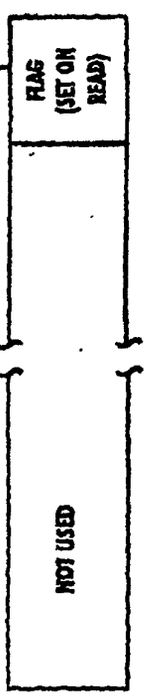


Fig. 7K SP SEMAPHORE (R) (WRITE CLEARS FLAG)

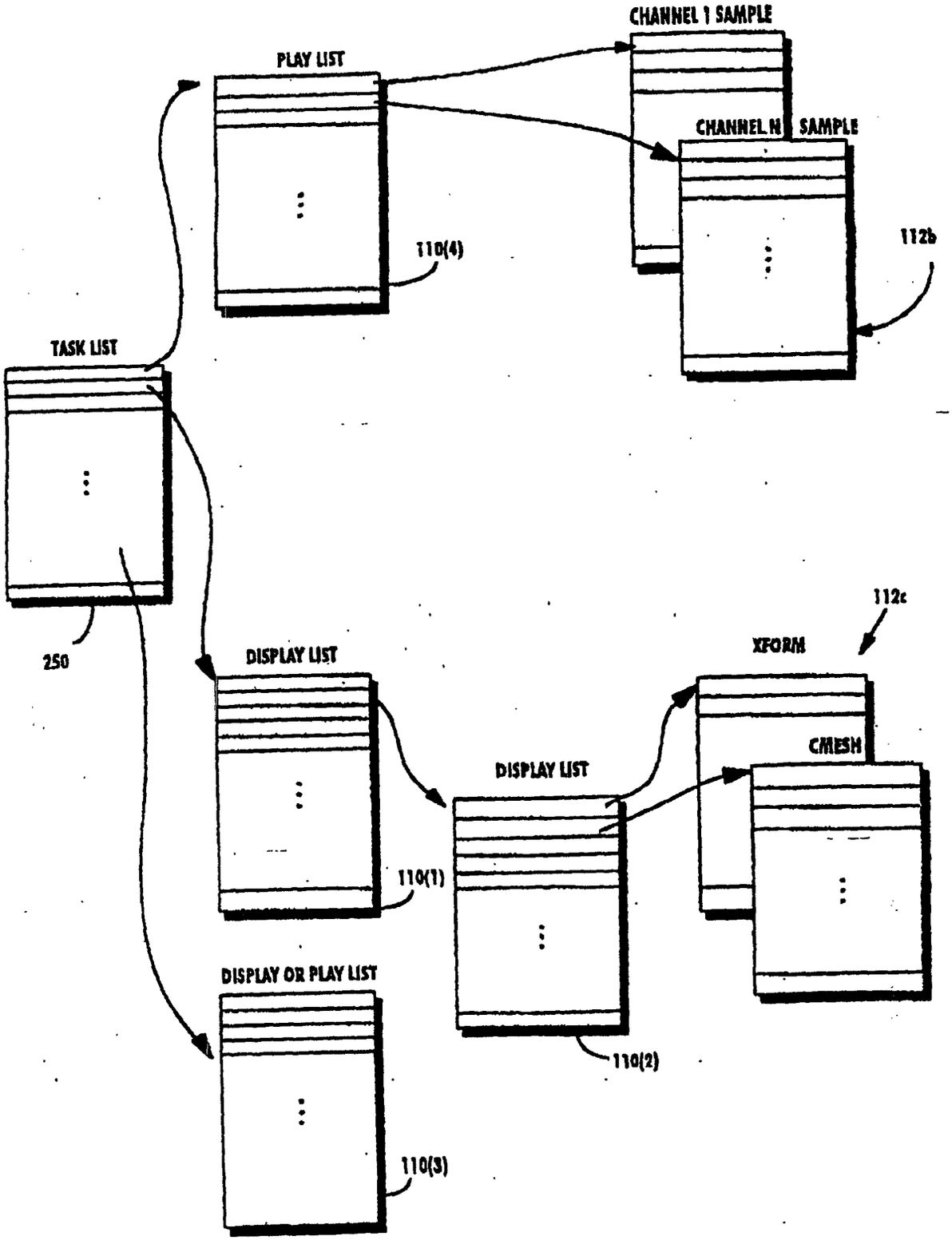


Fig. 8 TASK LIST

Fig. 9 EXAMPLE MICROCODE LOAD ROUTINE

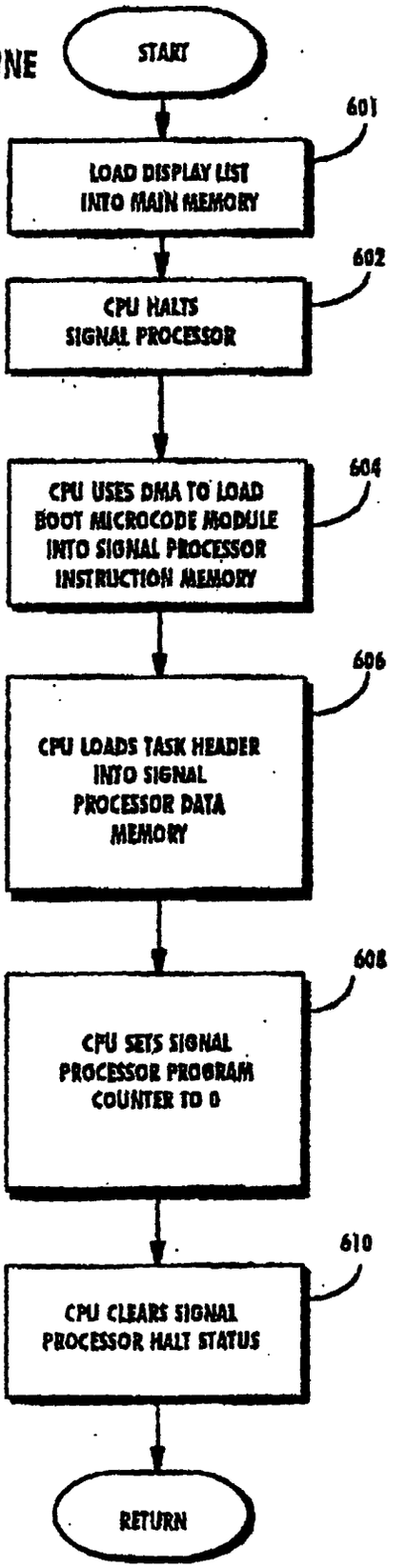
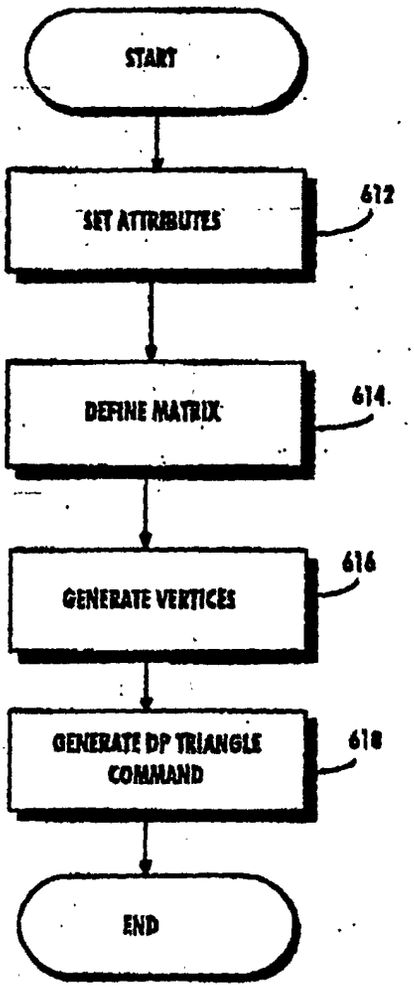


Fig. 10
SIMPLE SP DISPLAY LIST PROCESS EXAMPLE



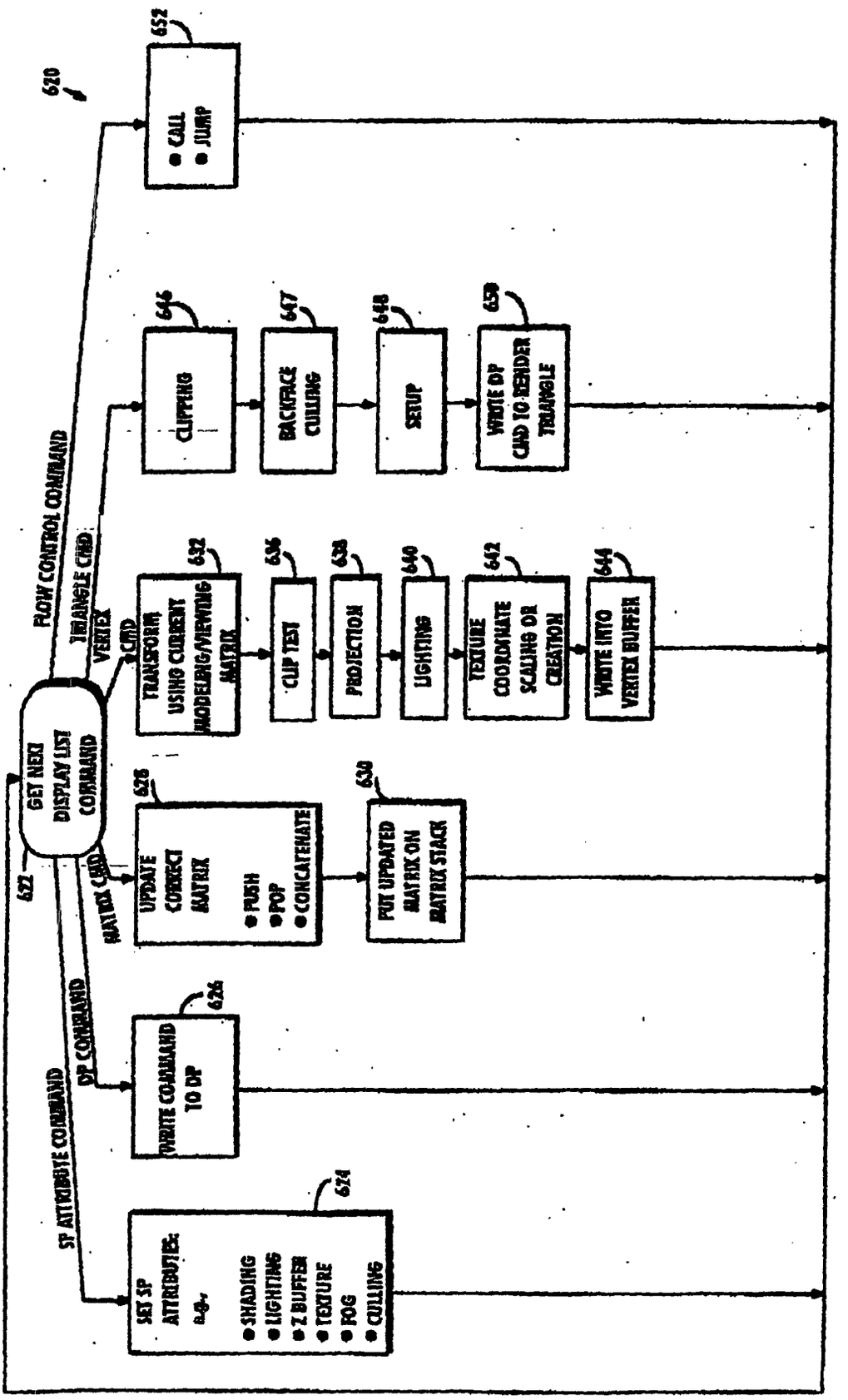


Fig. 11
SIGNAL PROCESSOR
MICROCODE-GRAPHICS

FIG. 12A

EXAMPLE "DOUBLE PRECISION" VALUE

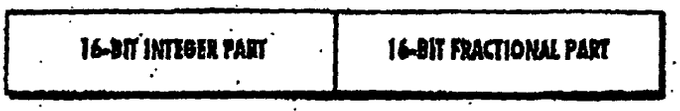


FIG. 12B

EXAMPLE MATRIX FORMAT

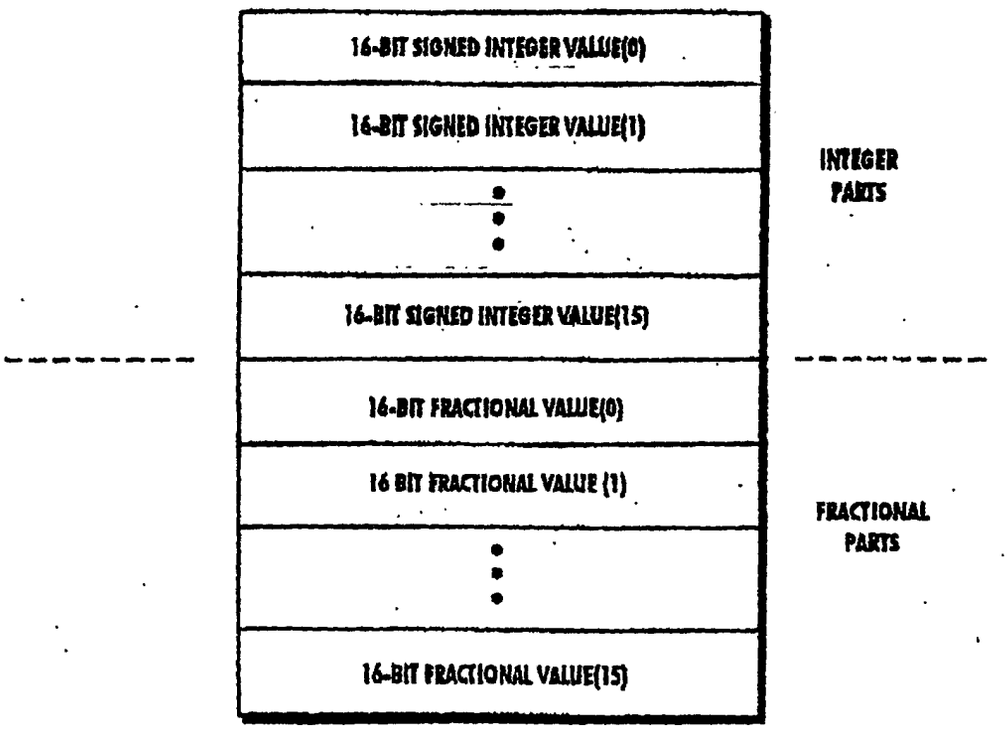
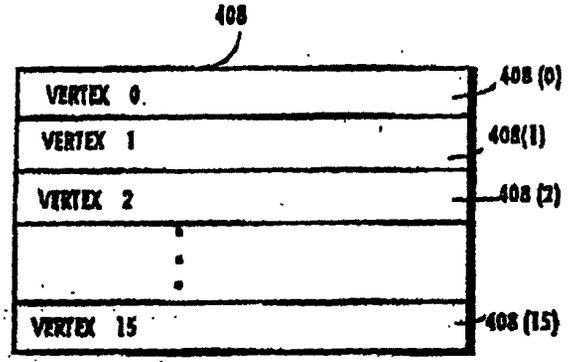
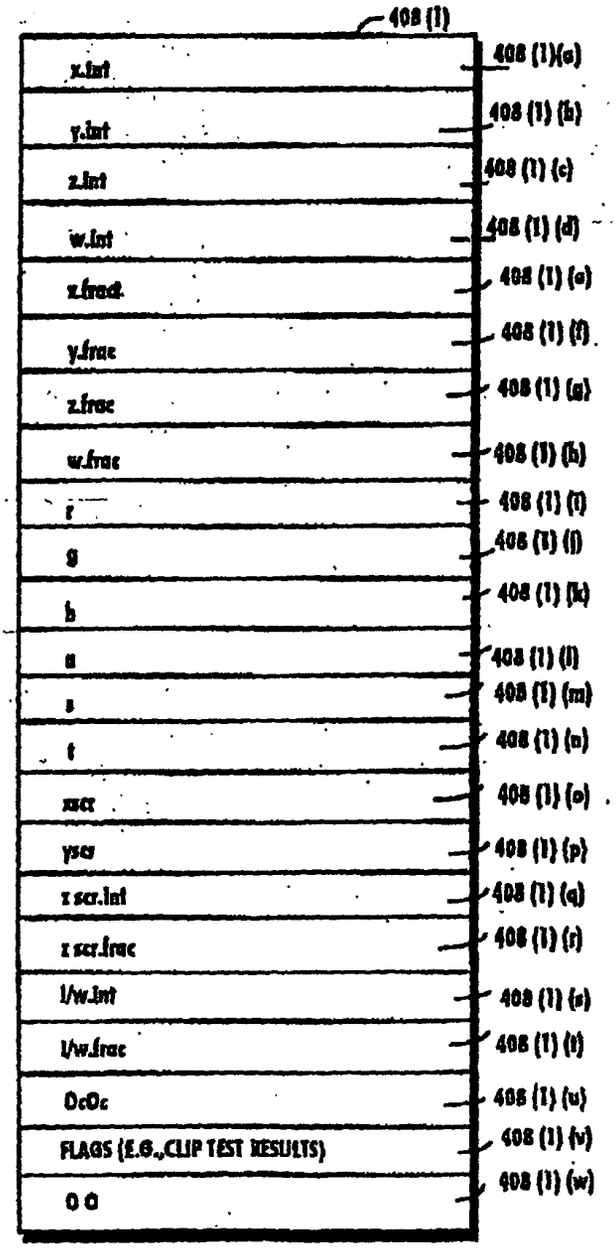


FIG. 13A

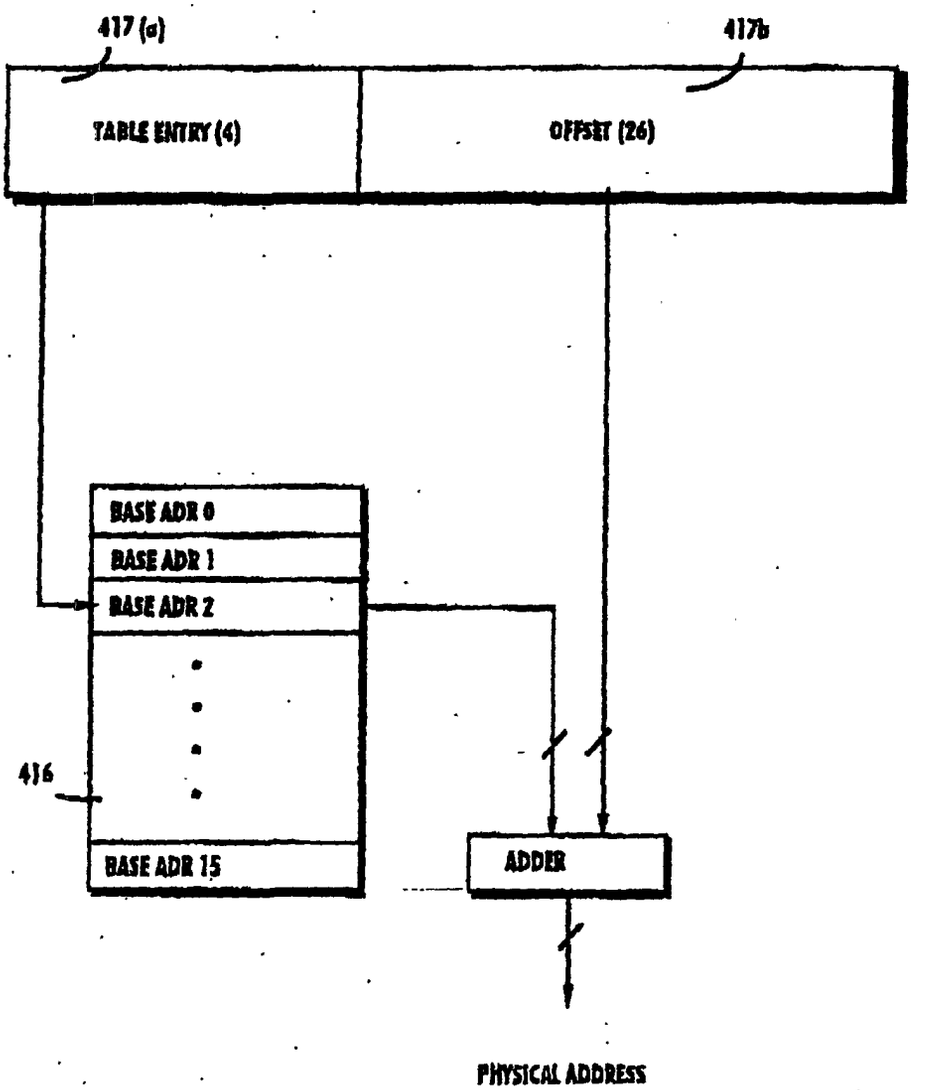


VERTEX BUFFER

FIG. 13B



EXAMPLE VERTEX DATA STRUCTURE



EXAMPLE SIGNAL PROCESSOR SEGMENT ADDRESSING

FIG. 13C

Fig. 14 EXAMPLE AUDIO SOFTWARE ARCHITECTURE

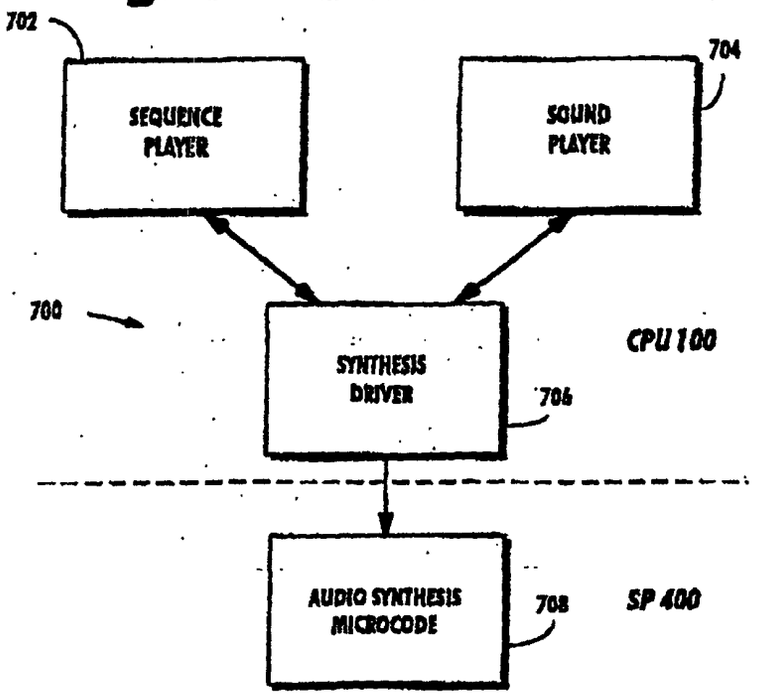


Fig. 17 EXAMPLE SP AUDIO PROCESSING

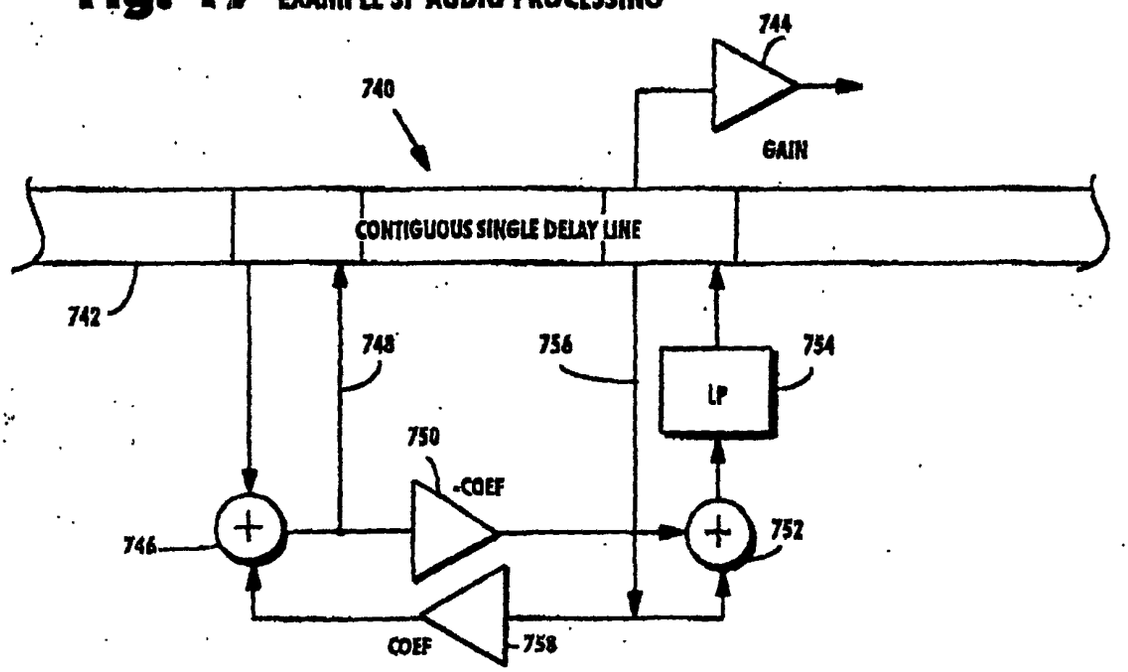
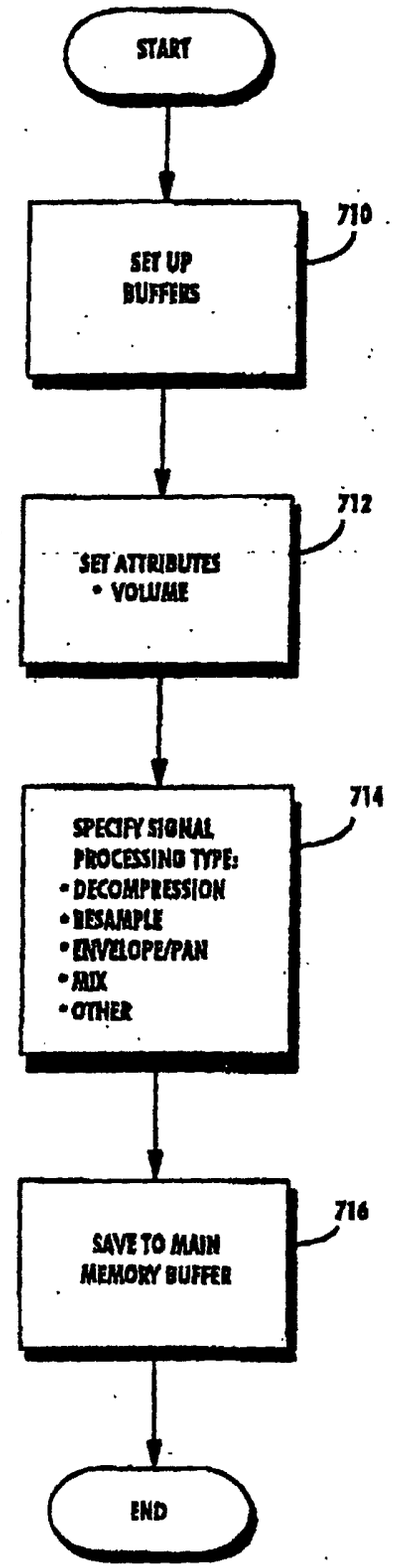


Fig. 15 SIMPLE SP PLAY LIST PROCESS EXAMPLE



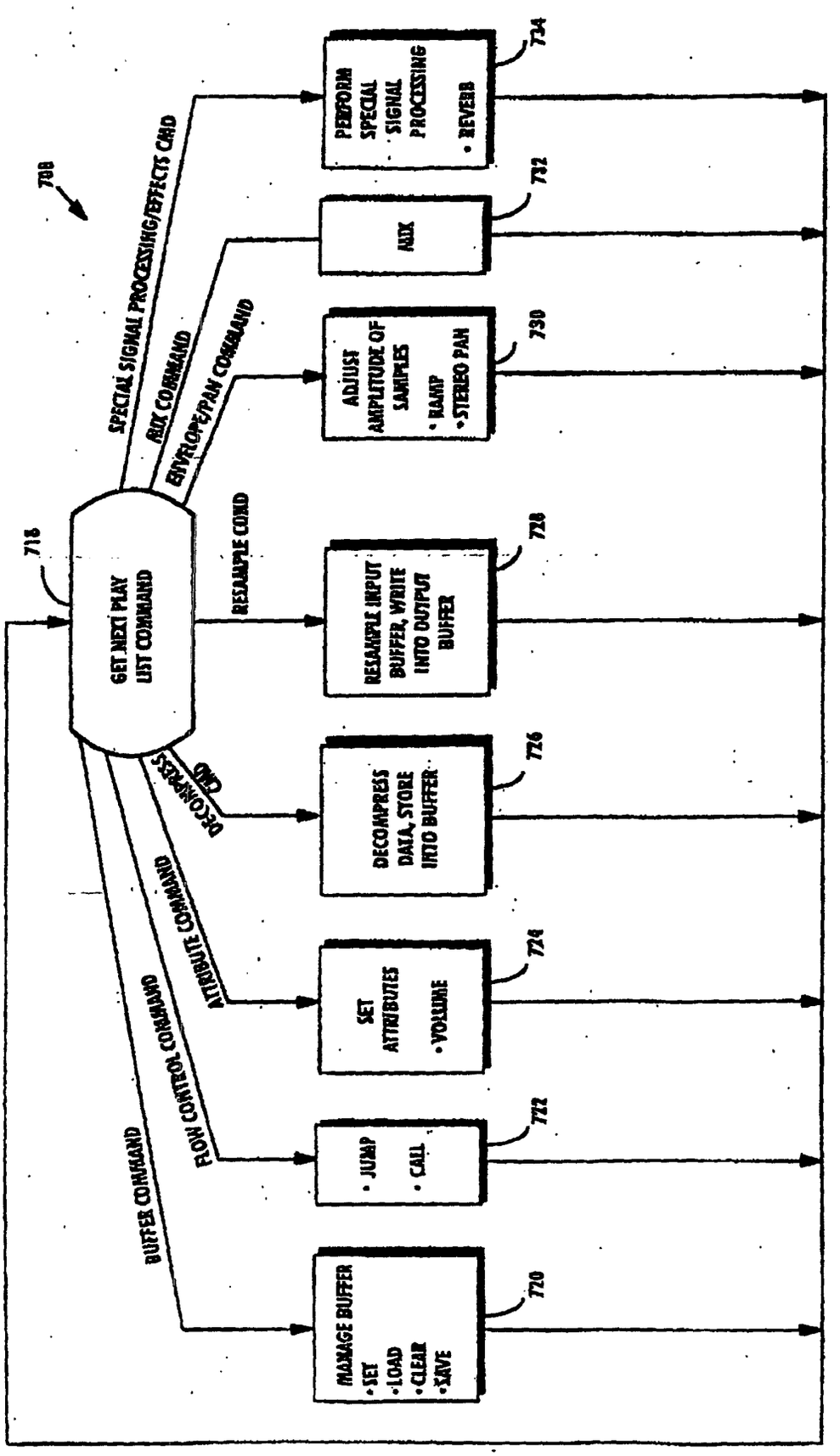
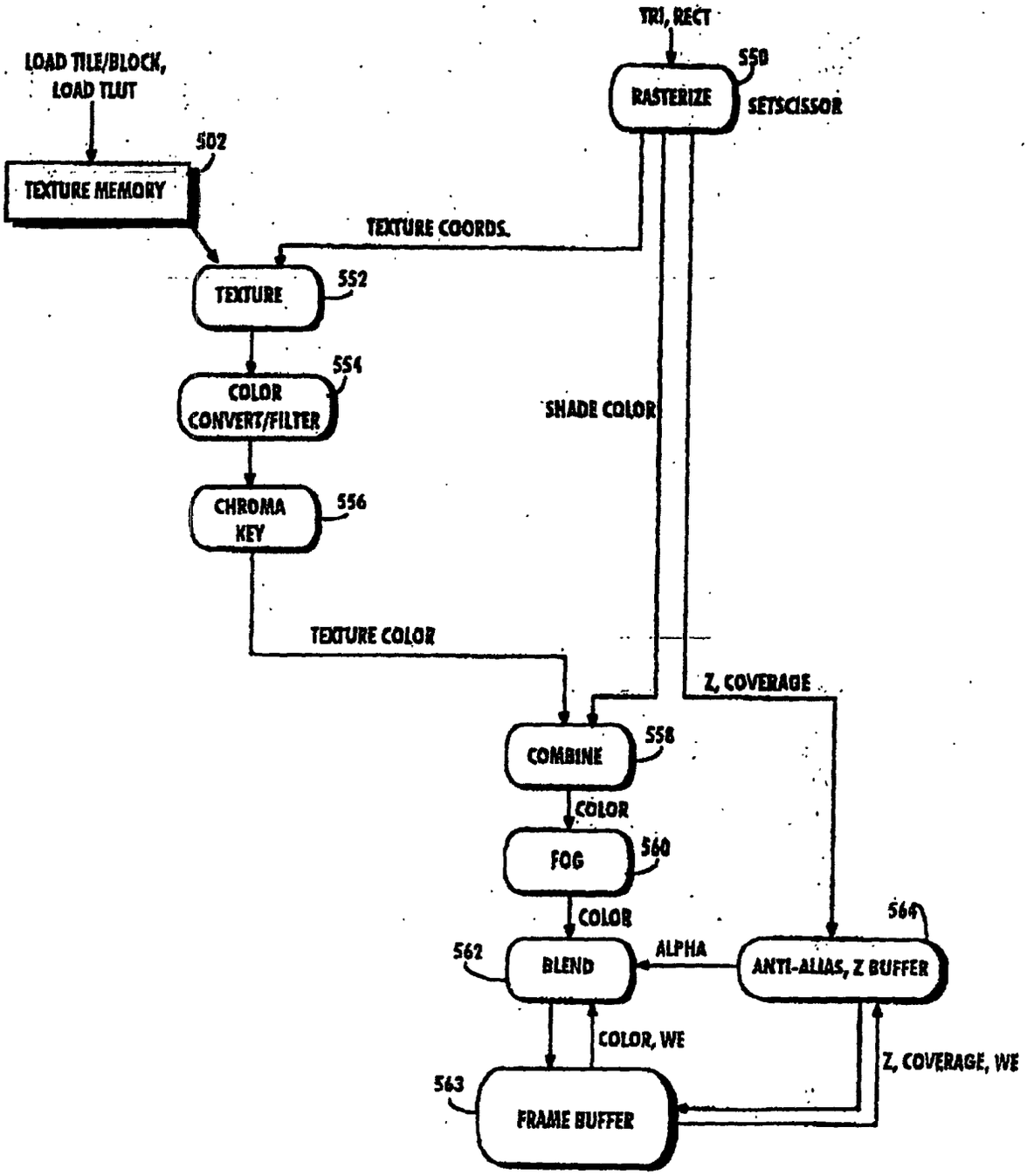


Fig. 16 SIGNAL PROCESSOR MICROCODE - AUDIO

Fig. 18
DISPLAY PROCESSOR PROCESSING



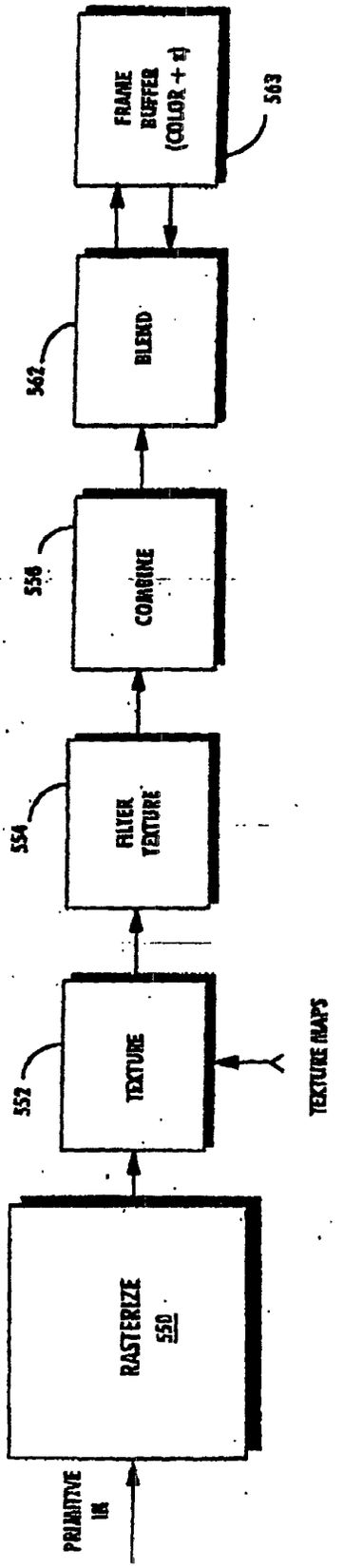


Fig. 19A DISPLAY PROCESSOR PIPELINE - 1 CYCLE MODE

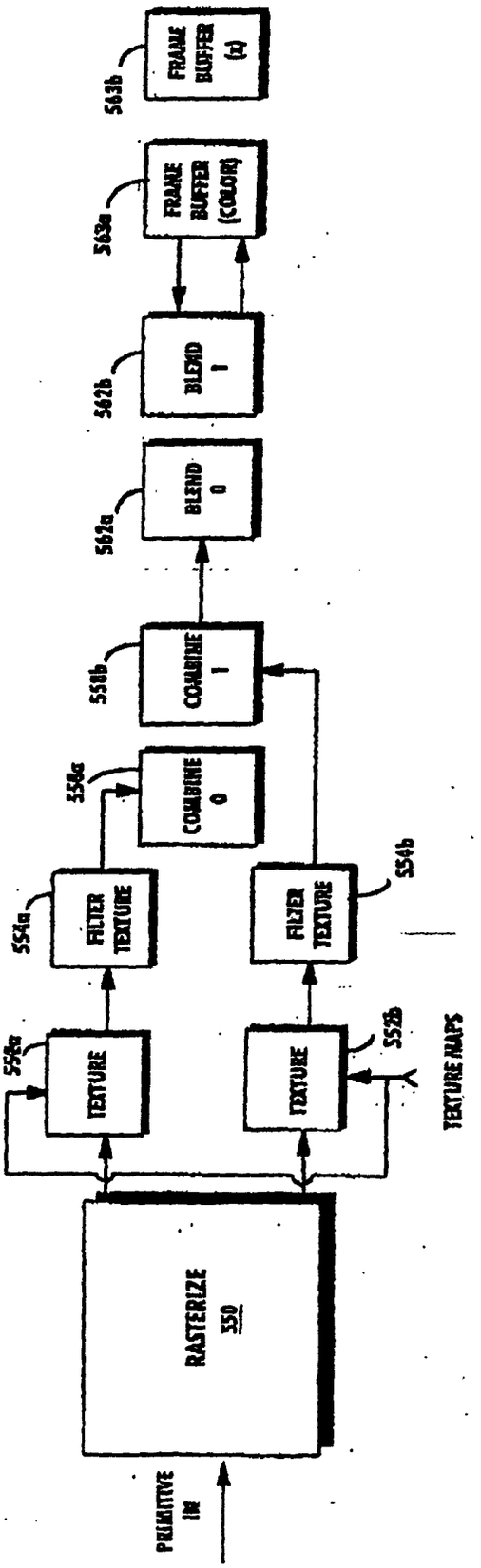


Fig. 19B DISPLAY PROCESSOR PIPELINE - 2 CYCLE MODE

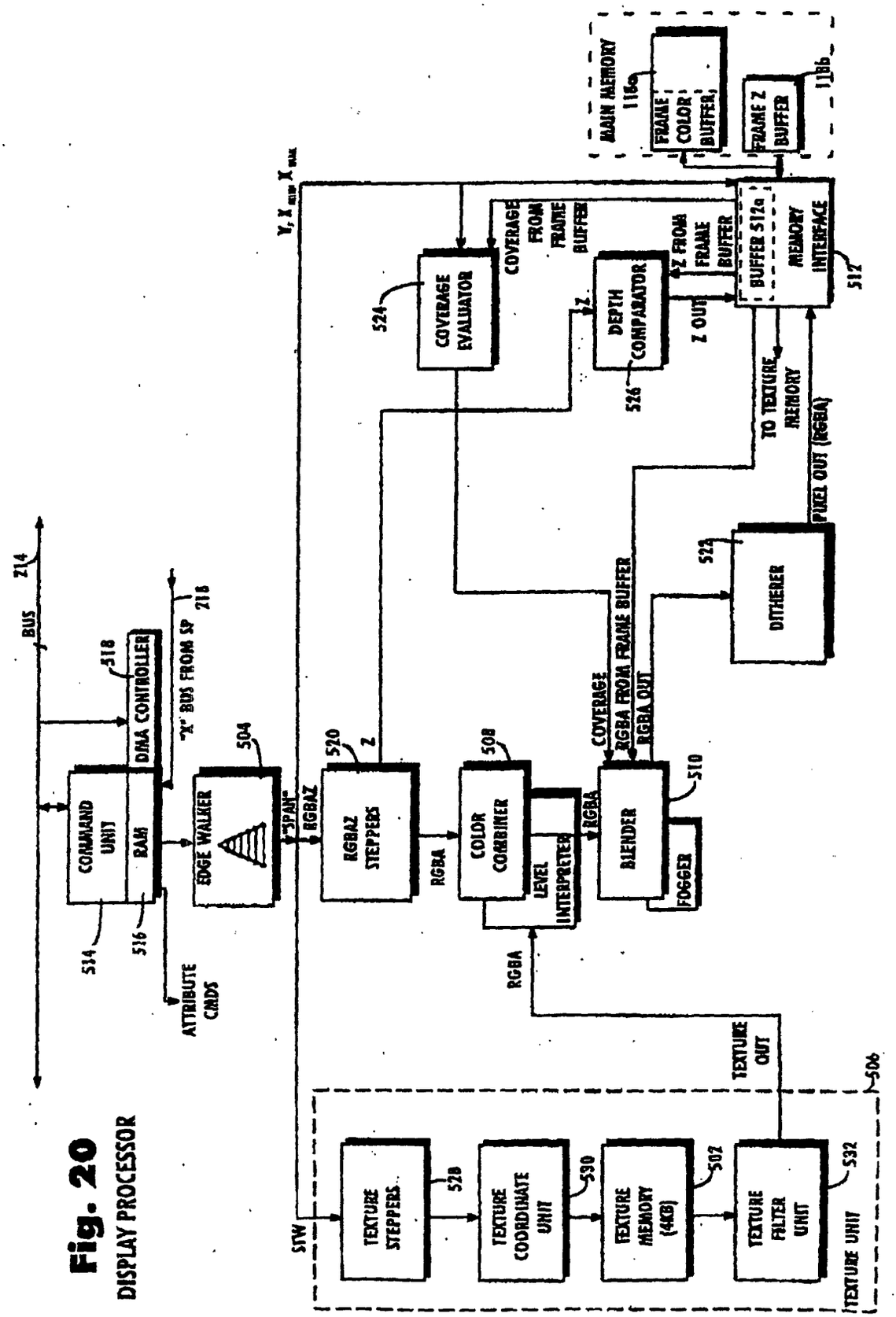


Fig. 20
DISPLAY PROCESSOR

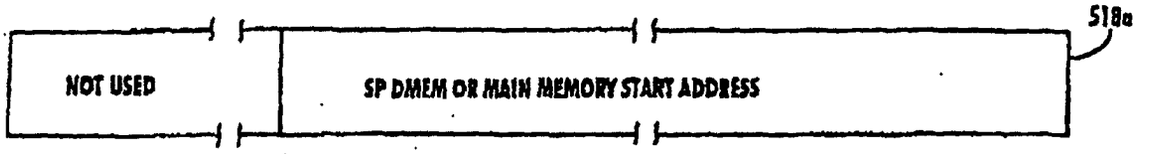


Fig. 21A DP CMD DMA START ADDRESS REGISTER (R/W)

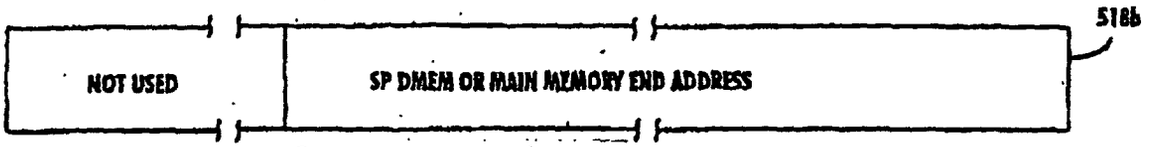


Fig. 21B DP CMD DMA END ADDRESS REGISTER (R/W)

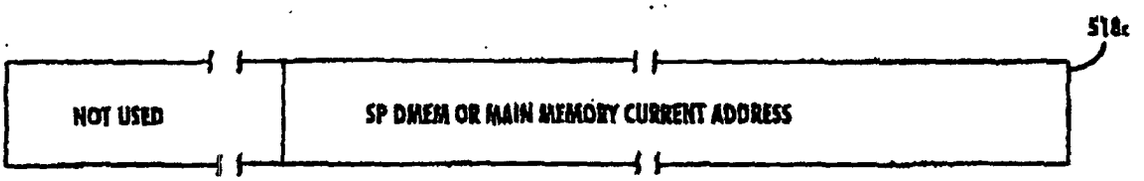


Fig. 21C DP CMD DMA CURRENT ADDRESS REGISTER (R/W)

Fig. 21E CLOCK COUNTER (R)

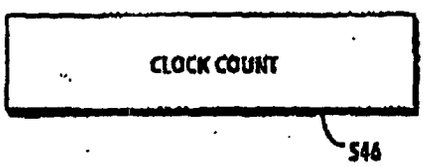


Fig. 21F DP PIPELINE BUSY CTR (R)

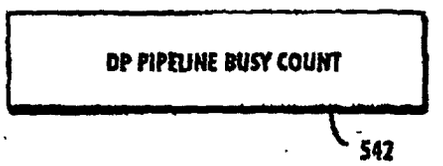


Fig. 21G CMD BUFFER BUSY CTR (R)

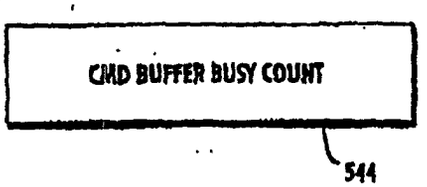


Fig. 21H TMEM LOCAL CTR (R)

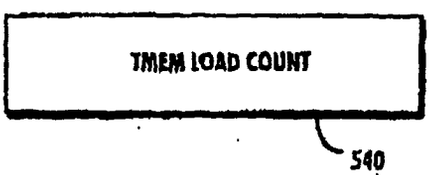
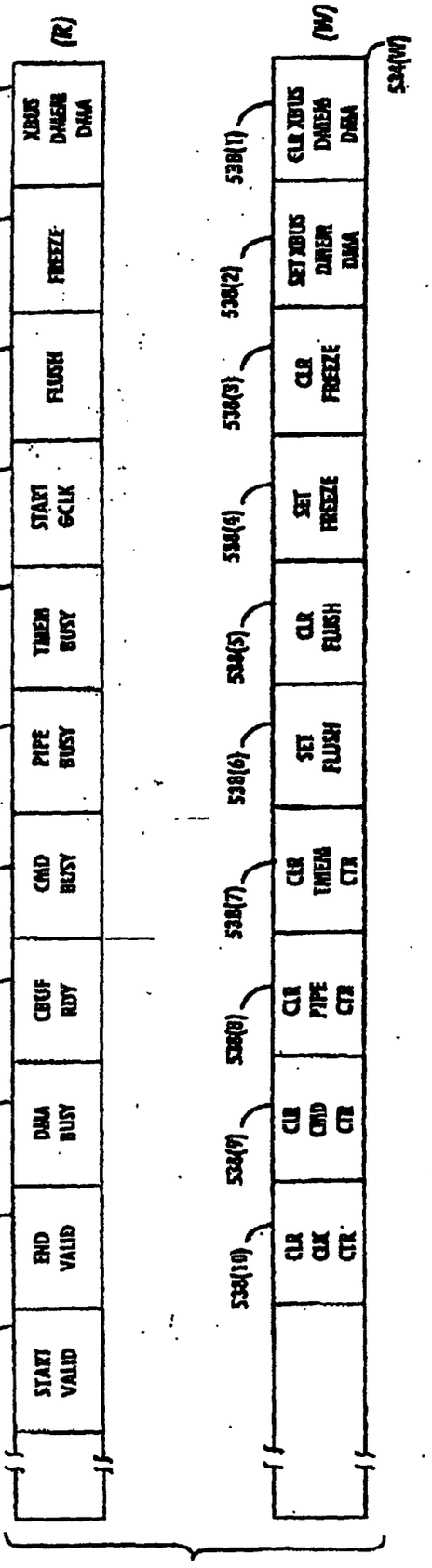


Fig. 21D DP CMD STATUS/COMMAND REGISTER



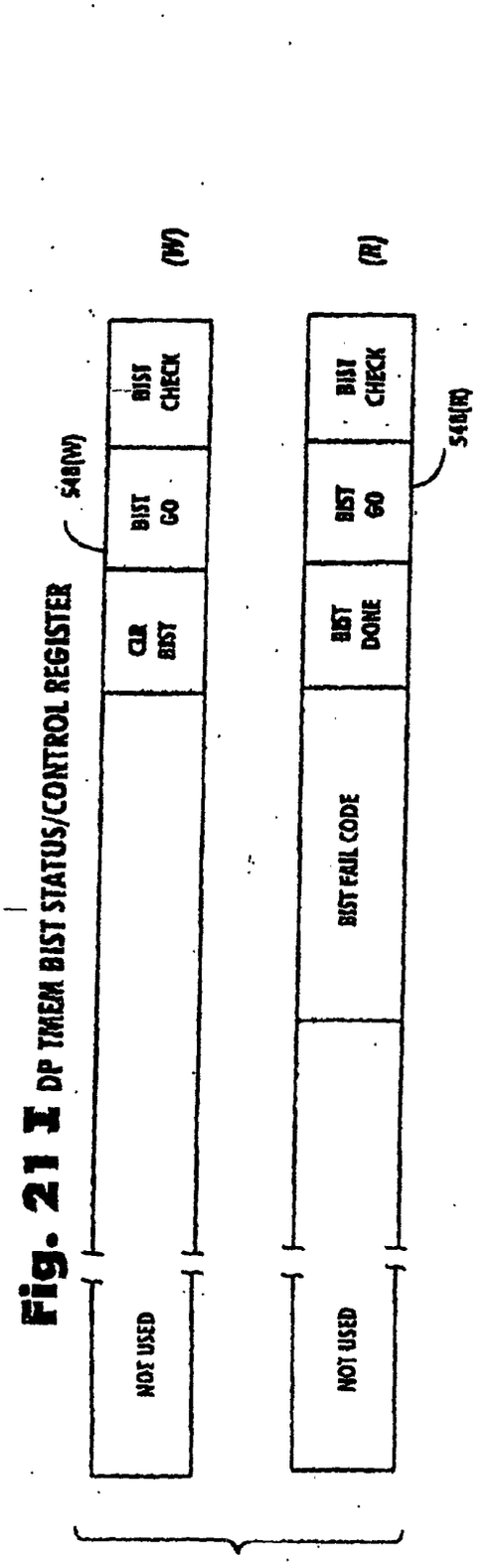


Fig. 21I DP TMEM BIST STATUS/CONTROL REGISTER

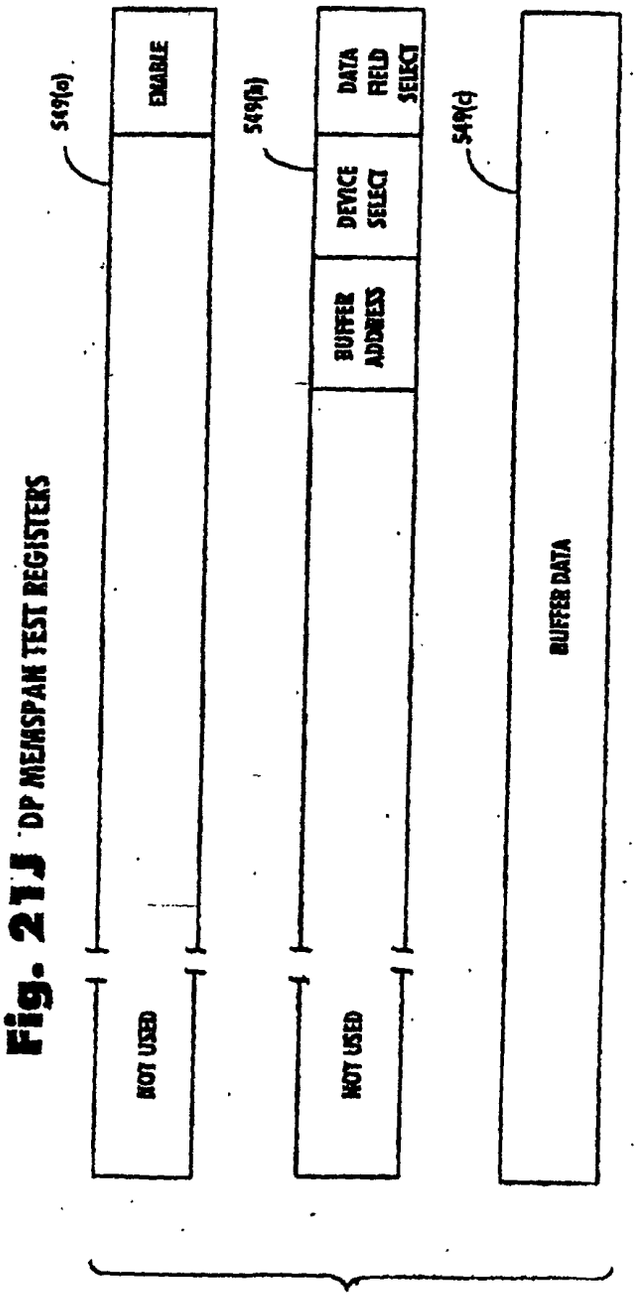
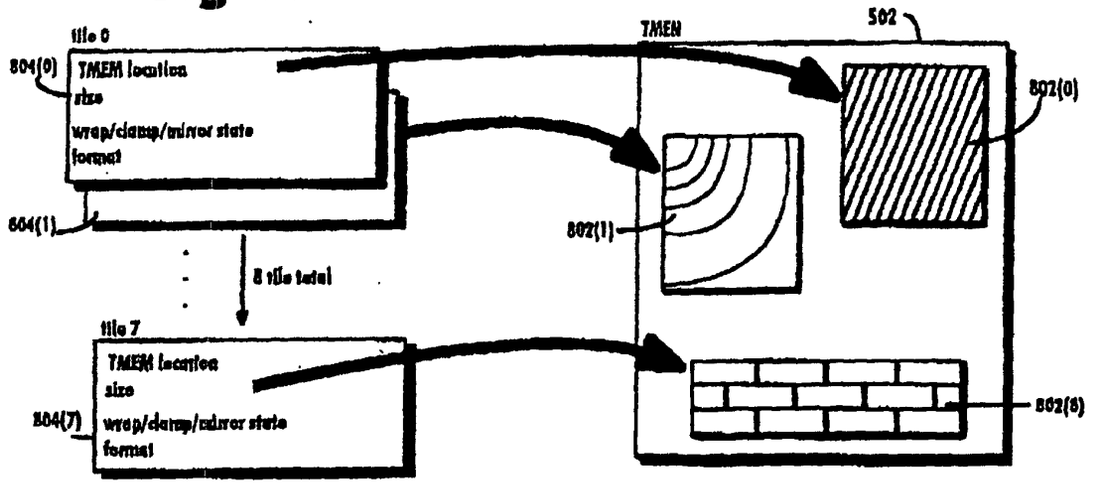


Fig. 21J DP MEMSPAN TEST REGISTERS

Fig. 22 TEXTURE MEMORY DESCRIPTORS



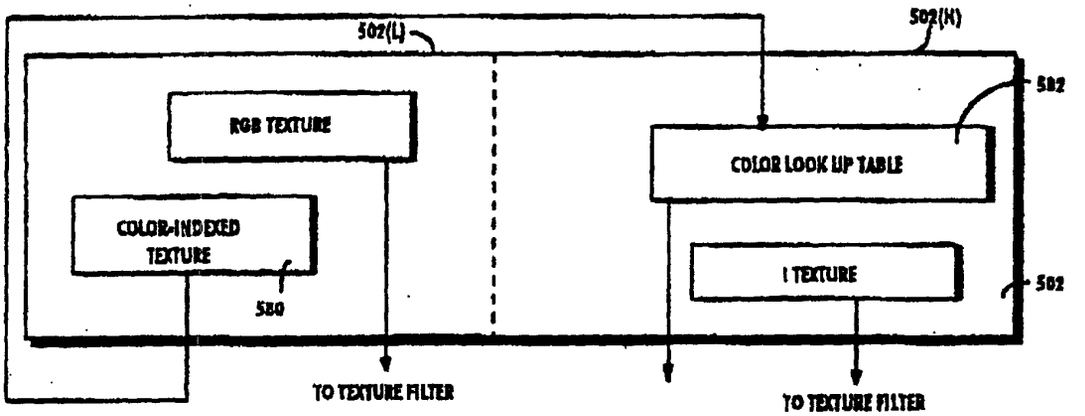


Fig. 25 TEXTURE MEMORY STORING COLOR INDEXED TEXTURES

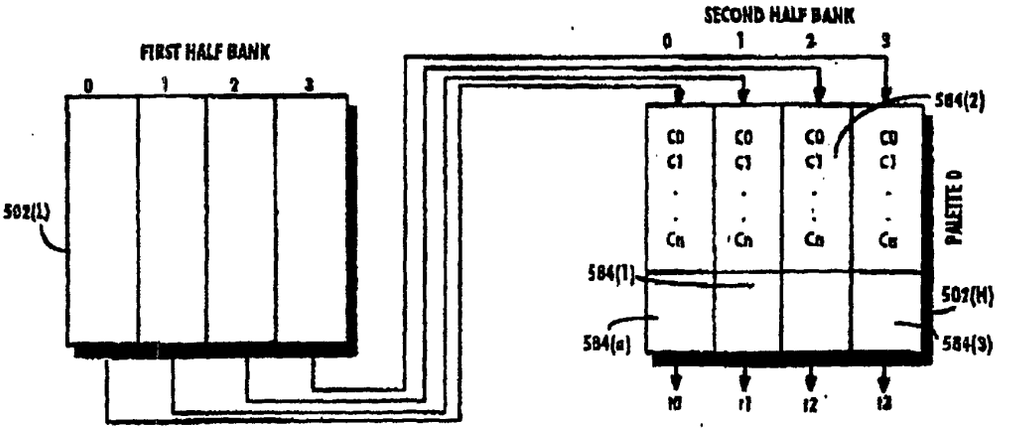
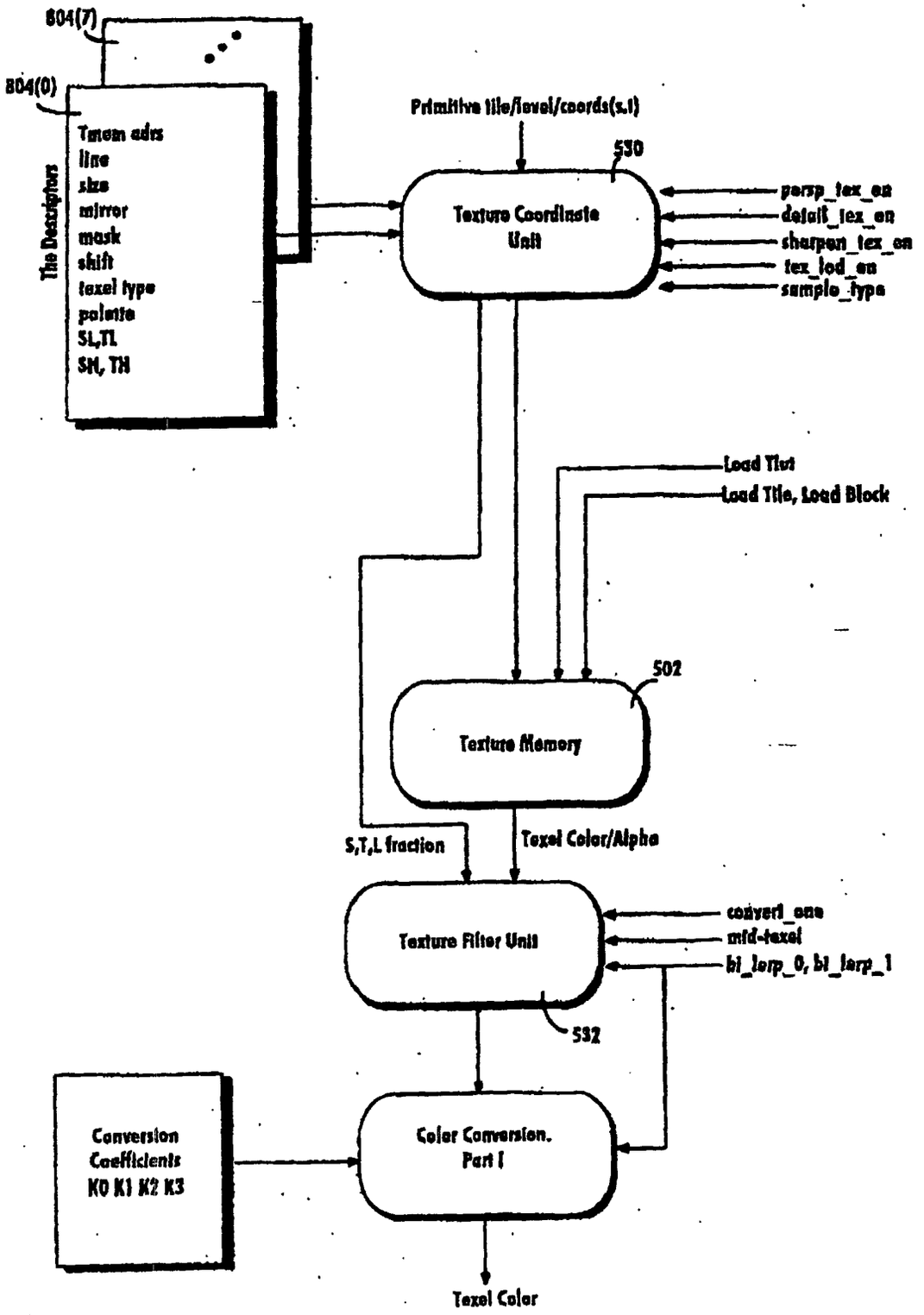


Fig. 26 DETAILED USE OF TEXTURE MEMORY TO STORE COLOR INDEXED TEXTURES

Fig. 23 TEXTURE UNIT EXAMPLE PROCESSES



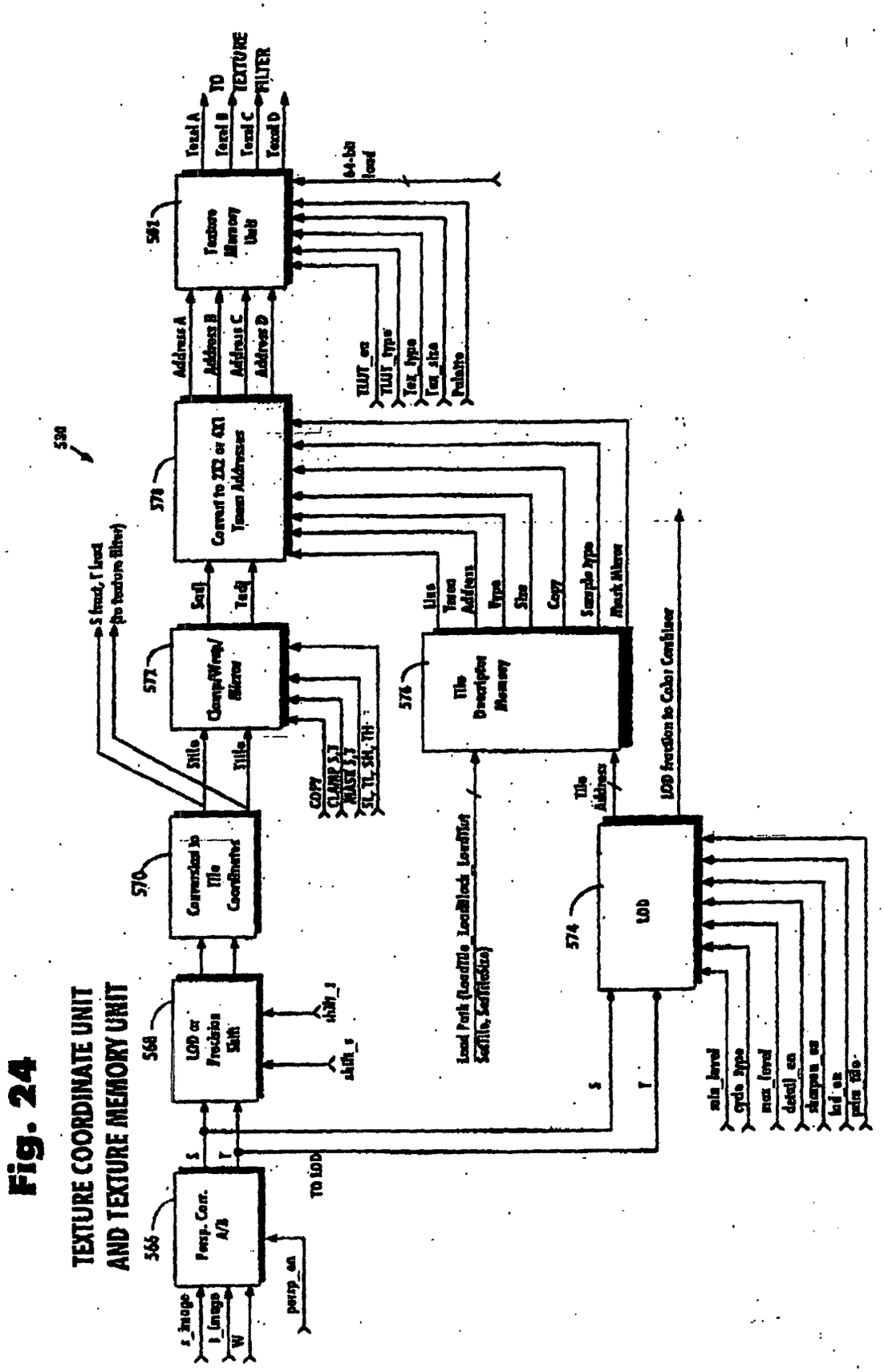

TEXTURE COORDINATE UNIT
AND TEXTURE MEMORY UNIT
Texel A
Texel B
Texel C
Texel D
TD
TEXTURE
FILTER
581
Texture Memory Unit
64-bit Texel
Address A
Address B
Address C
Address D
TEXT_en
TEXT_type
Tex_size
Palette
530
578
Convert to 2X2 or 4X1 Texture Addresses
Sout
Tout
Line
Texel Address
Type
Size
Copy
Sample type
Mask Mirror
572
Clamp/Wrap/ Mirror
576
Tile Descriptor Memory
SMin
TMin
COPY
CLAMP S,T
MASK S,T
SL_TL_SH_TH
Tile Address
LOD function to Color Combiner
570
Conversion to Tile Coordinates
574
LOD
Level Path (LevelTile, LevelBlock, LevelOffset, SubTile, SubTileBlock)
S
T
568
LOD or Precision Shift
shift_t
shift_s
TO LOD
566
Persp. Corr. 1/S
s_Image
t_Image
W
persp_en
min_level
cycle_type
max_level
detail_en
clamp_en
lod_en
pan_tile

Fig. 24

Fig. 27 RGB COLOR COMBINER EXAMPLE INPUT SELECTION

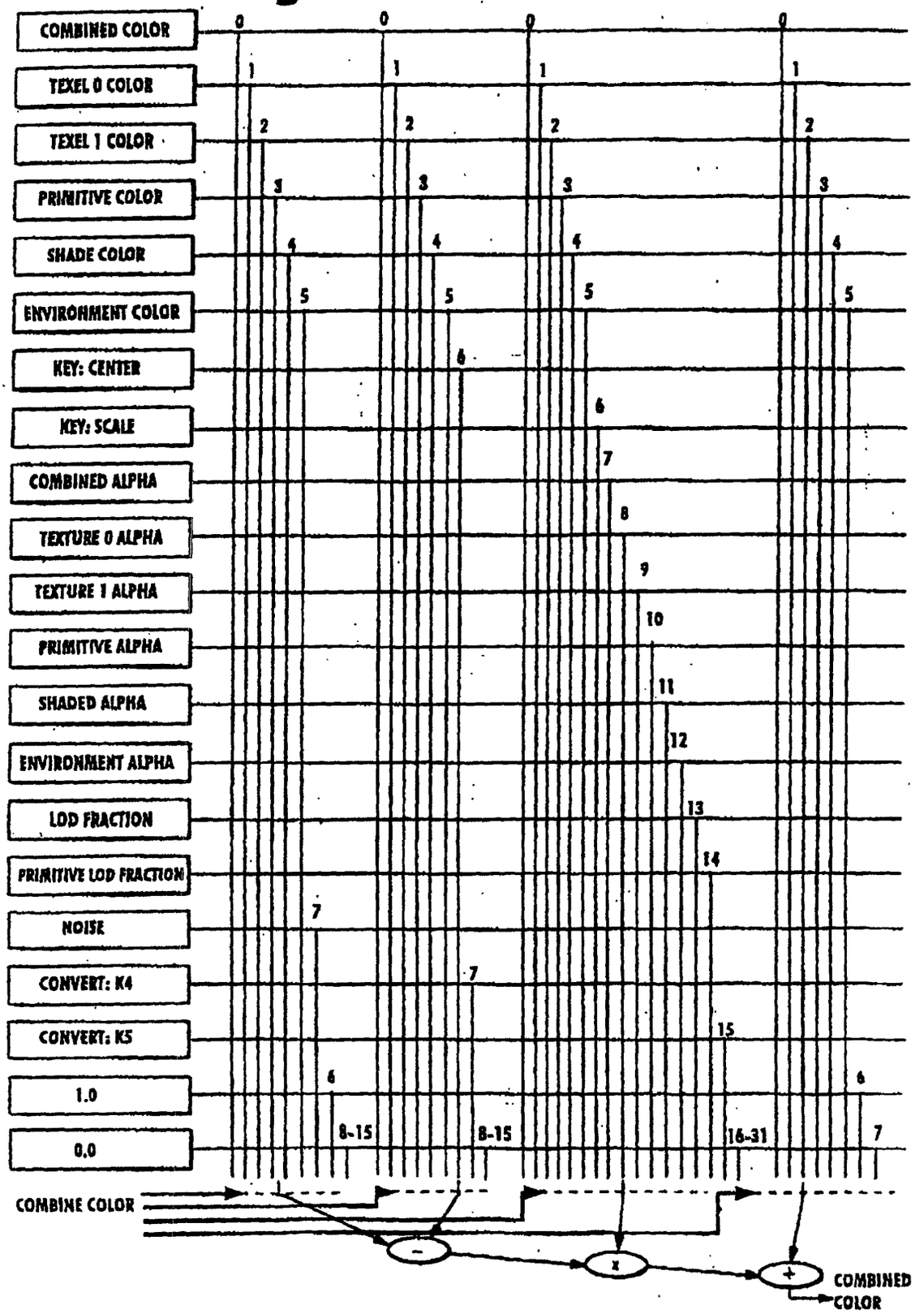


Fig. 28 ALPHA COMBINER EXAMPLE INPUT SELECTION

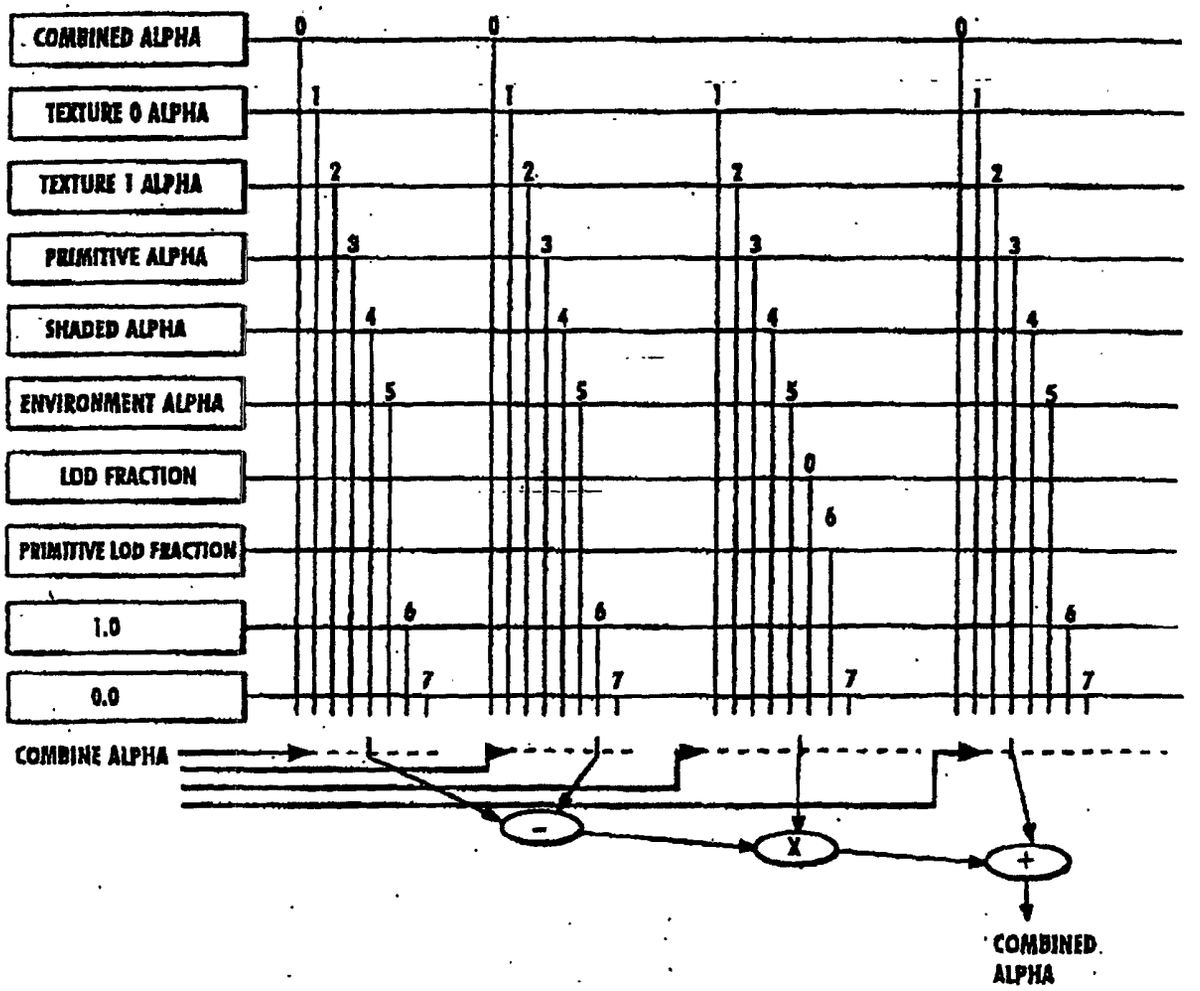


Fig. 29
ALPHA FIXUP

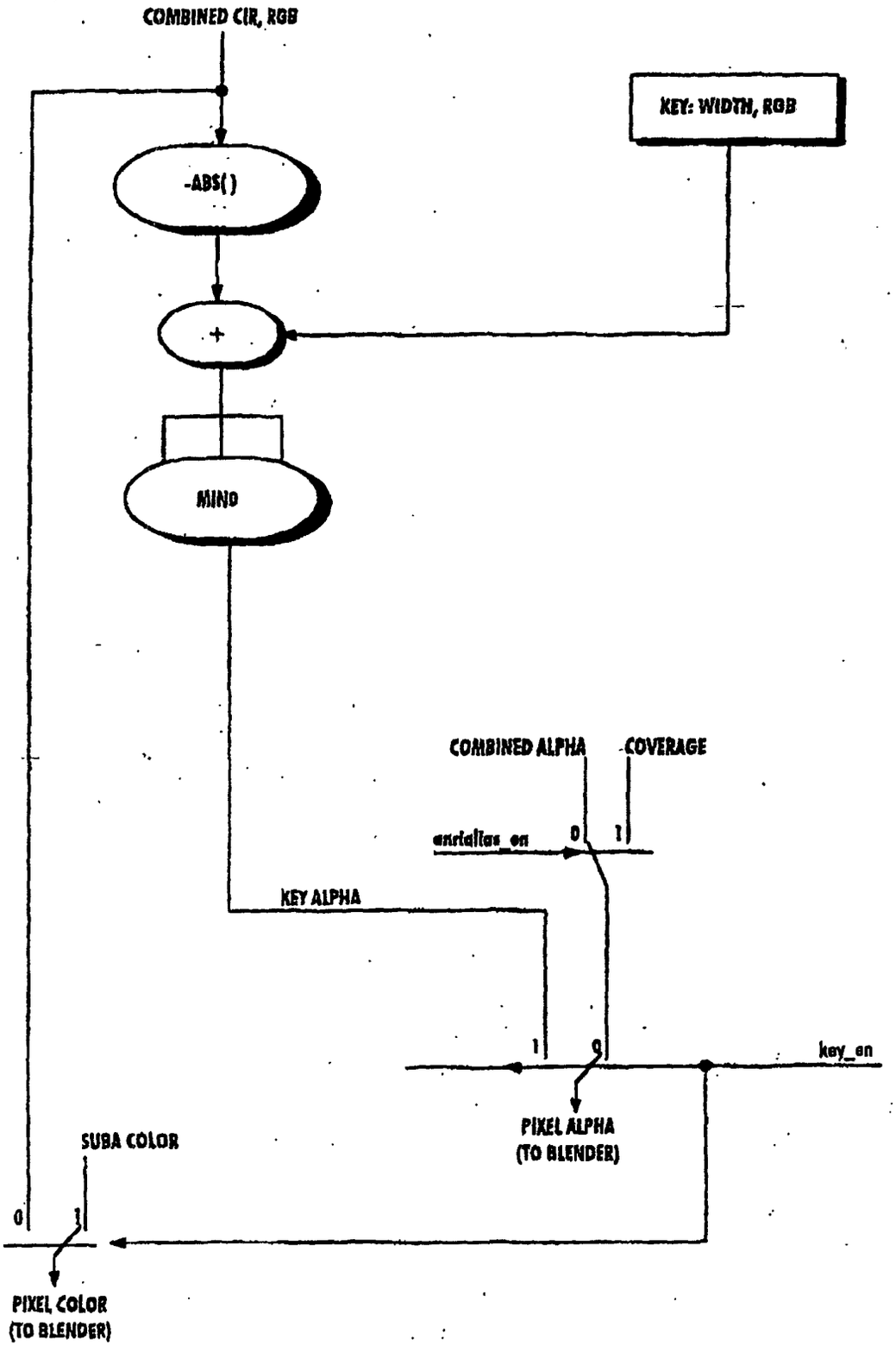


Fig. 30 BLENDER OPERATION

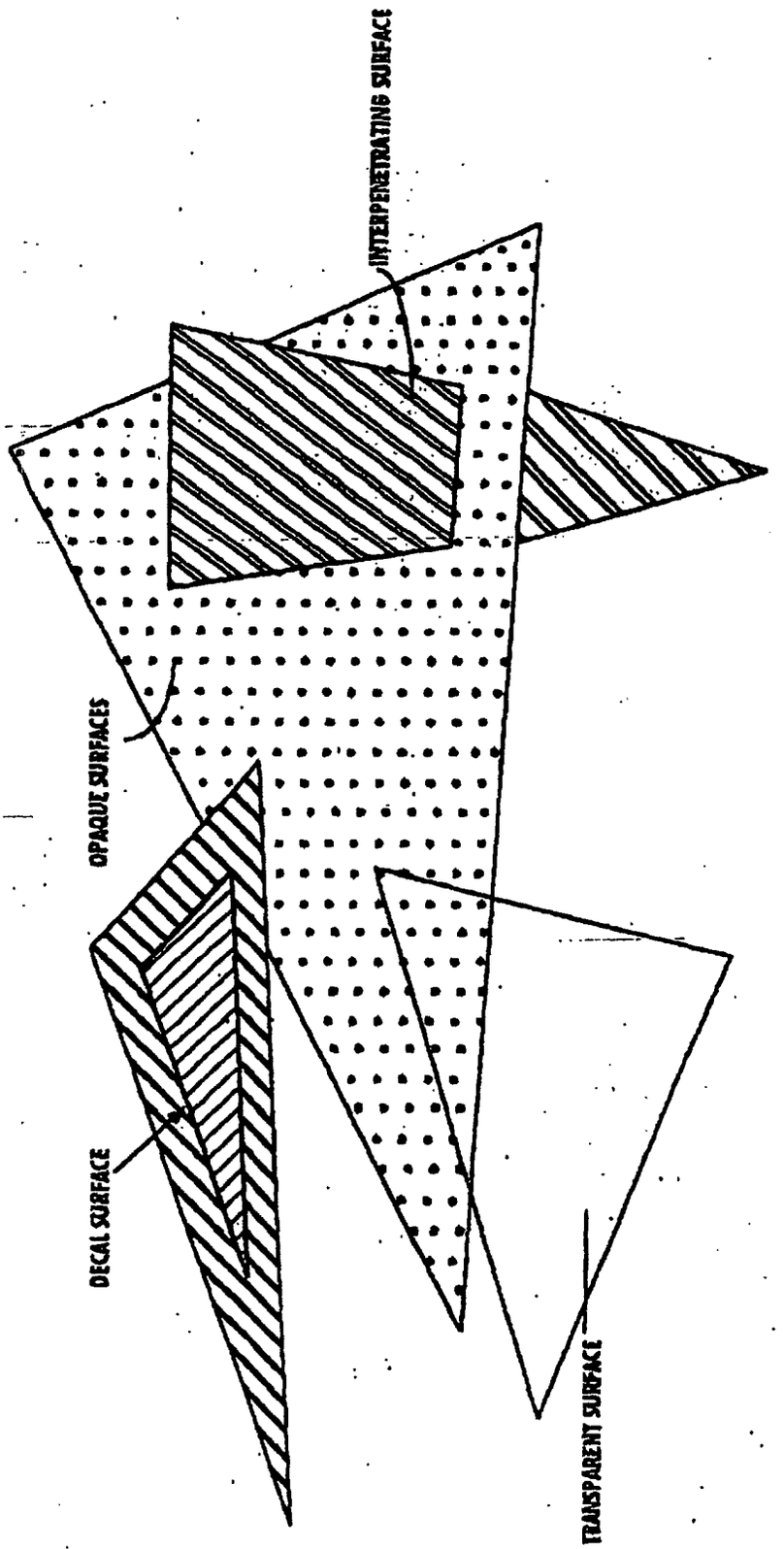


Fig. 31 BLENDER

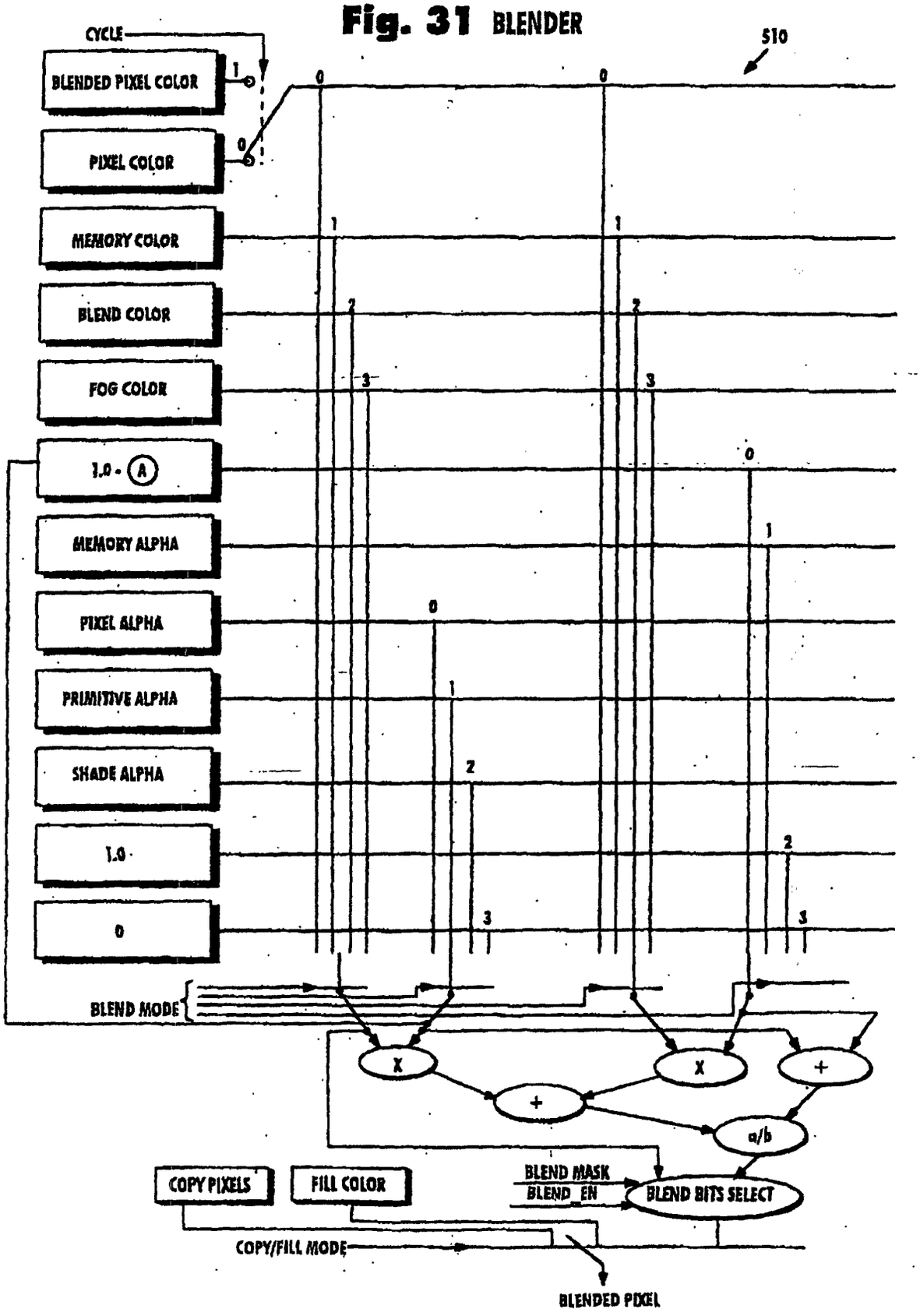


Fig. 32 COLOR PIXEL FORMAT

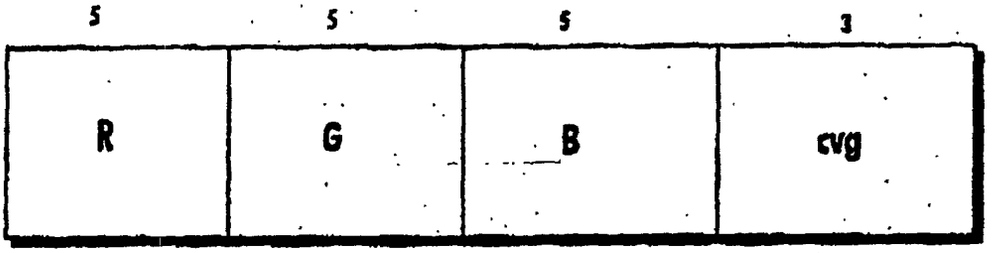


Fig. 33 DEPTH PIXEL FORMAT

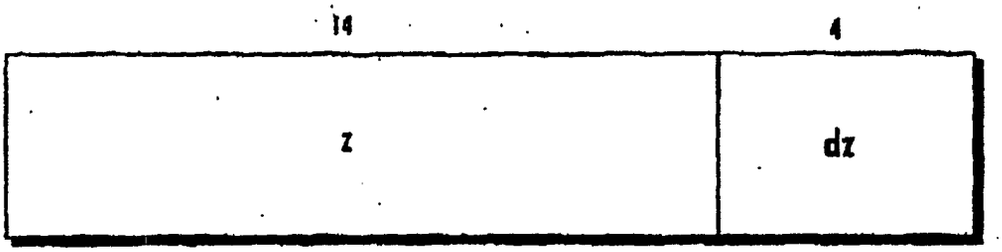
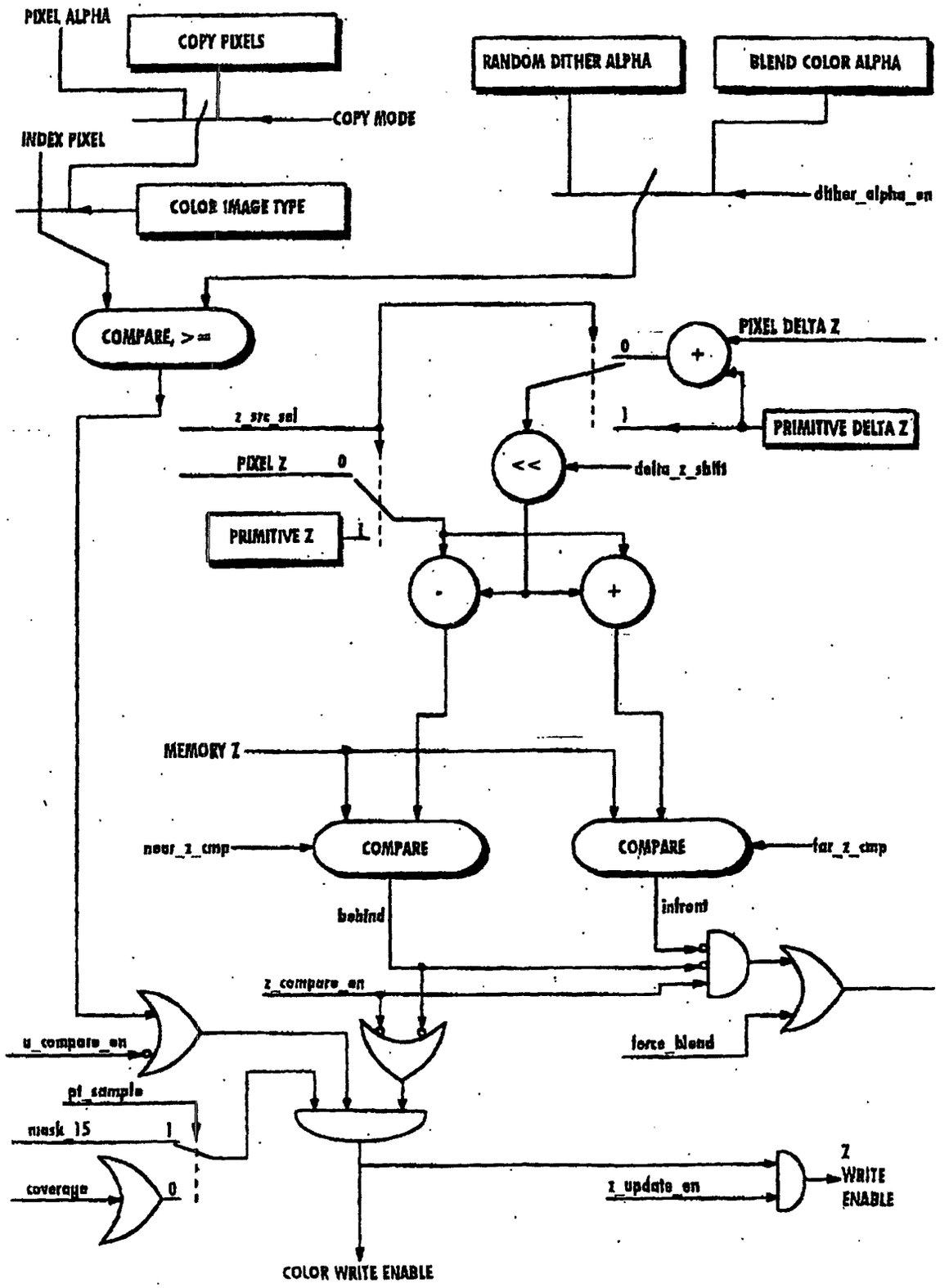


Fig. 33A WRITE ENABLE GENERATION



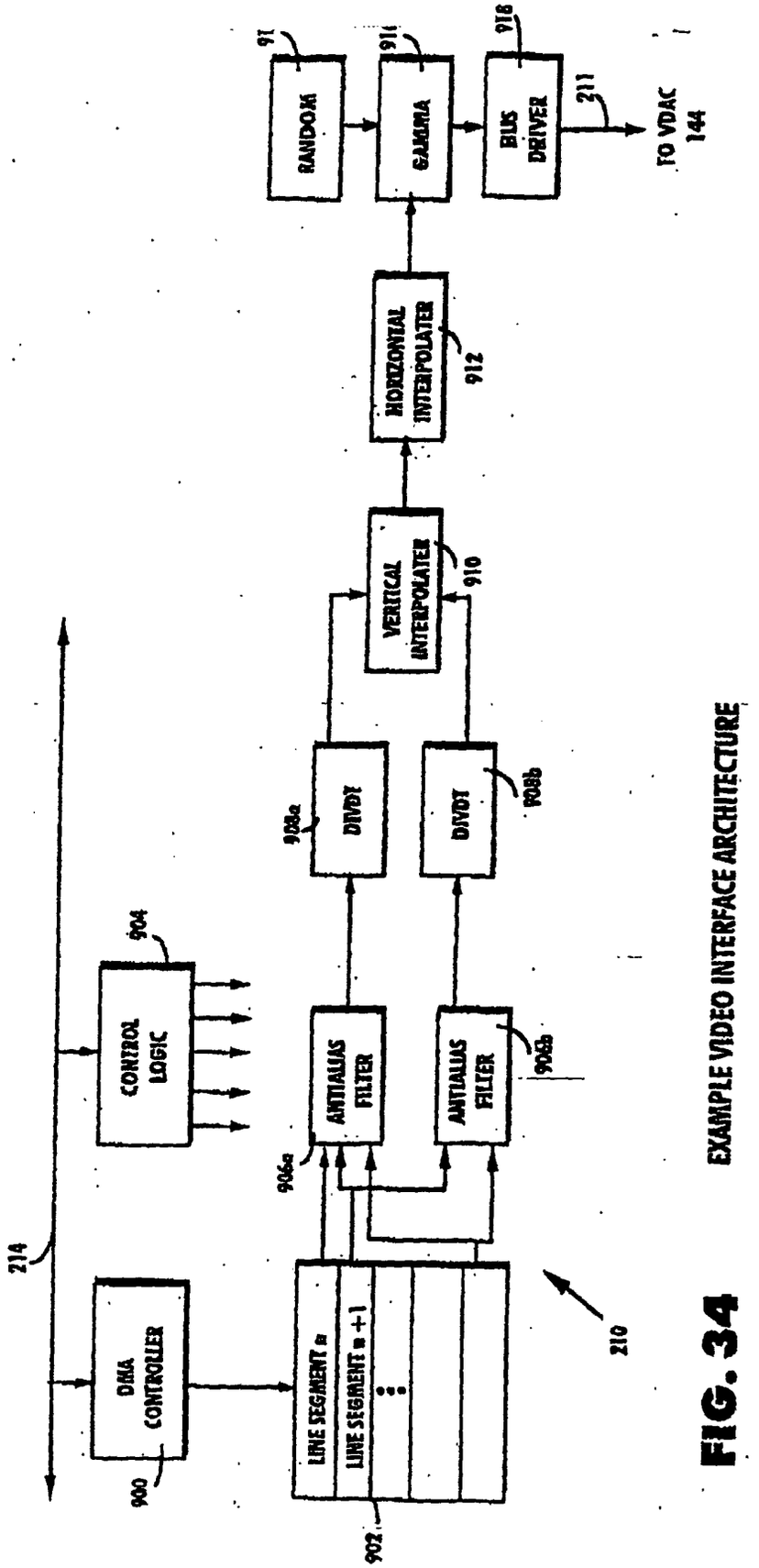


FIG. 34 EXAMPLE VIDEO INTERFACE ARCHITECTURE

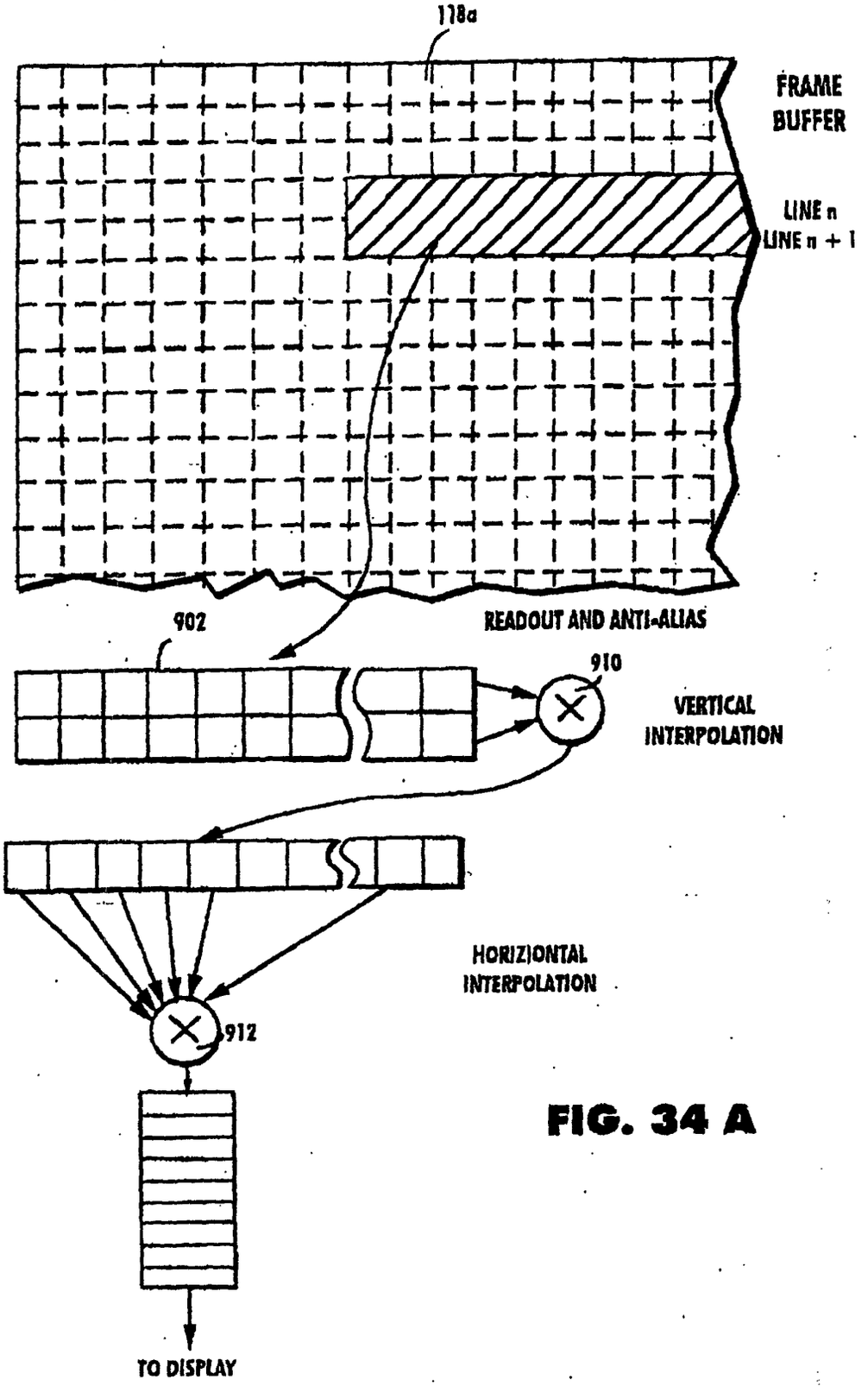


FIG. 34 A

EXAMPLE VIDEO INTERFACE OPERATION

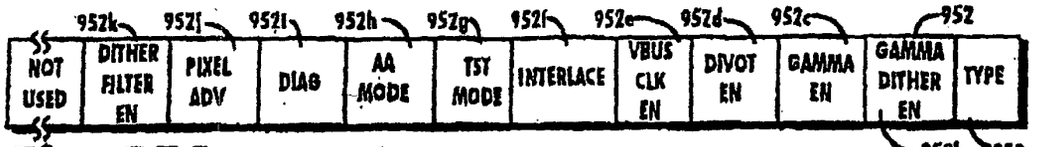


Fig. 35A VIDEO INTERFACE CONTROL REGISTER (R/W)

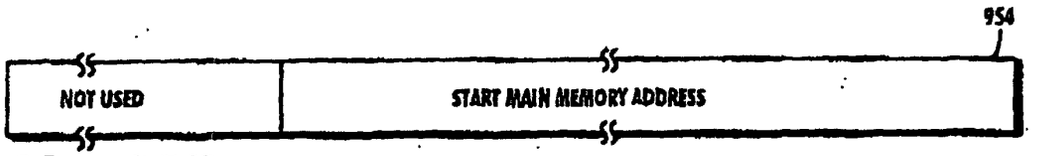


Fig. 35B VIDEO INTERFACE ORIGIN REGISTER (R/W)

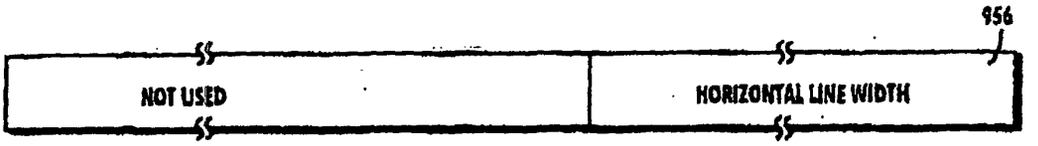


Fig. 35C VIDEO INTERFACE LINE WIDTH REGISTER (R/W)

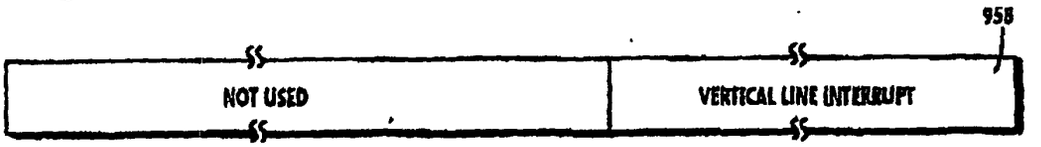


Fig. 35D VIDEO INTERFACE VERTICAL INTERRUPT REGISTER (R/W)

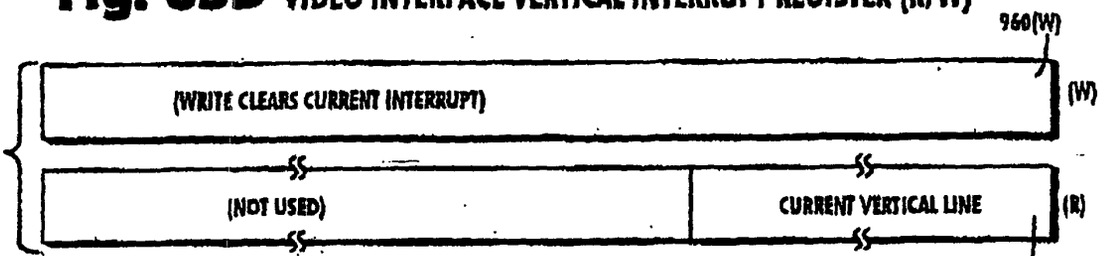


Fig. 35E VIDEO INTERFACE CURRENT LINE REGISTER

Fig. 35F VIDEO INTERFACE VIDEO TIMING REGISTER (R/W)

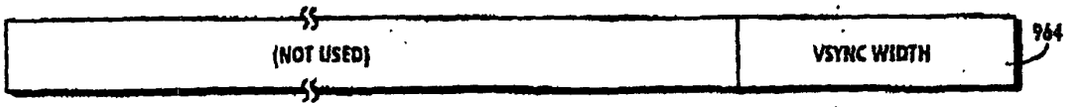


Fig. 35G VIDEO INTERFACE VSYNC REGISTER (R/W)

Fig. 35H VIDEO INTERFACE HSYNC REGISTER (R/W)

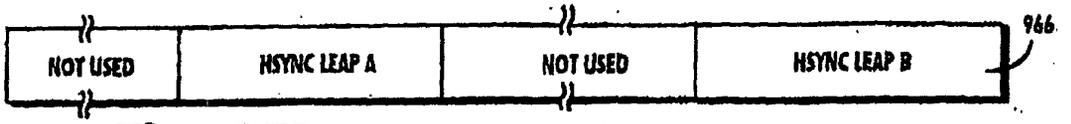


Fig. 35I VIDEO INTERFACE HSYNC LEAP REGISTER (R/W)

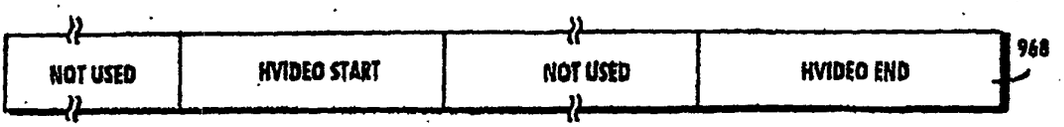


Fig. 35J VIDEO INTERFACE H VIDEO REGISTER (R/W)

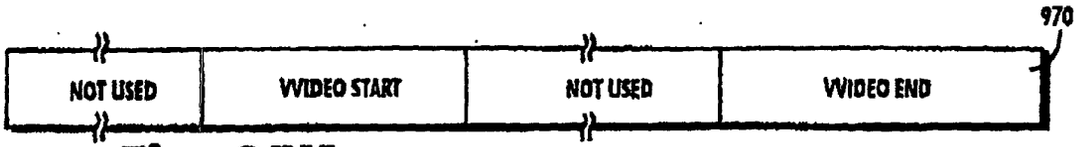


Fig. 35K VIDEO INTERFACE V VIDEO REGISTER (R/W)

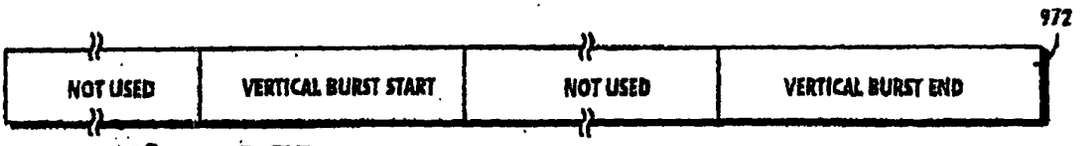


Fig. 35L VIDEO INTERFACE V BURST REGISTER (R/W)

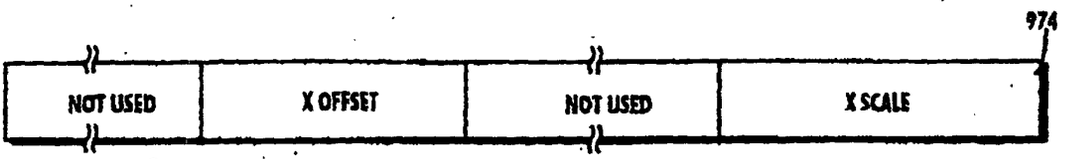


Fig. 35M VIDEO INTERFACE X SCALE REGISTER (R/W)

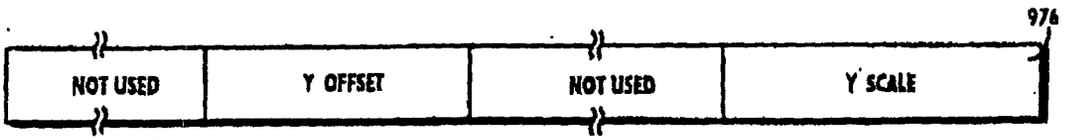


Fig. 35N VIDEO INTERFACE Y SCALE REGISTER (R/W)

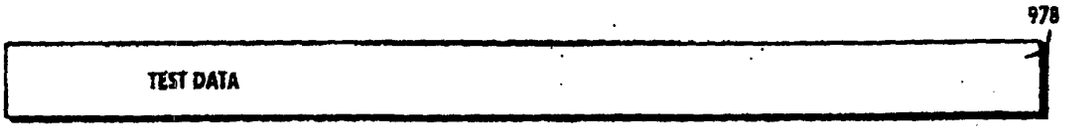


Fig. 35O VIDEO INTERFACE TEST DATA REGISTER (R/W)

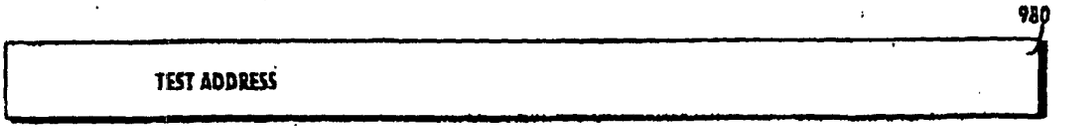


Fig. 35P VIDEO INTERFACE TEST ADDRESS REGISTER (R/W)

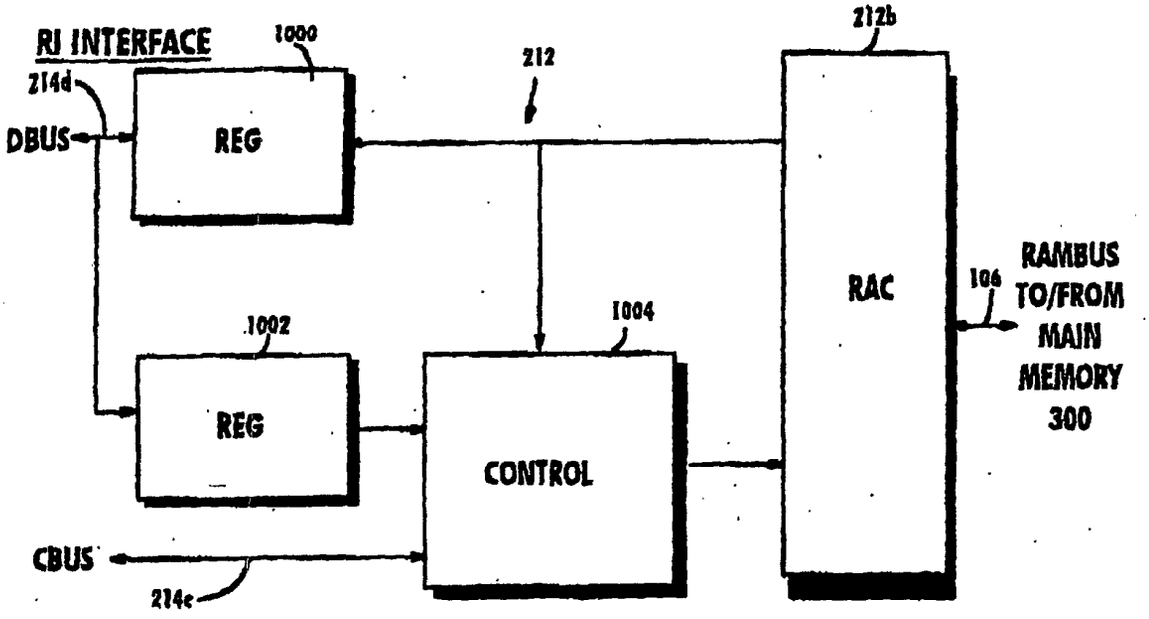


Fig. 36 EXAMPLE MEMORY INTERFACE

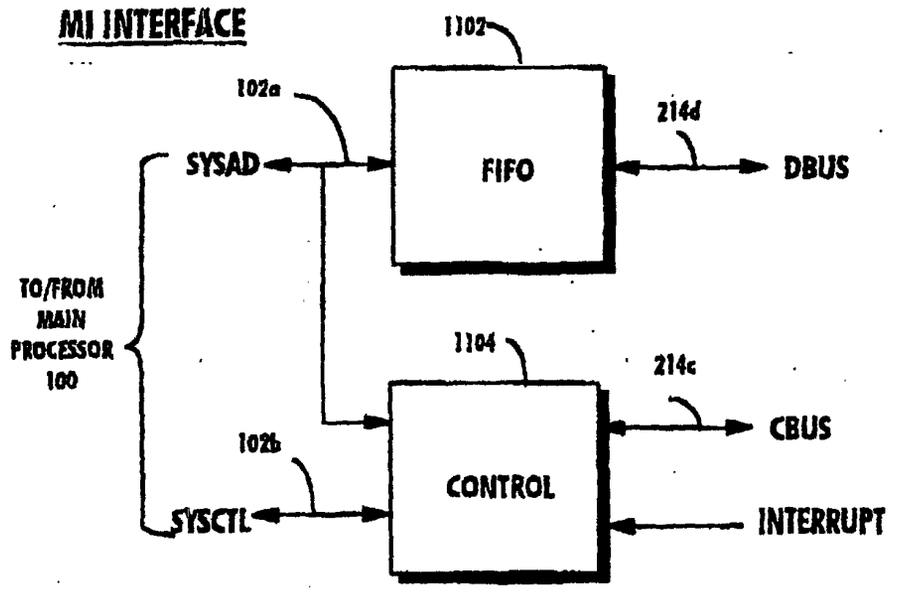


Fig. 38 EXAMPLE CPU INTERFACE

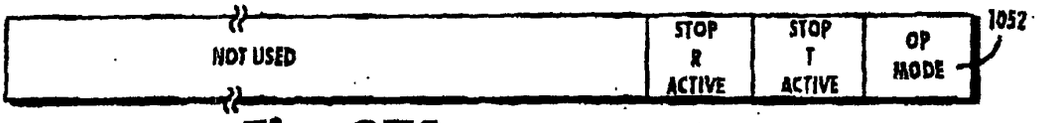


Fig. 37A MI MODE REGISTER(R/W)

Fig. 37B MI CONFIG REGISTER (R/W)

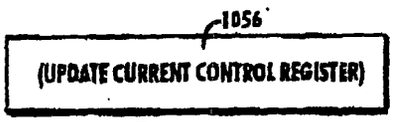


Fig. 37C CURRENT LOAD REG(W)

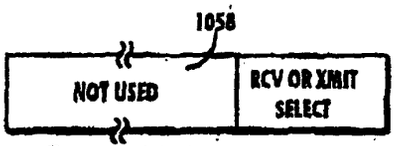


Fig. 37D SELECT REG

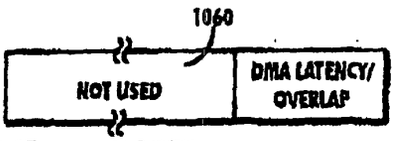


Fig. 37E LATENCY (R/W)

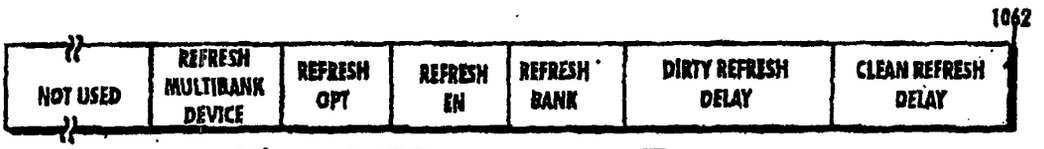


Fig. 37F REFRESH REGISTER (R/W)

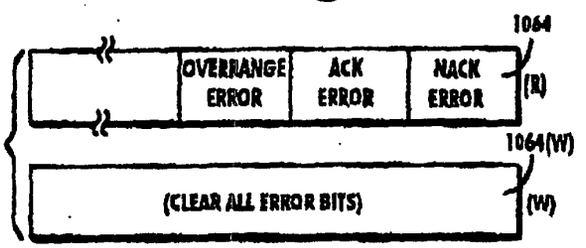


Fig. 37G ERROR REGISTER

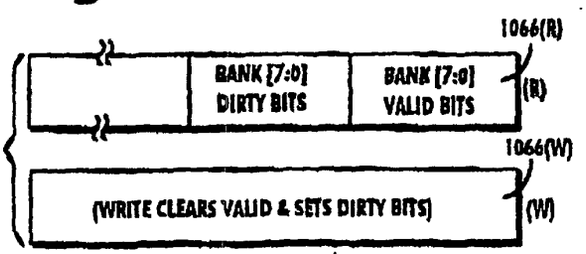


Fig. 37H BANK STATUS REG

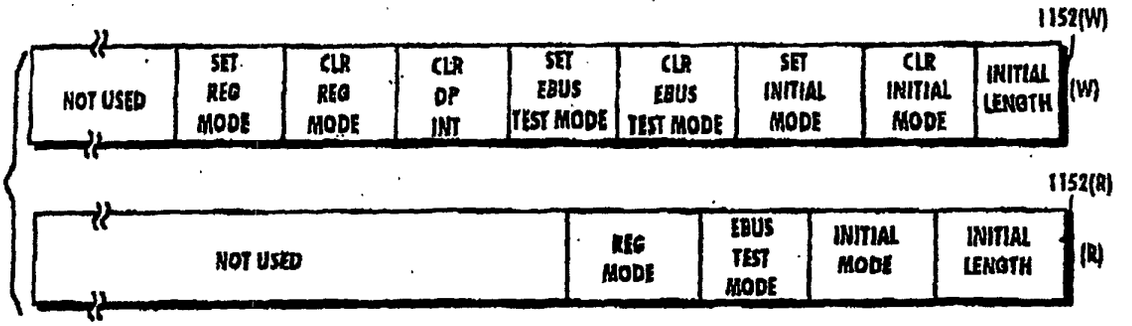


Fig. 39A CPUI STATUS/CONTROL REGISTER (R/W)

Fig. 39B CPUI VERSION REGISTER (R)

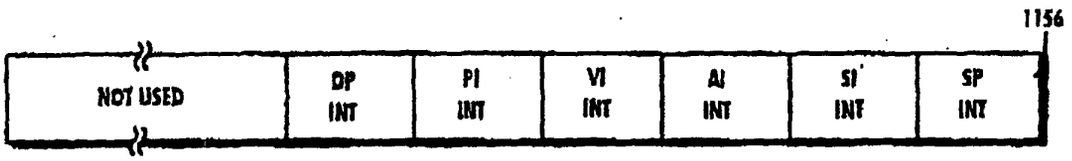


Fig. 39C INTERRUPT REGISTER (R)

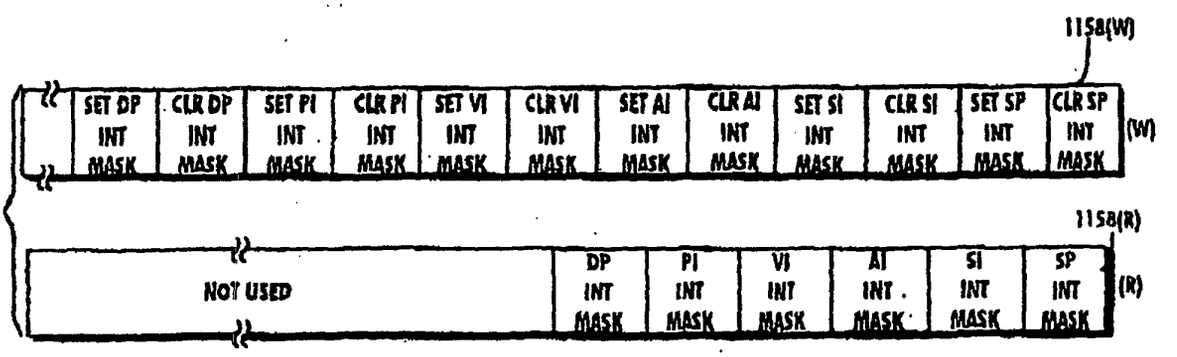
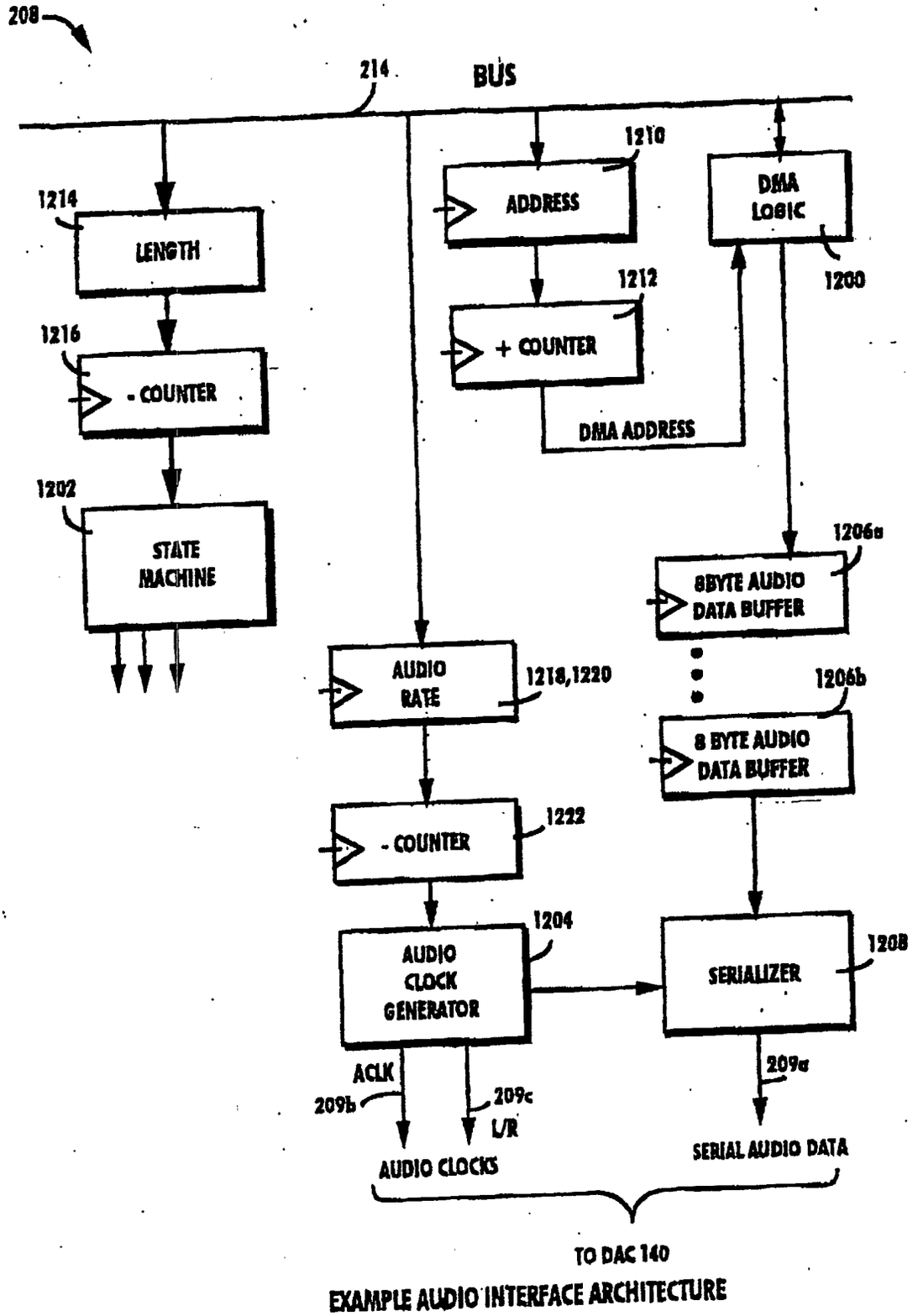


Fig. 39D INT MASK REGISTER



EXAMPLE AUDIO INTERFACE ARCHITECTURE

Fig. 40

FIG. 41A

ASI DRAM ADDRESS REGISTER (W)

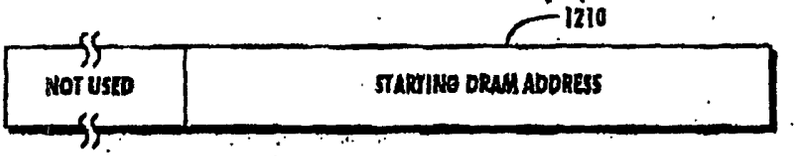


FIG. 41B

AI TRANSFER LENGTH REGISTER (R/W)

FIG. 41C

AI DMA ENABLE REGISTER (W)

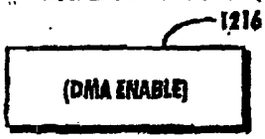


FIG. 41D

AI STATUS REGISTER (R)

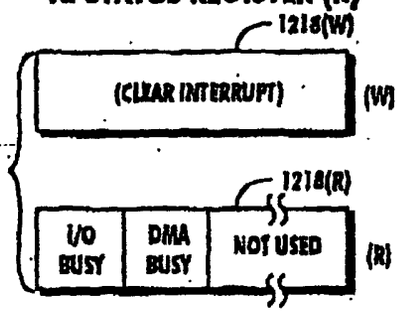


FIG. 41E

AI AUDIO DAC PERIOD REGISTER (W)

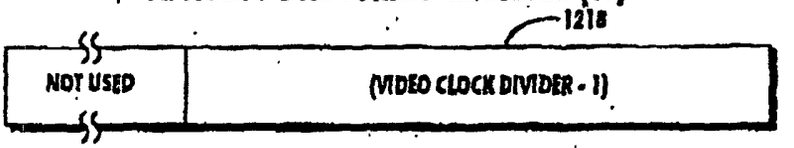
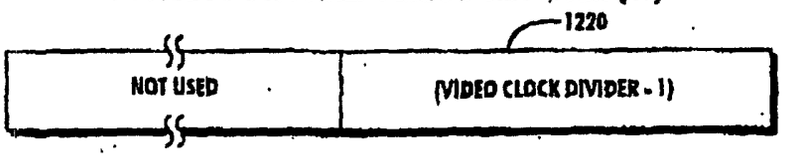


FIG. 41F

AI AUDIO DAC HALF PERIOD REGISTER (W)



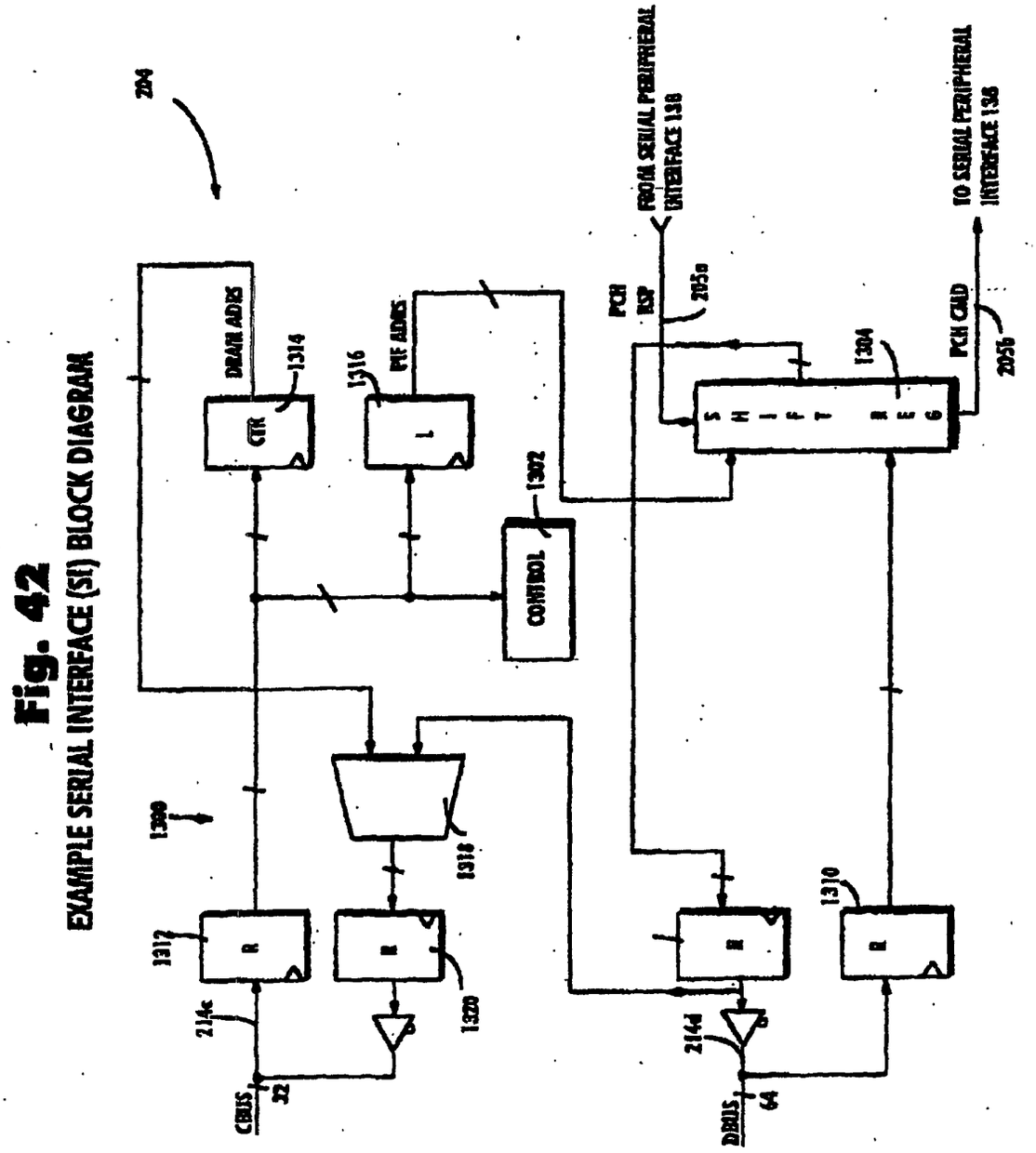


Fig. 42
EXAMPLE SERIAL INTERFACE (SI) BLOCK DIAGRAM

FIG. 43A

SI DRAM ADDRESS REGISTER (R/W)

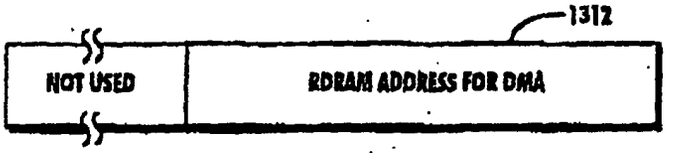


FIG. 43B

SI/PIF DMA WRITE REQUEST REGISTER (W)

FIG. 43C

SI/PIF DMA READ REQUEST REGISTER (W)

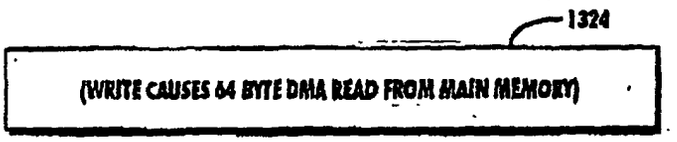
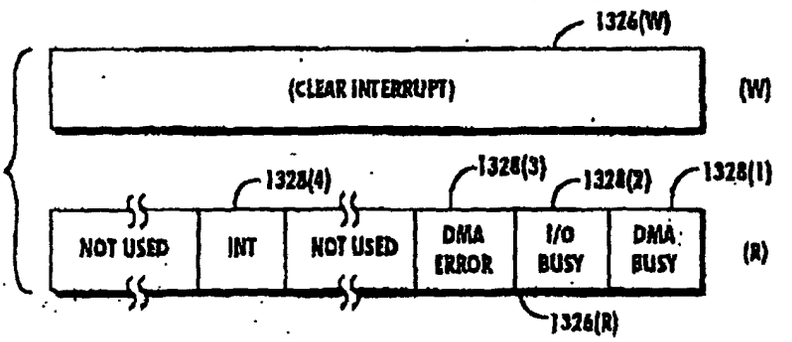


FIG. 43D

SI STATUS CONTROL REGISTER (R/W)



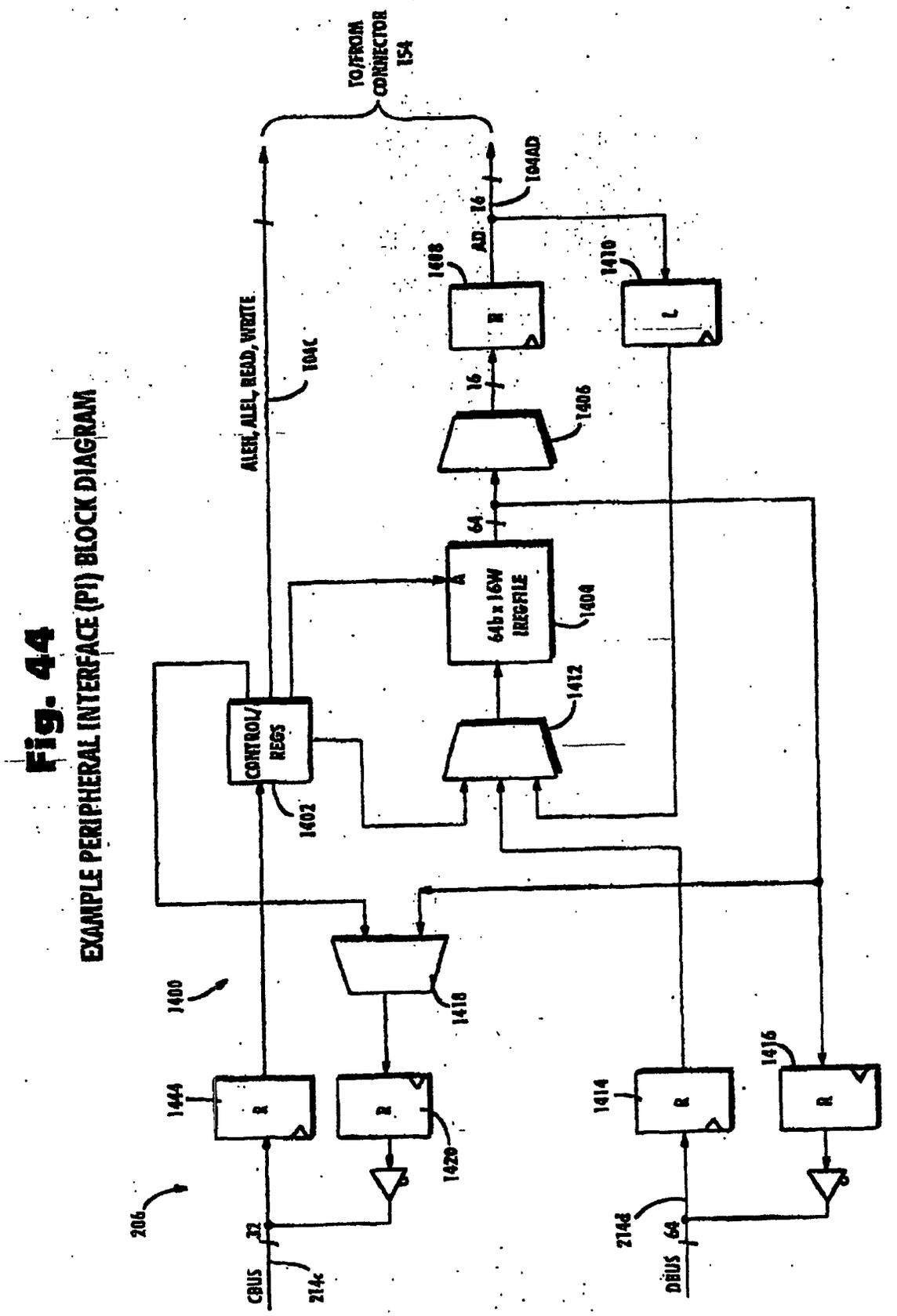


Fig. 44
EXAMPLE PERIPHERAL INTERFACE (PI) BLOCK DIAGRAM

Fig. 45A PI DRAM ADDRESS (R/W)

Fig. 45B PI PERIPHERAL BUS ADDRESS (R/W)

Fig. 45C PI DMA READ LENGTH (R/W)

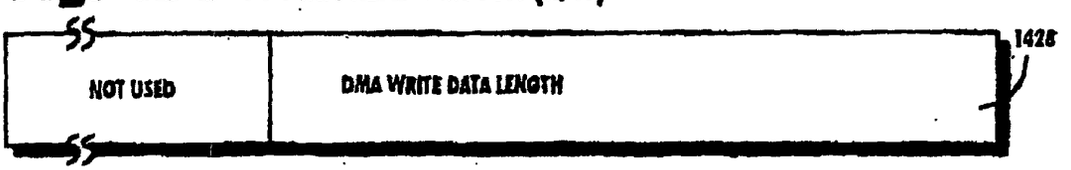


Fig. 45D PI WRITE LENGTH (R/W)

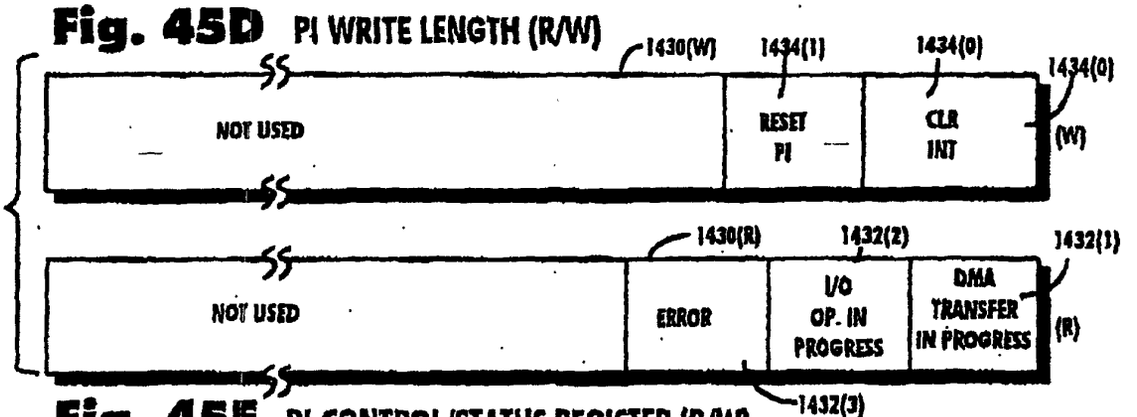


Fig. 45E PI CONTROL/STATUS REGISTER (R/W)

Fig. 45F PI LATENCY REG (R/W)

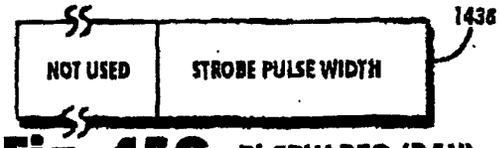


Fig. 45G PI SPW REG (R/W)

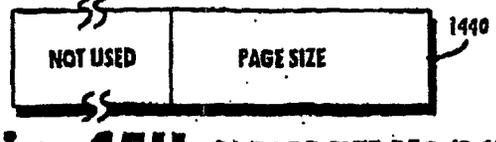


Fig. 45H PI PAGE SIZE REG (R/W)

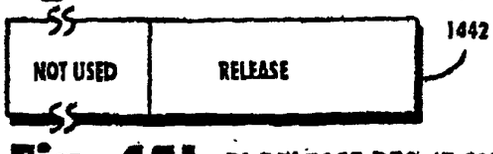


Fig. 45I PI RELEASE REG (R/W)