3D Programmer's Guide

3. 3D PROGRAMMER'S GUIDE

This chapter presents the information necessary to program 3D functions on the CL-GD5464 (the CL-GD5462 does not incorporate a 3D engine). The 3D rendering engine can draw polygons with Gouraud shading, alpha blending, Z-buffering, and texture mapping. Autonomous execution from a display list and a rich instruction set combine to minimize the load on the host while rendering 3D scenes.

The registers used by the 3D engine are covered in the *Laguna VisualMedia™ Accelerators Family — CL-GD546X Volume I (Hardware Reference Manual, Second Edition, September 1996)*. These registers are all accessible in memory-mapped I/O space, with or without byte swapping. When the engine is running in Display List mode, these registers are loaded from the display list by the engine.

3.1 Architectural Overview

This section provides a brief overview of the CL-GD5464 graphics system from a programmer's point of view. It begins with an overall system block diagram that covers the entire graphics system. This is followed with a block diagram showing a conceptual view of the CL-GD5464.

3.1.1 System Block Diagrams

Figure 3-1 shows a graphics subsystem based on the CL-GD5464. The blocks shown as solid lines are in the CL-GD5464 device. The blocks shown as dotted lines are outside the CL-GD5464 device.

The graphics subsystem provides a visible rectangular display mapped onto a rectangular memory space. This memory space is the frame buffer. The frame buffer is implemented using RDRAMs.

On the input side of the frame buffer is a standard SVGA controller, a 2D/3D engine, a direct frame-buffer interface unit, and a V-Port™ video bus interface. On the output side of the frame buffer is the RAMDAC, which in turn drives the monitor.

The CRTC controller generates the display timing, providing horizontal and vertical synchronization terms for the monitor and display refresh requests to the frame buffer. The CRTC controller also provides a blanking term to the RAMDAC.

The RAMDAC maps memory contents to RGB color values. The frame buffer contains a description of each pixel on the screen. It can also contain an off-screen color buffer, Z buffer, and texture maps. The format of the pixel and texel descriptions in the frame buffer can be palletized, RGB, or YUV. The frame buffer can contain pixels in more than one format.

The various blocks within the CL-GD5464 can be programmed by the CPU via the PCI bus interface. Alternately, the CL-GD5464 can become a PCI bus master under the control of the prefetch unit to fetch instructions and parameters from host system memory. The HostXY unit can also initiate bus master operation to fetch texture maps or render a color buffer and Z buffer to host memory.

Also available on the PCI bus are a set of standard VESA VBE v2.0 BIOS software routines for implementing the VESA SVGA standard, and for initializing and testing the system.

Finally, the CL-GD5464 has a set of PCI Configuration registers.

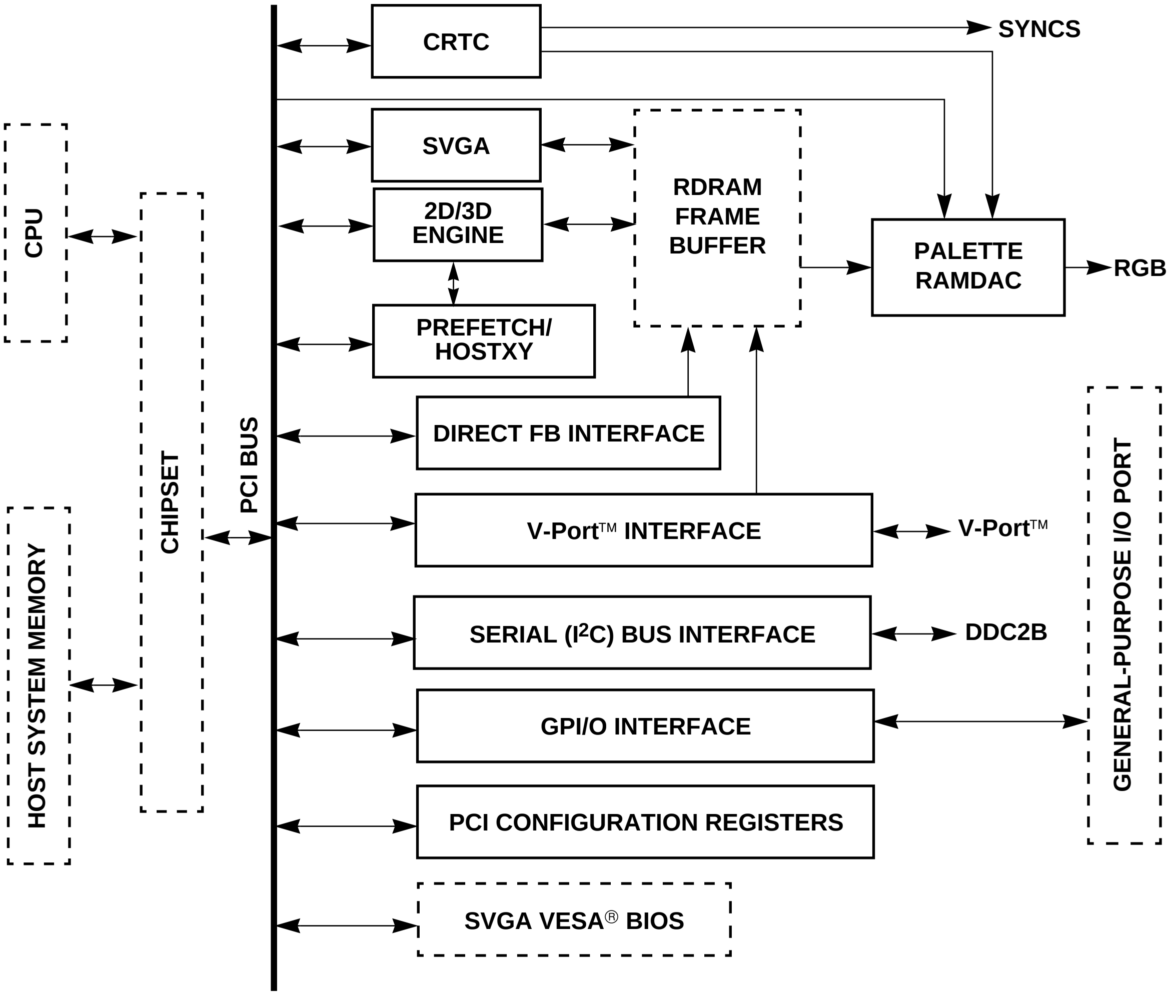

Figure 3-1. Graphics System Based on the CL-GD5464

3.1.2 Internal Architecture

Figure 3-2 is a conceptual diagram of the internal architecture of the CL-GD5464. It is implemented around two internal buses. The host bus is shown as HIFBUS and the memory bus is shown as RIFBUS. The HIFBUS is connected to the external PCI bus through the host interface module. The RIFBUS is connected to the Rambus channel through a RIF (Rambus interface) and RAC (Rambus access channel). The host interface module synchronizes the external bus clock to the internal memory clock. Both the HIFBUS and RIFBUS are synchronous to the internal memory clock (nominally 62.5 MHz).

The functional modules of the CL-GD5464 are connected to one or both of these buses. The host interface and 2D/3D modules are described in the following sections. Descriptions of the remaining modules can be found in the Chapter 2, "2D Programmer's Guide".

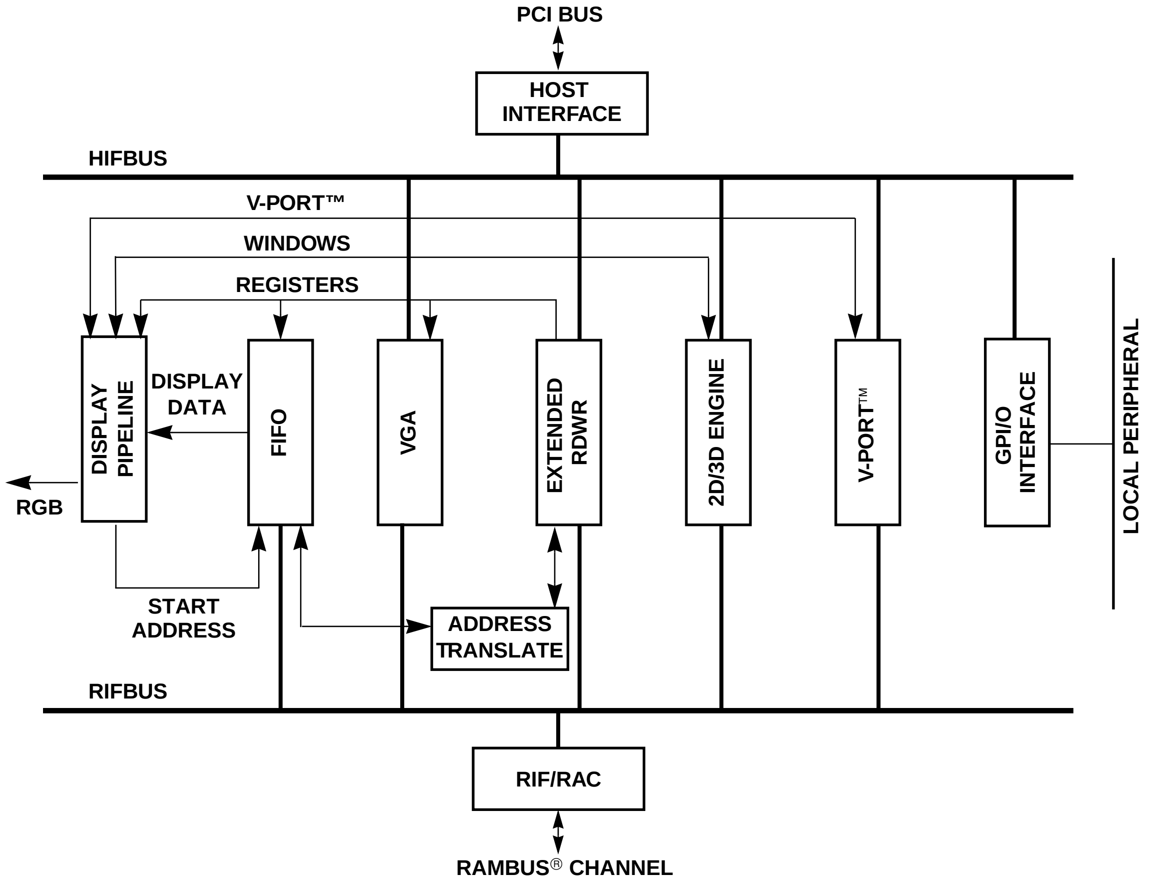

Figure 3-2. CL-GD5464 Internal Architecture

3.1.2.1 Host Interface

The host interface provides a PCI v2.1 compliant target and bus master interface, functional units to permit the CL-GD5464 to behave as a graphics processor with its own instruction set, and an interface to the internal HIF bus.

Figure 3-3 presents the functional units of the host interface. The command queue, read queue, bi-endian swap logic (for writes and reads), address decoder, and PCI control (target) units provide PCI-compliant target transfers of data to or from the CL-GD5464. The prefetch unit and the PCI control (master) unit use PCI bus master cycles to fetch display list instructions and parameters from host system memory. Similarly, the HostXY and PCI control (master) units allow reads of textures maps stored in host memory and rendering (reads and writes) of a color buffer and Z buffer to host memory.

The 8 × 43 command queue allows the CL-GD5464, when acting as a PCI target, to release the host as soon as the transaction parameters have been recorded. This enables the host and media accelerator to operate with a high degree of parallelism.

The host address bus (specifically the address phase of the multiplexed AD bus) enters the address decoder where the CL-GD5464 determines if it is the target of the transaction about to occur. If it is the target of the transaction, the appropriate acknowledge lines are activated by the bus control block and the address is placed in the command queue along with a tag value that indicates the transaction type.

Entries are removed from the command queue and passed on to the appropriate internal block for execution. If the command queue is full, the bus control unit inserts wait cycles until one or more free entries are available.

Read transactions must be executed by the CL-GD546X before the host can be released (since the data must be made available to the host). Generally, this requires a number of wait states. For BIOS reads, up to 4 bytes are assembled into the read queue before the data is placed on the data bus and the host is released.

The prefetch unit is responsible for the fetch and pre-decode of display list instructions from host memory. Rendering instructions are forwarded to the 2D/3D engine by the command queue. Control instructions are executed immediately by the prefetch unit. Internal interrupt or wait events such as a display buffer switch or a vsync are handled by this unit to achieve maximum animation performance.

The HostXY unit translates texture or pixel transactions from the 2D/3D Engine into PCI bus master transactions to the host memory and checks that the addresses are in a valid range. Pixel or Z data writes are queued in the read queue for optimal performance.

Both the prefetch and HostXY units contain virtual-to-physical address translation logic that reads a translation table from host memory.

Bi-endian swaps (dword or word) can occur (as needed) on data passing in either direction through the host interface.

The general-purpose I/O port is closely integrated with the host interface. If the CL-GD5464 is configured for general-purpose I/O port, accesses to a specific range of memory-mapped I/O offsets are converted into accesses to the local peripheral.

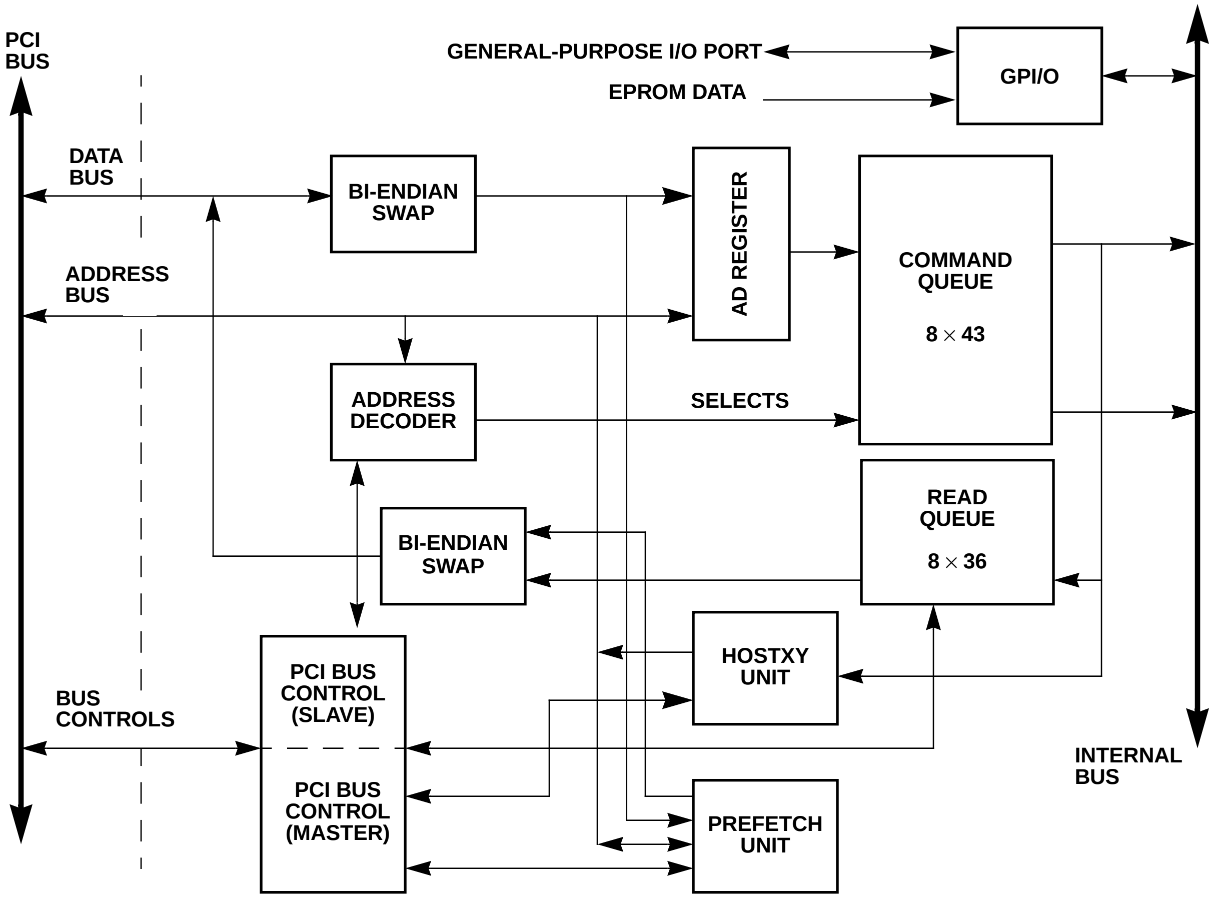

Figure 3-3. Host Interface Block Diagram

3.1.2.2 2D/3D Engine

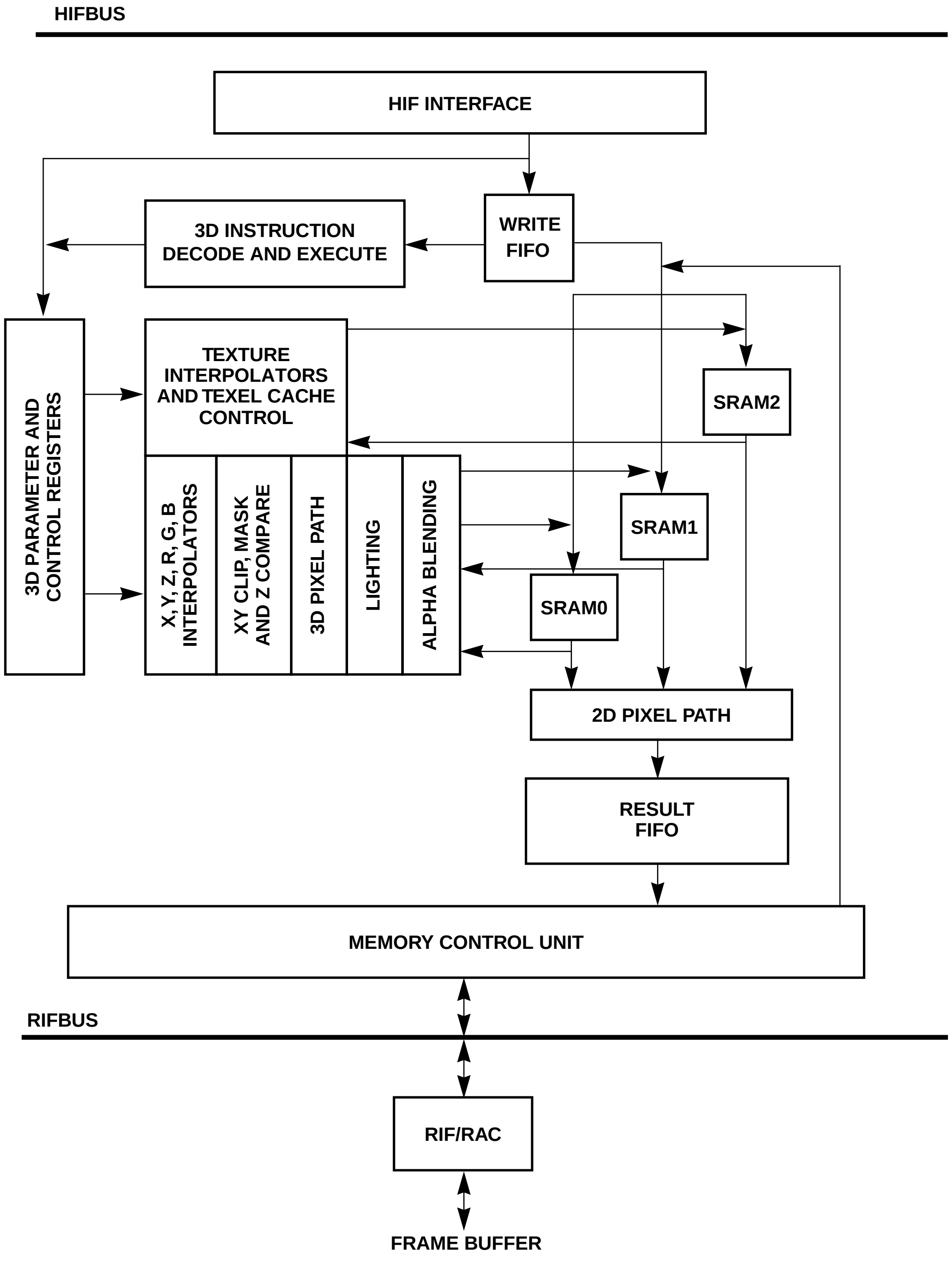
HIFBUS
HIF INTERFACE
WRITE FIFO
3D INSTRUCTION DECODE AND EXECUTE
3D PARAMETER AND CONTROL REGISTERS
TEXTURE INTERPOLATORS AND TEXEL CACHE CONTROL
X, Y, Z, R, G, B INTERPOLATORS
XY CLIP, MASK AND Z COMPARE
3D PIXEL PATH
LIGHTING
ALPHA BLENDING
SRAM2
SRAM1
SRAM0
2D PIXEL PATH
RESULT FIFO
MEMORY CONTROL UNIT
RIFBUS
RIF/RAC
FRAME BUFFER

Figure 3-4. 2D/3D Graphics Engine Model and Data Flow

3.2 3D Programming Model

There are three mechanisms for programming the 3D engine. These methods are described in the section summarized in Table 3-1.

Table 3-1. 3D Programming Modes

Programming Mode	Section
Direct Programming	Section 3.2.1
Coprocessor Indirect Programming	Section 3.2.2
Display List Programming	Section 3.2.3

The actual instructions and instruction formats are similar regardless of the mode being used. Bits 5:0 of drawing instructions are set to '0' for Direct programming (since the registers have already been loaded). Also, the Branch and Idle instructions are used only to enter and exit Display List mode.

3.2.1 Direct Programming

In Direct Programming mode, registers are loaded, typically by writing directly to them in the register address space beginning at 0x4000. The 3D engine must be idle while the registers are being loaded. When the appropriate registers have been loaded, the engine is started by writing a drawing instruction to the OPCODE_3D register at 0x40FC. The engine executes a single instruction and then stops. This method is also known as the Coprocessor Direct method.

Direct programming is often used to see the effect of a single instruction without the possibility of the interference of additional instructions. This would allow one to 'single-step' through a display list.

Direct programming is the slowest method of initializing the device state and issuing drawing commands. It should be used for initialization and situations where time is not a constraint. No PCI burst transfers occur when direct programming is being used.

Flow control instructions, data movement instructions, and the like are generally not available in the Direct Programming mode.

3.2.2 Coprocessor Indirect Programming

In the Coprocessor Indirect Programming mode, instructions and data can be transferred across the PCI bus by writing to offsets 0x4800 through 0x4BFC. To execute a 3D Draw instruction in Coprocessor Indirect mode, execute a Write_Register instruction to Host_3D_Data_Port at MMIO offset 0x4800. The data transferred during the next group of A 32-bit writes are written beginning at register address M, where A and M are values in the Write_Register instruction.

This mode is intended as a test mode and is not recommended for normal operation.

3.2.3 Display List Programming

This is the preferred mode for programming the CL-GD5464. Display list allows the most concurrence between the host and graphics processor. Display list also allows PCI bursting and bus mastering. The CL-GD5464 is optimized for operation in this mode.

In Display List Programming (or Instruction Fetch) mode, the CL-GD5464 fetches instructions and register values from system memory. The instruction list can include load instructions that set up control registers for drawing as well as the drawing instructions themselves. A complete drawing instruction takes the form of the draw opcode followed by a sequence of parameters that define the region to be drawn, its color and texture, and other characteristics.

When all the register values required for an instruction have been fetched and loaded, the instruction itself is executed (some instructions do not require any register values). The CL-GD5464 is put into instruction fetch mode by the execution of a BRANCH instruction (the instruction is written to the PF_INST_3D register at 0x4480). Usually the CL-GD5464 exits Display List mode by the execution of an IDLE instruction. The PF_CTL_3D register at 0x4404 has bits that control pausing and instruction fetching. The PF_STATUS_3D register at 0x4424 returns the status of the 3D engine.

The host must build the display list (instruction/register values) in system memory prior to starting the 3D engine. Once started, the engine proceeds autonomously until it encounters an IDLE instruction. The INT instruction can be used to report intermediate progress. The INT instruction causes a pause in display list execution, requiring the application to restart execution. This is done by writing a RET instruction to the PF_INST_3D register.

3.2.4 Host Memory-Based Formats

TBD

3.3 3D Rendering Overview

The following sections discuss 3D rendering as it is done on the CL-GD5464. First, the DDA (digital differential analyzer) is reviewed. Rendering of a flat polygon is covered to introduce the method used by the CL-GD5464. Then, shaded coloring (Gouraud shading) of the polygon is explained. Methods of clipping and masking are covered. Finally, lighting, alpha blending, and texture mapping are covered.

Figure 3-5 is a simplified block diagram of the 3D engine. The polygon interpolator determines the pixels that fall within the polygon and includes the color (RGB) interpolators. This runs in parallel with the texture interpolator and its cache. X-Y clipping, the depth comparison logic (hidden surface removal), and destination masking all are used to determine whether pixels are actually written. The pixels that are actually written go through the lighting and saturate stages, through the alpha blending stage to determine their final color, and are then written.

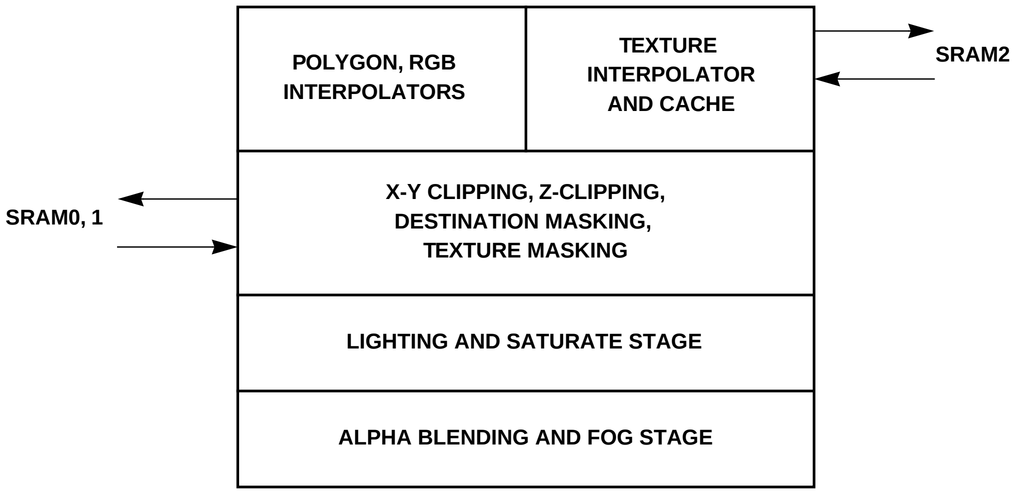
POLYGON, RGB INTERPOLATORS
TEXTURE INTERPOLATOR AND CACHE
SRAM2
SRAM0, 1
X-Y CLIPPING, Z-CLIPPING, DESTINATION MASKING, TEXTURE MASKING
LIGHTING AND SATURATE STAGE
ALPHA BLENDING AND FOG STAGE

Figure 3-5. Simplified Block Diagram of 3D Engine

3.3.1 Incremental Line-Drawing Algorithm

The straight line is the basis of all objects drawn by the 3D engine. To understand how lines are drawn (or more properly, how pixels along a line are selected), one must first understand the incremental line-drawing algorithm or DDA (digital differential analyzer). All modern line-drawing engines use incremental line-drawing algorithms since they are well understood and well suited to evaluation with adders. Bresenham's Algorithm is an example of a DDA that is especially well suited to integer arithmetic.

An incremental algorithm begins at the starting point and proceeds for some number of iterations, calculating the location of a single pixel for each iteration. For each iteration, an increment or delta is added to each coordinate to calculate the location of the next pixel in the line. The number of iterations is the number of points in the line or the distance to be spanned.

The axis whose span is greater is called the step or major axis. For rendering into a discrete bit map, the increment for the major axis is always set to unity. Exactly one pixel is drawn in each scanline (for a Y-major line) or column (for an X-major line). There is no sense in calculating positions *between* pixels (in the major axis). On the other hand, a dense line requires that pixels be drawn just as close together as possible.

The increment for the other axis (the axis that is not the major axis) is the span for that axis divided by the number of iterations (steps in the major axis). For lines drawn at 45 degrees, this is unity. For any other angle, the increment for the minor axis is a proper fraction.

Figure 3-6 shows a line whose major axis is Y. The Y increment is unity (every pixel along the major axis is plotted). The X increment is some fraction whose value depends on the exact slope of the line. In this example, the actual value is somewhat less than two-fourths (a change of four pixels along the Y axis is accompanied by a change of somewhat less than two in the X axis). Pixels are plotted which most nearly correspond to the theoretical line.

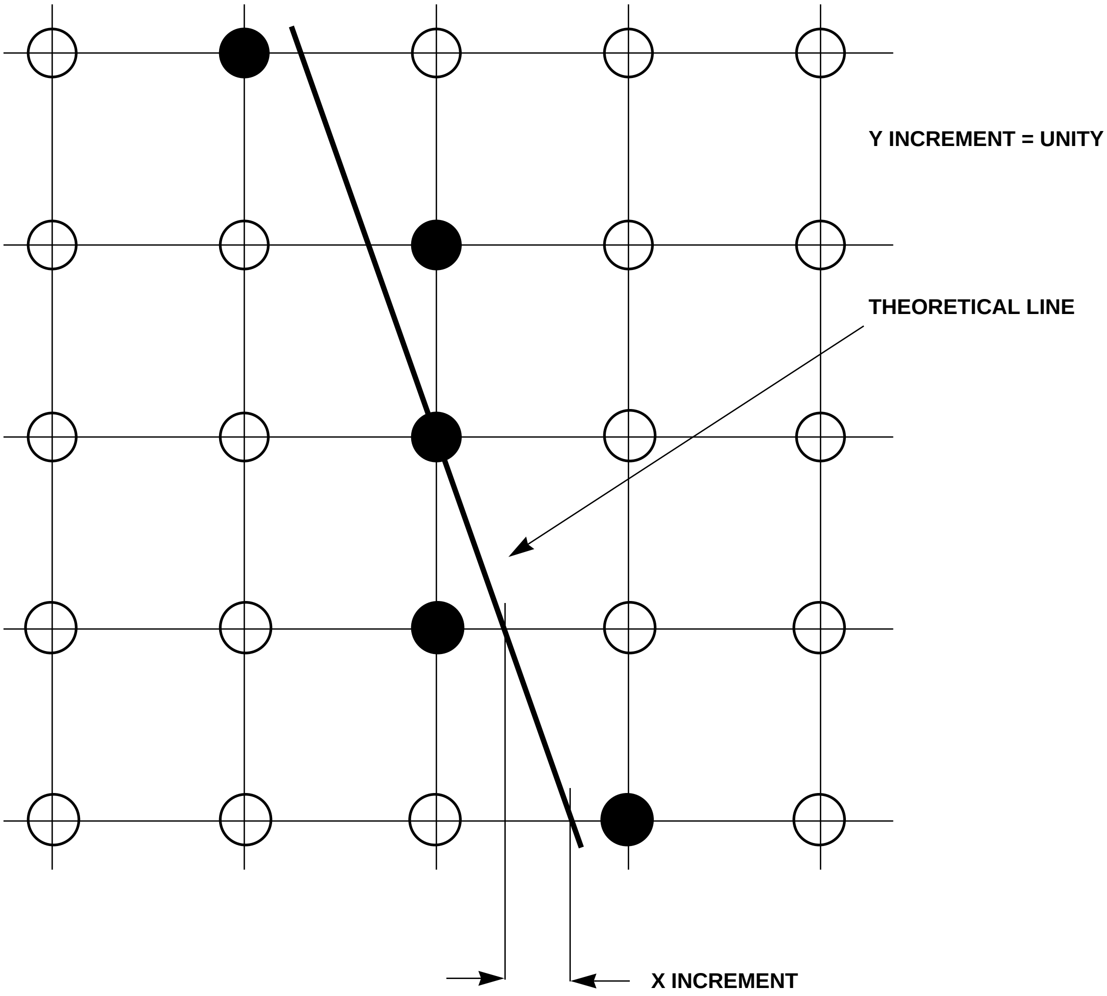

Figure 3-6. Incremental Line-Drawing Algorithm

In addition to calculating the position of a pixel, DDAs are also used to calculate the color, depth, of the pixel, as seen.

See Section 3.3.16 on page 3-30 for information on drawing lines using the CL-GD5464.

3.3.2 Flat (Unshaded) Polygon

The CL-GD5464 can fill polygons that are random triangles of any orientation or quadrangles with at least one flat top or bottom. The CL-GD5464 fills polygons by evaluating two (sometimes more than two) incremental algorithms in parallel. Figure 3-7 illustrates a random triangle. It will be seen that quadrangles with a flat top or bottom constitute an extension.

Set aside trivial triangles and co-linear triangles (they are either points or lines). That is, consider only triangles with three unique vertices that are not all on a straight line.

Any triangle, regardless of its orientation, can be reduced to two triangles with a common horizontal side. In particular, if the three vertices are sorted vertically, the common side is horizontally in line with the center (or opposite) vertex. The side of the triangle opposite this center vertex always spans the entire height of the triangle. This side is called the main slope. By definition, each side of any triangle, including the main slope, is a straight line.

Once a random triangle has been reduced to two triangular areas that are each guaranteed to have a horizontal side, it can be filled in two stages. Refer to Figure 3-7. In the first stage, Area 1 is filled, beginning at the Base vertex and working down, one scanline at a time, to the common horizontal side. In the second stage, Area 2 is filled, beginning at the common horizontal side and working down, one scanline at time, to the bottom vertex.

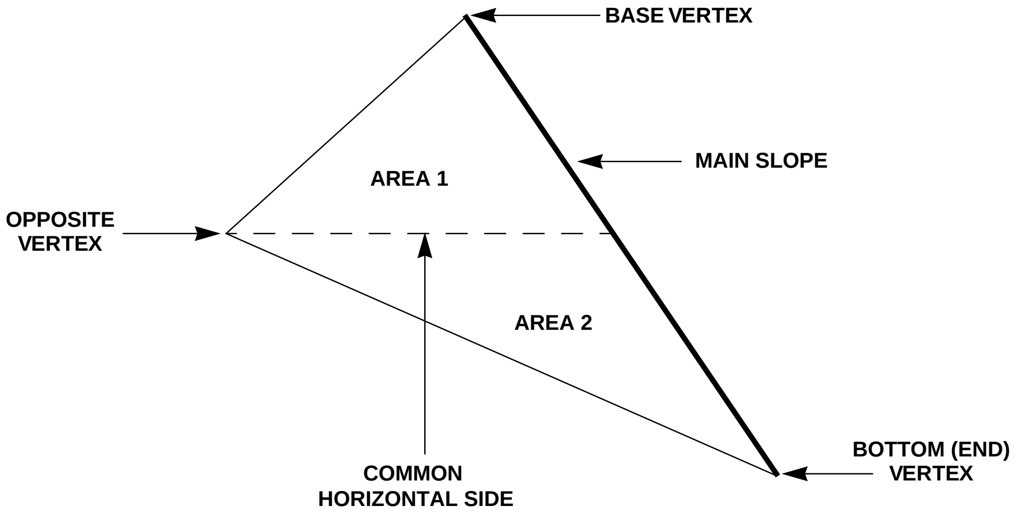

Figure 3-7. Two Triangles with Horizontal Side

For each stage, two parallel DDAs are used. This is illustrated in Figure 3-8. One DDA finds the main slope and the other DDA finds the width. The major axis is always Y, regardless of the orientation of the main slope. This is an important difference from the DDA described in Section 3.3.1 and requires that the X delta be able to be larger than unity.

The Y-span of Area 1 is specified in Y_COUNT_3D[26:16]. This is an unsigned integer. The base point is always at the lowest Y address in the triangle (the closest to the top of the screen). The X delta is specified in DX_MAIN_3D. This is a signed real number. Since the X delta is signed, the main slope can proceed either down and to the right or down and to the left.

The second DDA finds the width of the triangle. The width delta is specified in DWIDTH1_3D. This is a signed real number. Since it is signed, the width can either increase or decrease (in practice, it always increases in Area 1 of the triangle and decreases in Area 2). Figure 3-8 shows how DWIDTH depends on both the slope of the main slope and the opposite slope. More precisely, the opposite slope is determined by DX_MAIN and DWIDTH.

The triangle is filled a scanline at a time. For each scanline, beginning at the base, the corresponding X point along the main slope is found. Then pixels are filled, along the scanline, to the current width. The DDAs are incremented, calculating the next X point along the main slope, and the next width. This continues for COUNT1 + 1 scanlines (specified in Y_COUNT_3D [26:16]), filling in Area 1 of the random triangle.

Filling in the triangle a scanline at a time is efficient in terms of memory cycle usage and computationally.

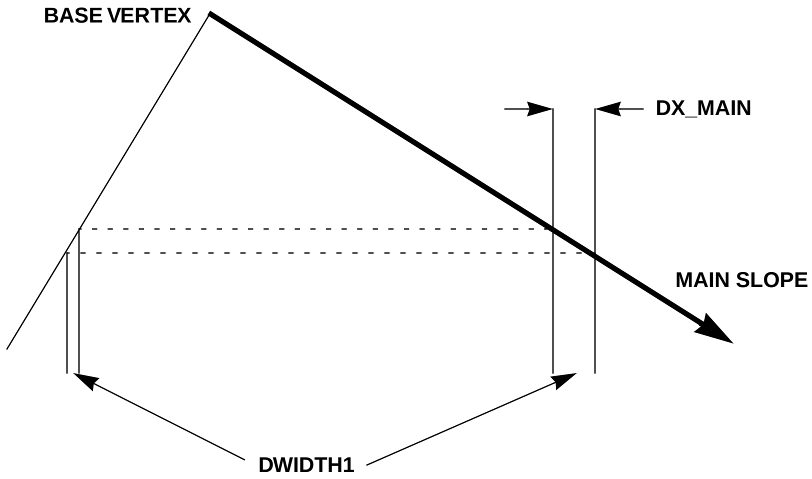

Figure 3-8. DDAs for Main Slope and Width

When the engine has filled in Area 1, it selects a new delta width constant from DWIDTH2_3D (the second delta width is nearly always negative) and continues for COUNT2 (Y_COUNT_3D_[10:0]) scanlines, filling in Area 2. In Figure 3-9, the DWIDTH2 value is negative. The triangle becomes more narrow as Area 2 is filled, coming to a point at the very bottom.

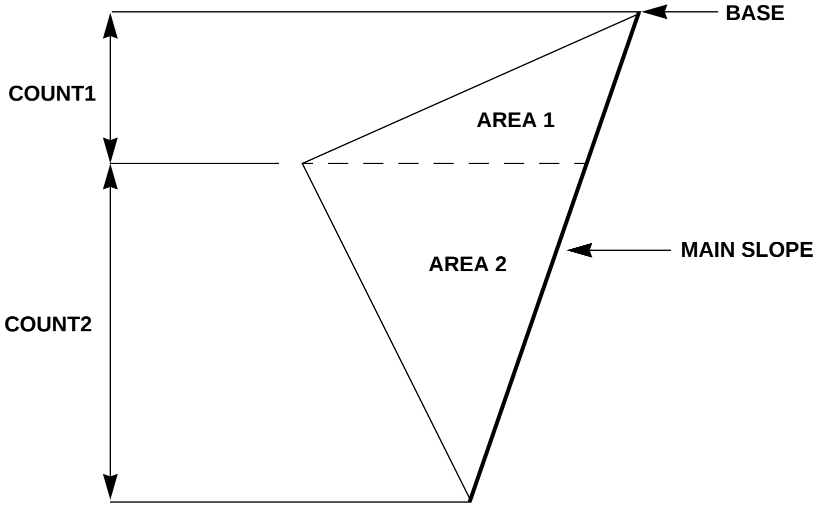

Figure 3-9. Completing the Triangle: Area 2

Observe that the main slope does not have an inflection point in it. DX_MAIN does not change when the new DWIDTH value is loaded.

If the Count2 field is zero, only Area 1 is drawn, resulting in a triangle that is flat on the bottom.

3.3.3 Summary of Values Used for Flat Triangle

Table 3-2 summarizes the values required for drawing a flat triangle. In the range column, the notation ‘.X’ indicates a 16-bit fraction. The formal description of these registers is in the *Laguna VisualMedia™ Accelerators Family — CL-GD546X Volume I (Hardware Reference Manual, Second Edition, September 1996)*.

Table 3-2. Values Used for Flat Triangle

Value	Register Name	Register Offset	Range	Note
Beginning X co-ordinate	X_3D	0x4000	0 to 2047.X	Bit 31 is X-direction
Beginning Y co-ordinate	Y_3D	0x4004	0 to 2047.X	
Red color value	R-3D	0x4008	0 to 255.X	
Green color value	G_3D	0x400C	0 to 255.X	Skip if palettized
Blue color value	B_3D	0x4010	0 to 255.X	Skip if palettized
Main Slope X increment	DX_MAIN_3D	0x4014	–2048.X to 2047.X	
Y-Count for Area 1	Y_COUNT_3D	0x4018	0 to 2047	Bits 26:16
Y-Count for Area 2	Y_COUNT_3D	0x4018	0 to 2047	Bits 10:0
Width increment for Area 1	DWIDTH1_3D	0x4024	–2048.X to 2047.X	
Width increment for Area 2	DWIDTH2_3D	0x4028	–2048.X to 2047.X	

These are the values the 3D engine has to have to draw a flat triangle. To draw a triangle in Coprocessor mode, these registers would be loaded and the engine started. To draw a triangle in Display List mode, the instruction is followed by N register values, where N is specified in the draw instruction.

3.3.4 Scaled Numbers

Most of the values listed in Table 3-2 are real numbers (they have an integer part and a fractional part). In each case where the value has a fractional part, it is scaled so that the radix point is between bit position 15 and bit position 16. Even though these are real numbers, the radix point is fixed and fixed point adders can be used. This is important both for gate count and speed considerations.

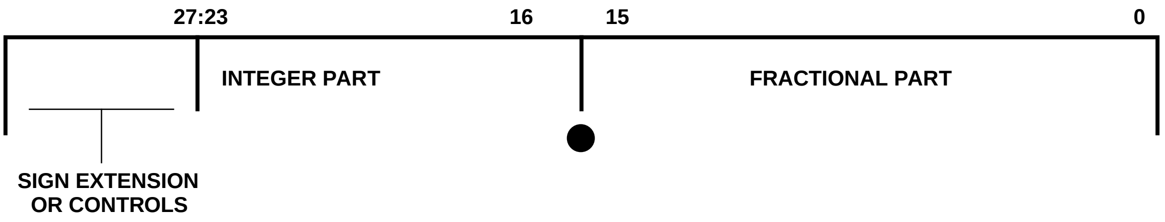

Figure 3-10. Real Number Scaling

Several of these registers have flags or control bits. Sign bits must be written into all unused bits. The control bits in these registers are summarized in Table 3-3.

Table 3-3. Control Bits

Register	Bit Position	Use
X_3D	31	X Direction: Polygon drawn to left or right from main slope
X_3D	30	Left Edge Disable: Do not draw left pixel in each scanline
X_3D	29	Right Edge Disable: Do not draw right pixel in each scanline
Y_3D	30	Top Edge Disable: Do not draw top pixel or scanline
Y_3D	29	Bottom Edge Disable: Do not draw bottom pixel or scanline

X_3D[31] controls whether the triangle is drawn to the left or the right of the main slope. This, in conjunction with the sign of the DX_MAIN_3D, determines the orientation of the triangle. All four cases are used. Figure 3-11 shows the four cases. The base point is at the top. The main slope is drawn as the heavy line in each case.

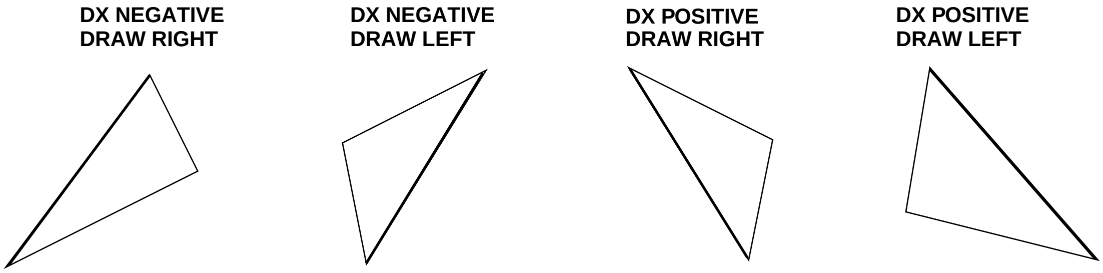

Figure 3-11. DX Sign, Draw Left/Draw Right

The four edge disables are used when triangles abut. If two adjacent triangles are drawn with blending, visible artifacts can result along their common edge if both write that common edge. By suppressing one of the two from writing the edge, these artifacts can be prevented.

3.3.5 Gouraud Shading

In addition to flat triangles, the CL-GD5464 can draw triangles with Gouraud shading. Gouraud shading uses linear interpolation, readily adaptable to a DDA. The application provides color values at the vertices, which are converted to register values by the 3D driver supplied by Cirrus Logic. This process is called 'triangle setup'.

There are a total of nine values that are involved with shading, three values for each of the three colors: Red, Green, and Blue. Each color has its base value, and its Delta_Main and Delta_Ortho values. Each base value is an unsigned number and each delta is a signed number. As usual, the radix point for each is between bit position 15 and bit position 16.

The color for each pixel along the main slope is calculated by incrementing the base color by the delta_main for each Y increment. This is exactly analogous to the calculation of the X value along the main slope. Now, as the scanline is filled, the color for each pixel is calculated by incrementing the color at the main slope by the Delta_Ortho. Since each delta for each color is signed, each color can change smoothly in two dimensions. This is illustrated in Figure 3-12 for Red. The other two colors are exactly the same, except different registers contain the constants.

Gouraud shading is enabled by programming bit 12 of the drawing instruction (that is DRAW_LINE or DRAW_POLY) to ‘1’. If Gouraud shading is enabled for draw line, the result is a shaded or depth cued line. Gouraud shading does not make any sense for a point and bit 12 must be programmed to ‘0’ in a draw point instruction. Gouraud shading can also be used with mapped color. If the color palette contains several color ranges, the shading can be arranged to fit within the ranges. In this case, the mapped color value is calculated using the Red parameters.

Note that the Delta_Main and Delta_Ortho are not necessarily orthogonal to each other. Delta_Ortho always is along the X-axis, but Delta_Main is along the main slope, which may very well not be vertical.

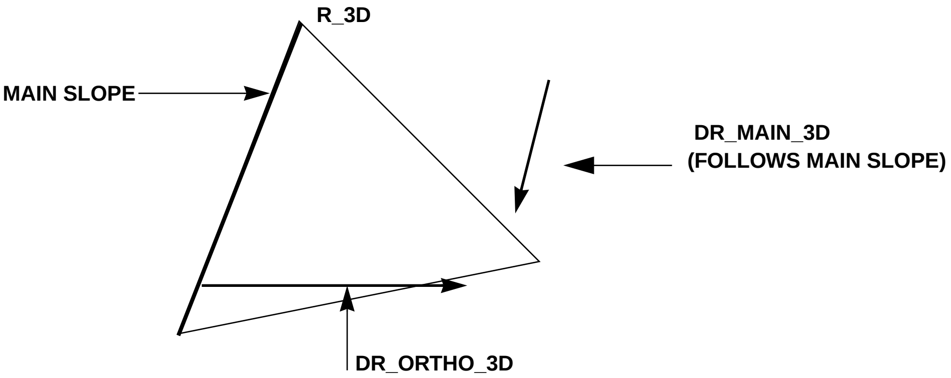

Figure 3-12. Gouraud Shading

Table 3-4 summarizes the values that are used for Gouraud shading. The Delta-Ortho values are not used for lines.

Table 3-4. Values used for Gouraud Shading

Color	Red	Green	Blue
Base color	R_3D: 0x4008	G_3D: 0x400C	B_3D: 0x4010
Delta main	DR_MAIN_3D: 0x402C	DG_MAIN_3D: 0x4030	DB_MAIN_3D: 0x4023
Delta_Ortho	DR_ORTHO_3D: 0x4038	DG_ORTHO_3D: 0x403C	DB_ORTHO_3D: 0x4040

3.3.6 X-Y Clipping

X-Y clipping is used to confine the polygon to an arbitrary rectangular region. X-Y clipping does not use any interpolators, the edges of the rectangle are fixed. Each of the four edges can be separately enabled with a '1' in the respective enable bit. Table 3-5 summarizes the clipping controls and values. If an edge is not enabled, the corresponding clipping value is ignored. The clipping values have to make sense. Programming the Max value of either dimension to less than the corresponding Min value when both are enabled is an error.

Table 3-5. X-Y Clipping Controls

Edge	Register	Enable	Clipping Value
XMax	X_CLIP_3D: 0x4160	Bit 31	26:16
XMin	X_CLIP_3D: 0x4160	Bit 15	10:0
YMax	Y_CLIP_3D: 0x4164	Bit 31	26:16
YMin	Y_CLIP_3D: 0x4164	Bit 15	10:0

3.3.7 Z-Buffering

Z-Buffering is used to remove surfaces (or parts of surfaces) that lie behind objects that are already in the scene. The depth (usually defined as the distance from the viewer) of each pixel of an object is calculated as the object is rendered. The depth of current pixel is compared to the depth of the corresponding pixel (in the frame buffer) of the object previously rendered. This requires a buffer large enough to contain a depth value for every pixel, usually called the Z-Buffer. Depending on the outcome of the comparison, the new pixel and depth can replace the previous pixel and depth. Typically the comparison is: 'Is this pixel in the current object closer to the viewer than the same pixel already in the scene?'. Thus, objects (or parts of objects) closest to the viewer are displayed on the screen.

The calculation of the Z-value for a pixel is exactly analogous to the calculation of any of the three color values. The initial value is specified for the base point. This is incremented by DZ_MAIN for each iteration in Y as the main slope is traversed. Each value along the main slope is incremented in turn by the value DZ_ORTHO for each pixel in the scanline. Since the increments are both signed, the depth of the object can change smoothly in two dimensions as the object is rendered. If the entire object is at a constant depth, the delta values would be set to zero. Table 3-6 summarizes the values that are used for Z-depth. The Delta_Ortho value is not used for lines.

Table 3-6. Values Used for Z-depth

Value	Register
Base depth	Z_3D: 0x4044
Delta main	DZ_MAIN_3D: 0x4048
Delta_Ortho	DZ_ORTHO_3D: 0x404C

The Z-depth function is enabled by programming bit 13 of the drawing instruction to '1'. Z-depth can be used for all the draw instructions. There are a number of fields that control the Z-depth function, summarized in Table 3-7 and Table 3-8.

Z-mode in CONTROL0_3D[30:28] controls whether a comparison is necessary and which buffers are updated when the compare is true.

Table 3-7. Z Mode: CONTROL0_3D[30:28]

Z-Mode	Z-Mode Name	Z-Buffer Update	Pixel Update	Description
000	Z-normal	Z-compare	Z-compare	Update both Z-buffer and frame buffer if compare true
001	Z-mask	Never	Z-compare	Update only frame buffer if compare true
010	Z-always	Always	Always	Update both Z-buffer and frame buffer without comparison
011	Z-only	Z-compare	Never	Update Z-buffer if compare true
100	Z-hit	Never	Never	Set Z-collision flag and records Z-hit and Z-value
101	Reserved	–	–	–
110	Reserved	–	–	–
111	Reserved	–	–	–

Z_COMPARE_MODE in CONTROL0_3D[23:20] specifies the compare function (that is, the relationship between the old value and new value that results in the comparison being true).

Table 3-8. Z-Compare Mode: CONTROL0_3D[23:20]

Z-Compare Mode	Compare is TRUE if:	Note
0000	New value >= old value	Greater than or equal to
0001	New value > old value	Strictly greater than
0010	New value <= old value	Less than or equal to
0011	New value < old value	Strictly less than
0100	New value != old value	Not equal
0101	New value == old value	Equal
0110–1111	Reserved	–

Z_COLLISION_DETECT_EN in CONTROL0_3D[24] specifies whether a Z-collision sets the Z_COLLISION Event Status bit in STATUS0_3D[0]. In this context, a collision is any true compare. A '1' enables the collision detection.

Z-collision provides a method of determining whether an object is totally occluded and can be bypassed. The depth of each pixel in the object is computed and compared to the depth of the corresponding pixel in the frame buffer without changing either the frame buffer or Z-buffer. Since only the position and depth of each pixel is computed, this can be substantially faster than actually rendering the object (calculating color, lighting, and so on). Once the application knows if the object is at least partially visible, it can then go back and actually render it (or at least the part that is visible).

Another use for Z-collision is to detect that two objects are within the same 'Z' space.

Z_STRIDE_CONTROL in CONTROL0_3D[16] specifies whether the Z-buffer is 8 or 16 bits per pixel. A '1' selects eight bits. This allows the Z-buffer to be stored with the actual pixels in 24-bpp modes. Each pixel actually occupies 32 bits, one byte each of Red color, Green color, Blue color, and Depth. If the fourth byte is used for the Z-buffer, it cannot simultaneously be used for Alpha.

For game programming, sometimes objects are rendered only once. Examples of such objects are the dashboard of the car in a racer simulation or the cockpit of an aircraft in a flight simulator. The 8-bit Z-stride allows for 256 levels of overlay. Thus, if the game already controls the depth, the overlay field can increase rendering speed.

Z_BUFFER_LOCATION in BASAE0_ADDR_3D[14] specifies whether the Z-buffer in is host processor memory ('1') or in the RDRAM ('0').

3.3.8 Color Transparency

Color transparency, another form of overlay, is also supported during triangle rendering. A color compare range for each of the three colors allows the background to project through. The control bits for color compare are summarized in Table 3-9.

Table 3-9. Color Compare Controls (CONTROL0_3D: 0x4104)

Control	CONTROL0_3D	Function
Color_Compare_Mode	Bit 10	1: Mask inclusive to bounds 0: Mask exclusive to bounds
Blue_Color_Compare_En	Bit 9	1: Enable blue compare
Green_Color_Compare_En	Bit 8	1: Enable green compare
Red_Color_Compare_En	Bit 7	1: Enable red compare

Table 3-10 summarizes the source of the comparison values.

Table 3-10. Color Compare Bounds

Color	Minimum COLOR_MIN BOUNDS_3D: 0x4108	Maximum COLOR_MAX_BOUNDS_3D: 0x410C
Red	[23:16]	[23:16]
Green	[15:8]	[15:8]
Blue	[7:0]	[7:0]

Depending on the color depth, the bits defining each color have to be replicated to fill the 8-bit comparison value. The following three tables show, for each pixel mode, how the color bits must be expanded to 8 bits. Each color value is replicated from left to right as many times as is necessary to fill the 8-bit compare value.

Table 3-11. Color Bit Expansion: Red

Red Value	23	22	21	20	19	18	17	16
24-bpp, a:8:8:8	23	22	21	20	19	18	17	16
16-bpp, 5:6:5	15	14	13	12	11	15	14	13
15-bpp, a:5:5:5	14	13	12	11	10	14	13	12
8-bpp, 3:3:2	7	6	5	7	6	5	7	6

Table 3-12. Color Bit Expansion: Green

Green Value	15	14	13	12	11	10	9	8
24-bpp, a:8:8:8	15	14	13	12	11	10	9	8
16-bpp, 5:6:5	10	9	8	7	6	5	10	9
15-bpp, a:5:5:5	9	8	7	6	5	9	8	7
8-bpp, 3:3:2	4	3	2	4	3	2	4	3

Table 3-13. Color Bit Expansion: Blue

Blue Value	7	6	5	4	3	2	1	0
24-bpp, a:8:8:8	7	6	5	4	3	2	1	0
16-bpp, 5:6:5	4	3	2	1	0	4	3	2
15-bpp, a:5:5:5	4	3	2	1	0	4	3	2
8-bpp, 3:3:2	1	0	1	0	1	0	1	0

3.3.9 Lighting

The lighting stage allows each of the three colors to be multiplied by a common value. This function is enabled by programming the instruction modifier bit 18 to '1'. The source for the lighting multiplier is selected with Light_Src_Sel in CONTROL0_3D[26:25], as summarized in Table 3-14.

Table 3-14. Lighting Multiplier Source

Light_Src_Sel	Source	Used For
00	Polygon engine	RGB source from interpolators in 3D engine
01	LM interpolator	Interpolated lighting
10	COLOR_REG1_3D	Fixed color lighting
11	Reserved	(unused)

3.3.10 Saturation

When the CL-GD5464 is configured for 8-bpp Color LUT mode, color saturation can be enabled by programming bit 6 of CONTROL0_3D to '1'. When this bit is set, the color value (index into the LUT) from this stage is forced to be within the values programmed in the registers summarized in Table 3-15.

Table 3-15. Color Saturation Values

Value	Register	Bits
Minimum bound	COLOR_MINBOUNDS_3D: 0x4108	31:24
Maximum bounds	COLOR_MAX_BOUNDS_3D: 0x410C	31:24

This is used when the color palette has a series of color ramps for shaded objects. It clamps the index to prevent it from drifting into the color space of other objects. This mode does not require extra overlay fields as in the 24-bpp modes, but does require some color resolution trade-off for overlay capability.

3.3.11 Alpha Blending

Alpha blending provides a means of combining the data already in the frame buffer with the object being rendered. Alpha is the blending coefficient. It determines the ratio of source to destination of the value actually written. Alpha blending is enabled by programming CONTROL0_3D[15] to '1'. The Alpha-mode field in CONTROL0_3D controls the inputs into the alpha blending stage as summarized in Table 3-16.

Table 3-16. Alpha_Mode Field (CONTROL0_3D [12:11])

Alpha Mode	Source Alpha	Destination Alpha	Note
00	DA_MAIN_3D	DA_ORTHO_3D	Fixed alpha blending
01	n/a	n/a	Reserved

Table 3-16. Alpha_Mode Field (CONTROL0_3D [12:11]) *(cont.)*

Alpha Mode	Source Alpha	Destination Alpha	Note
10	LA interpolator	255 minus LA interpolator	Interpolated alpha blending
11	Alpha byte from frame buffer	255 minus Alpha byte	Frame buffer alpha blending

Interpolated alpha blending can be used when the transparency of the object being rendered is not a constant. The LA interpolator is exactly analogous to the Gouraud shading interpolators and can be used for either Lighting or Alpha (but not both at the same time). There is an unsigned base value and two signed deltas. These values are summarized in Table 3-17. When interpolated alpha is being used, these are real numbers. As usual, the radix point is between bit position 15 and bit position 16. The two deltas are signed numbers, allowing the interpolated alpha value to vary smoothly in two dimensions as the object is rendered.

When the alpha value is interpolated, the source alpha is the interpolated value and the destination alpha is 256 minus the interpolated value.

Table 3-17. LA Interpolator Values

Value	Register
Base Alpha	A_3D: 0x40C0
Delta Main	DA_MAIN_3D: 0x40C4
Delta Ortho	DA_ORTHO_3D: 0x40C8

Fixed alpha blending can be used when the transparency of the object being rendered is constant. The contributions of the source and destination pixels are specified separately. The values are summarized in Table 3-18. The constants are interpreted as the numerator of a fractional multiplier whose denominator is fixed at 256. There is a radix point between bit position 15 and bit position 16. A constant of 0x00 means that the corresponding source or destination contributes nothing to the output color. A constant of 0x100 (decimal 256) means the corresponding source or destination contributes unity to the output value. Constants in between result in corresponding amounts contributed to the output value. *The constants do not have to add up to unity.*

Table 3-18. Fixed Alpha Constants

Multiplier	Register
Source Pixel	DA_MAIN_3D: 0x40C4
Destination Pixel	DA_ORTHO_3D: 0x40C8

Alpha_Dest_Color_Select in CONTROL0_3D[14:13] specifies the source of the destination pixel that is mixed with the object being rendered. This is summarized in Table 3-19.

Table 3-19. Alpha_Dest_Color_Select

Alpha_Dest_Color_Select	Alpha Dest Source	Note
00	Existing frame buffer data	Fetch_Color (Instruction [23]) set
01	COLOR_REG1_3D	–
10	Polygon engine interpolators	With shading, patterning
11	Reserved	–

3.3.12 Additional Notes on Lighting

The CL-GD5464 color path supports lighting, blending, and fog within a single-render operation. The data rate, after the pipeline is filled, is 12 ns/texel or 80 MHz. This allows one pixel out for every clock cycle. Now the real throughput is determined by how well the source data (that is, alpha source, texture source, and Z-buffer source) are supplied to the input side of the equation.

A simple guideline can be used to estimate the rasterization rate for 16-bpp data source. Simply divide the number of sources by the master clock rate. Some examples are given in Table 3-20.

Table 3-20. Estimating Rasterization Rate

Example	Sources	Rate at 80 MHz	Note
2D texture mapped polygon	2	40 MHz	Master clock rate / 2
3D texture polygon	4	20 MHz	Master clock rate / 4
3D texture with alpha blending	6	13.3 MHz	Master clock rate / 6

This is only a guideline and many variables apply. For example, the internal texture cache eliminates some of the texture source fetches. In fact, if the texture fits entirely in the cache, then the texture adder is eliminated altogether. In all cases, the data-path pipeline always runs through the master equation at the full 80-MHz clock rate (that is, if all source data is present, then perspective, 3D lighted, textures will output at the full 80-MHz clock).

3.3.13 Data Path Equation

The CL-GD5464 master data path equation is shown below:

$$[\{Cs(T) + Ct(1 - T)\} \bullet A] \bullet Lm + Cc(255 - A) \qquad \textbf{Equation 3-1}$$

where:

Cs is the RGB components of the source color. These are the Gouraud values from the CL-GD5464 RGB color interpolators.

T is the Transparency Enable bit, valid only in the 1:8:8:8 and 1:5:5:5 Texture modes. This bit can be considered the Alpha Texture Enable bit.

Ct is the RGB components of the texture map when in any of the 4- or 8-bit Texture Index mode or the Direct Color modes 3:3:2, 5:6:5, 1:5:5:5, 8:8:8, or 1:8:8:8. Ct is indexed texture when in the 8-bit Color Lookup mode. It shares the same color palette index (the color value) with the normal color lookup.

A is the Alpha blending term and has three sources, as shown in Table 3-16.

Lm is the modulator for lighting, and has three sources, as shown in Table 3-14.

Cc is typically the RGB destination read values, but can also take on two other forms. For DECAL texture Cc can be the RGB Gouraud interpolation values. For fog or depth cueing, Cc can be a fixed value from COLOR_REG1_3D.

This equation allows for full lighting and blending operation in a single pass, supporting fog, texture, and alpha blend.

An additional interpolation unit is used to satisfy fog rendering. For fog, an independent color ramp blended with the source fragment is required. The Cc(255-A) term in the equation becomes the Fog adder, where ‘A’ is from the separate alpha interpolator. This allows FOG on Decal textured objects when in the 1:8:8:8 and 1:5:5:5 display modes. The Lm term allows for lighting during the same render operation. It is important to remember that this can be scaled down by entering zeroes for the T and Lm fields. Here the term {CS(T)+Ct(1-T)} can be consolidated as Cf, or the result fragment. Thus the FOG equation can be re-written without lighting:

$$Cf \bullet A + Cc(255 - A) \qquad \textbf{Equation 3-2}$$

and with lighting:

$$(Cf \bullet A) \bullet Lm + Cc(255 - A) \qquad \textbf{Equation 3-3}$$

Note that during Decal Texture mode, lighting the fog requires an extra rendering step. Alternately, for Non-Decal Texture mode, the Cc term can be sourced as an independent light source for the fog component.

Depending on the speed, quality, and function requirements, the CL-GD5464 can ‘Decal’ texture in multiple ways. The simplest and most common method is to use the 1:8:8:8 or 1:5:5:5 texture modes. This gives a single-pass rendering of transparent Gouraud shading seen through textured objects. In this case the MSB of the texture map selects between shaded and textured operation. This selection is done ‘on the fly’ as the texture map information is read in the device. During the same operation, the result can also be lit with white by the addition of the ‘Lm’ term. In addition, blending of previously rendered backgrounds seen through the Decal can be added during this single-pass operation.

The following equations show three cases for the 1:8:8:8 and 1:5:5:5 modes. The MSB in the texture map is used to select between texture and shaded sources ('T' in the equations).

Trivial, no lights, no blending. 'T' is either '0' or '1':

$$\{Cs(T) + Ct(1 - T)\}$$

Equation 3-4

Decal Texture with lighting, no blending. Lm is from alpha interpolator. A=0xff.

$$\{Cs(T) + Ct(1 - T)\} \bullet Lm$$

Equation 3-5

Decal texture with lighting and blending. Cdest is read from the frame buffer, A is either fixed alpha or frame alpha (the blend factor):

$$[\{Cs(T) \bullet Ct(1 - T)\} \bullet A] \bullet Lm + Cdest(255 - A)$$

Equation 3-6

3.3.14 Texture and Perspective Texture Mapping

Texture mapping is the process of reading a 2D map or image and stretching ('mapping') that image onto a 3D surface. Conventional techniques use many small triangles to give more detail to the final image. With the process of texture mapping, large triangles can be used and the application of a detail map can be used to give a satisfactory appearance.

The CL-GD5464 uses first- and second-order differentiation to approximate the divide operation required by perspective mapping. Thus, as triangles that have texture applied are drawn into the distance, a perspective operation is applied to approximate the curve. Linear (or affine) texture does not require a perspective divide. Because of this, objects that traverse into the distance do not look correct. Algorithms that subdivide large polygons can be applied to correct the accumulated perspective error.

The texture engine uses inverse mapping techniques and runs in parallel with the polygon engine. The CL-GD5464 architecture allows for full random triangle mapping with perspective correction. The driver receives U, V, W, and texture map base-address information and converts them into values suitable for the registers. The texture map information can be in RDRAM, host system memory, or both and is read into an internal texture cache.

Texture mapping is illustrated in Figure 3-13. The location of the pixels are being calculated using the values and interpolators described in Section 3.3.2. In parallel, the U and V pointers into the texture map are being calculated with separate interpolators. Then the data from the texture map, addressed by U and V, is written into the object being rendered, addressed by X and Y. The normal shading calculations can be executed in parallel with this to provide a color to be used instead of the color from the texture map.

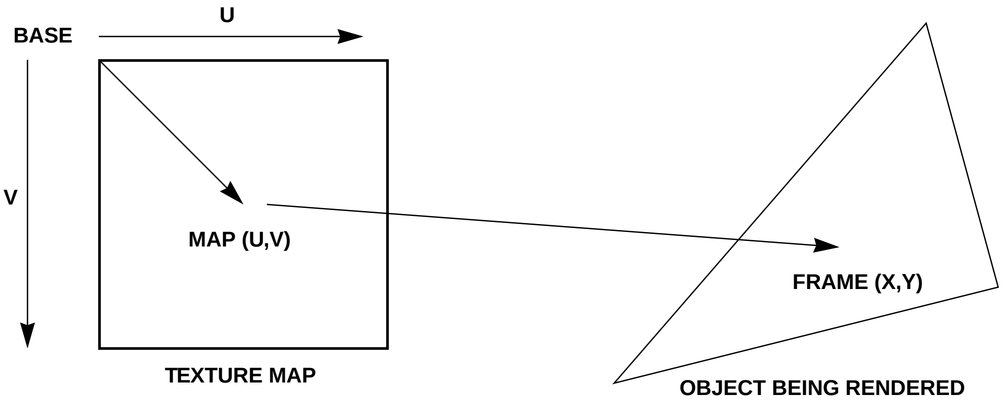

Figure 3-13. Texture Mapping Overview

In some cases, calculating pointers into the texture map using linear interpolation does not yield sufficiently convincing results. Consider the example illustrated in Section 3-14.

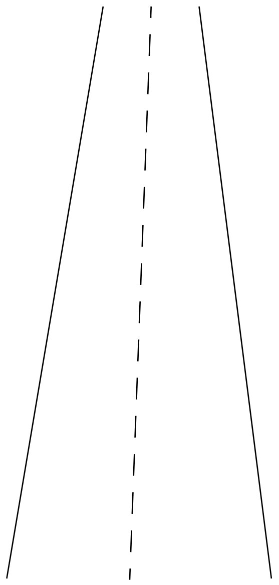

Figure 3-14. Perspective Texture Mapping Example

The actual distance (along the road) between the signs is a constant, as is the size of the signs. But the signs appear closer together as they go further into the distance (as well as appearing smaller). This is a curve and requires a divide to calculate exactly. The CL-GD5464 uses second-order terms to approximate the curve.

The texture formats are summarized in Table 3-21. The format is specified in TX_CTL0_3D [10:8].

Table 3-21. Texture Formats

Texel Size	Format	Note
4-bpp	TLUT	16 maps, each map of 16 colors from a palette of 16 million
8-bpp	TLUT	1 map, of 256 colors from a palette of 16 million
8-bpp	3:2:2	Hardware dither
16-bpp	5:6:5	Hardware dither
16-bpp	1:5:5:5	Bit 15 can be used as mask or as source select in Decal mode
32-bpp	a:8:8:8	Bit 31 can be used as mask or as source select in Decal mode
32-bpp	Z:8:8:8	8-bpp Z-buffer

In the 8-bpp LUT mode, the TLUT can be subdivided into multiple maps with the texture index offset address into the TLUT.

The source of the constants used to calculate 'V' and 'U' (the values used to select an entry in the texture map) are summarized in Table 3-22. The first six (V, DV_MAIN, DV_ORTHO, U, DU_MAIN, and DU_ORTHO) are evaluated using 'standard' interpolation, exactly the same as Gouraud shading. The second-order values are used for perspective texture mapping.

Table 3-22. Texture Mapping: V and U Constants

Constant	V	U	Note
Base point	V_3D: 0x4050	U_3D: 0x4054	'Standard' interpolation
Delta main	DV_MAIN_3D: 0x4058	DU_MAIN_3D: 0x405C	
Delta_Ortho	DV_ORTHO_3D: 0x4060	DU_ORTHO_3D: 0x4064	
Second-order main	D2V_MAIN_3D: 0x4068	D2U_MAIN_3D: 0x406C	Perspective only
Second-order ortho	D2V_ORTHO_3D: 0x4070	D2U_ORTHO_3D: 0x4074	
Ortho add	DV_ORTHO_ADD_3D: 0x4078	DU_ORTHO_ADD_3D: 0x407C	

3.3.15 Quadrangles

The CL-GD5464 can draw quadrangles that have a flat top or bottom (or both). Table 3-23 summarizes two registers that can be used to set the initial width (at the top of) Area 1 and Area 2. These are both unsigned numbers in the range 0 to 2047.X. As usual, there is a radix point between bit position 15 and bit position 16.

Table 3-23. Initial Width Value

Area	Register
1	WIDTH1_3D: 0x401C
2	WIDTH2_3d: 0x4020

A quadrangle that is flat on the top is shown in Figure 3-15. As always, one side has to span the entire height of the polygon. The base point is at the top of the main slope. Since the initial width is non-zero, the top of the polygon is not a point, but is rather a flat line. The initial widths are enabled by programming bit 24 of the draw instruction to '1'. Width1 and Width2 are enabled together. If one is required, the other must be specified even though it may not be otherwise required.

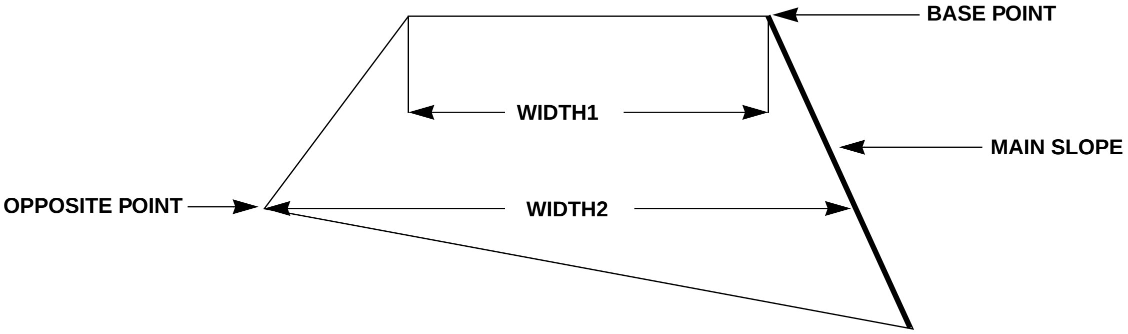

Figure 3-15. Quadrangle with Flat Top

A quadrangle that is flat on the bottom is shown in Figure 3-16. This can be drawn by the simple expedient of setting the delta width2 (DWIDTH2) so that the object does not come to a point at the bottom. It is possible to draw quadrangles that are flat both on the top and bottom.

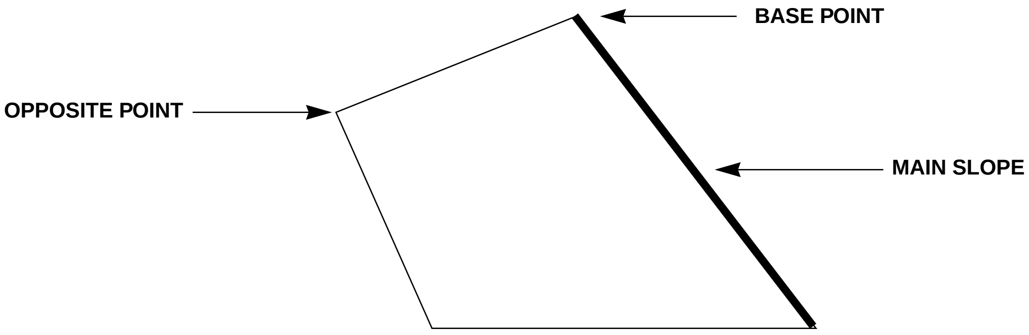

Figure 3-16. Quadrangle with Flat Bottom

3.3.16 Lines and Points

Lines and points can be considered small polygons. Different instructions are used to draw them and different (fewer) registers are used. Table 3-24 summarizes the registers required for each object. 'Skip if' means the object requires the register unless one or more of the skip cases is programmed.

The CL-GD5464 can draw lines with Gouraud shading, Z-buffering, and alpha blending. Basically, a line is just the main slope of a polygon. The Delta-Orthos are not used, but everything that applies to the main slope of a polygon applies to a line, including texturing.

When lines are drawn with a slope of less than 45 degrees, they are X-major rather than Y-major. See Section 3.5.3.5.

Table 3-24. Register Summary

Address Offset	Register	Polygon	Line	Point	Skip Case(s)
0x4000	X_3D	Always	Always	Always	–
0x4004	Y_3D	Always	Always	Always	–
0x4008	R_3D	Skip if	Skip if	Skip if	–
0x400C	G_3D	Skip if	Skip if	Skip if	Mapped color
0x4010	B_3D	Skip if	Skip if	Skip if	Mapped color
0x4014	DX_MAIN_3D	Always	Always	Never	–
0x4018	Y_COUNT_3D	Always	Always	Never	–
0x401C	WIDTH1_3D	Skip if	Always	Never	Initial width off
0x4020	WIDTH2_3D	Skip if	Never	Never	Initial width off
0x4024	DWIDTH1_3D	Always	Never	Never	–
0x4028	DWIDTH2_3D	Always	Never	Never	–
0x402C	DR_MAIN_3D	Skip if	Skip if	Never	Gouraud off
0x4030	DG_MAIN_3D	Skip if	Skip if	Never	Mapped color, Gouraud off
0x4034	DB_MAIN_3D	Skip if	Skip if	Never	Mapped color, Gouraud off
0x4038	DR_ORTHO_3D	Skip if	Never	Never	Gouraud off
0x340C	DG_ORTHO_3D	Skip if	Never	Never	Gouraud off
0x4040	DB_ORTHO_3D	Skip if	Never	Never	Gouraud off
0x4044	Z_3D	Skip if	Skip if	Skip if	Z off
0x4048	DZ_MAIN_3D	Skip if	Skip if	Never	Z off
0x404C	DZ_ORTHO_3D	Skip if	Never	Never	Z off

Table 3-24. Register Summary *(cont.)*

Address Offset	Register	Polygon	Line	Point	Skip Case(s)
0x4050	V_3D	Skip if	Skip if	Skip if	Texture off
0x4054	U_3D	Skip if	Skip if	Skip if	Texture off
0x4058	DV_MAIN_3D	Skip if	Skip if	Never	Texture off
0x405C	DU_MAIN_3D	Skip if	Skip if	Never	Texture off
0x4060	DV_ORTHO_3D	Skip if	Never	Never	Texture off
0x4064	DU_ORTHO_3D	Skip if	Never	Never	Texture off
0x4068	D2V_MAIN_3D	Skip if	Skip if	Never	Texture off, perspective off
0x406C	D2U_MAIN_3D	Skip if	Skip if	Never	Texture off, perspective off
0x4070	D2V_ORTHO_3D	Skip if	Never	Never	Texture off, perspective off
0x4074	D2U_ORTHO_3D	Skip if	Never	Never	Texture off, perspective off
0x4078	DV_ORTHO_ADD_3D	Skip if	Never	Never	Texture off, perspective off
0x407C	DU_ORTHO_ADD_3D	Skip if	Never	Never	Texture off, perspective off
0x40C0	A_3D	Skip if	Skip if	Never	Alpha load off
0x40C4	DA_MAIN_3D	Skip if	Skip if	Never	Alpha load off
0x40C8	DA_ORTHO_3D	Skip if	Never	Never	Alpha load off

3.4 3D Memory Organization

The CL-GD5464 has several models for 3D memory. The basic views of memory are listed below.

1) The view of frame buffer memory from the system processor (across PCI bus).
2) The view of the register set from the system processor (across PCI bus).
3) The view of the following system memory objects from the CL-GD5464 viewpoint:
   - **a)** Display list memory.
   - **b)** Z-buffer and color map buffers when in system memory.
   - **c)** Texture memory when in system memory.
4) The **READ_DEV_REGS_3D —** multiple command's view of the system memory location where to write data.

3.4.1 System Memory Space View of Frame Buffer Memory

The CL-GD5464 register set is visible in system memory at the location selected by the PCI configuration register for the Memory-Mapped registers (PCI10). Registers can be written directly by adding their offsets to the initial CL-GD5462 Memory-Mapped register address. Alternatively, one or more registers can be written in a stream by the WRITE_3D_REGISTER (new CL-GD5464 registers), or the WRITE_DEVICE_REGS (CL-GD5462 registers) commands. This last operation can either be done by a display list or by directly programming its data to the HOST_3D_DATA offset from the base PCI Address register for the Memory-Mapped registers.

3.4.2 System Processor (Across PCI Bus) View of the Register Set

The CL-GD5464 Memory-Mapped 3D registers are listed in Table 3-25.

Table 3-25. CL-GD5464 Memory-Mapped 3D Registers

	Byte Lane			
Offset	**3**	**2**	**1**	**0**
4000h	X_3D			
4004h	Y_3D			
4008h	R_3D			
400Ch	G_3D			
4010h	B_3D			
4014h	DX_MAIN_3D			
4018h	Y_COUNT_3D			
401Ch	WIDTH1_3D			
4020h	WIDTH2_3D			
4024h	DWIDTH1_3D			
4028h	DWIDTH2_3D			

Table 3-25. CL-GD5464 Memory-Mapped 3D Registers *(cont.)*

	Byte Lane			
Offset	**3**	**2**	**1**	**0**
402Ch	DR_MAIN_3D			
4030h	DG_MAIN_3D			
4034h	DB_MAIN_3D			
4038h	DR_ORTHO_3D			
403Ch	DG_ORTHO_3D			
4040h	DB_ORTHO_3D			
4044h	Z_3D			
4048h	DZ_MAIN_3D			
404Ch	DZ_ORTHO_3D			
4050h	V_3D			
4054h	U_3D			
4058h	DV_MAIN_3D			
405Ch	DU_MAIN_3D			
4060h	DV_ORTHO_3D			
4064h	DU_ORTHO_3D			
4068h	D2V_MAIN_3D			
406Ch	D2U_MAIN_3D			
4070h	D2V_ORTHO_3D			
4074h	D2U_ORTHO_3D			
4078h	DV_ORTHO_ADD_3D			
407Ch	DU_ORTHO_ADD_3D			
4080h:40BCh				
40C0h	A_3D			
40C4h	DA_MAIN_3D			
40C8h	DA_ORTHO_3D			
40CCh:40F8h				
40FCh	OPCODE_3D			
4100h	CONTROL_MASK_3D			
4104h	CONTROL0_3D			

Table 3-25. CL-GD5464 Memory-Mapped 3D Registers *(cont.)*

Byte Lane				
Offset	**3**	**2**	**1**	**0**
4108h	COLOR_MIN_BOUNDS_3D			
410Ch	COLOR_MAX_BOUNDS_3D			
4110h	CONTROL1_3D			
4114h	BASE0_ADDR_3D			
4118h	BASE1_ADDR_3D			
411Ch	Reserved			
4120h	TX_CTL0_3D			
4124h	TX_XYBASE_3D			
4128h	TX_CTL1_3D			
412Ch	TX_CTL2_3D			
4130h	COLOR_REG0_3D			
4134h	COLOR_REG1_3D			
4138h	Z_COLLIDE_3D			
413Ch	STATUS0_3D			
4140h	PATTERN_RAM_0_3D			
4144h	PATTERN_RAM_1_3D			
4148h	PATTERN_RAM_2_3D			
414Ch	PATTERN_RAM_3_3D			
4150h	PATTERN_RAM_4_3D			
4154h	PATTERN_RAM_5_3D			
4158h	PATTERN_RAM_6_3D			
415Ch	PATTERN_RAM_7_3D			
4160h	X_CLIP_3D			
4164h	Y_CLIP_3D			
4168h	TEX_SRAM_CTRL_3D			
416Ch:41FCh				
4200h	HXY_BASE0_ADDRESS_PTR_3D			
4204h	HXY_BASE0_START_3D			
4208h	HXY_BASE0_EXTENT_3D			

Table 3-25. CL-GD5464 Memory-Mapped 3D Registers *(cont.)*

	Byte Lane			
Offset	**3**	**2**	**1**	**0**
420Ch	Reserved			
4210h	HXY_BASE1_ADDRESS_PTR_3D			
4214h	HXY_BASE1_OFFSET0_3D			
4218h	HXY_BASE1_OFFSET1_3D			
421Ch	HXY_BASE1_LENGTH_3D			
4220h:43Ch				
4240h	HXY_HOST_CTRL_3D			
4244h:425Ch				
4260h	MAILBOX0_3D			
4264h	MAILBOX1_3D			
4268h	MAILBOX2_3D			
426Ch	MAILBOX3_3D			
4270h:43FCh				
4400h	PF_BASE_ADDR_3D			
4404h	PF_CTL_3D			
4408h	PF_DEST_ADDR_3D			
440Ch	PF_FB_SEG_3D			
4410h:441Ch				
4420h	PF_INST_ADDR_3D			
4424h	PF_STATUS_3D			
4428h:443Ch				
4440h	HOST_MASTER_CTL_3D			
4444h:447Ch				
4480h	PF_INST_3D			
4484h:47FCh				
4800h:4BFCh	HOST_3D_DATA_PORT			
4C00h:4FFCh	HOST_TEXTURE_DATA_PORT			

3.4.2.1 Memory-Mapped I/O

Most registers in the CL-GD5464 are accessed using memory-mapped I/O. There is a 32-Kbyte extent, comprising four 4-Kbyte apertures. The programmer should program the base address into PCI10: MMI/O Base Address register.

The registers that are accessible using memory-mapped I/O are described in the *Laguna VisualMedia™ Accelerators Family — CL-GD546X Volume I (Hardware Reference Manual, Second Edition, September 1996)*. The MMI/O offset for each register is given in the register description, and in the summary table at the beginning of each respective chapter.

The 2D Graphics Accelerator register set resides at the beginning of the 32-Kbyte extent of memory. This aperture is 16-Kbyte in length to provide for four different bi-endian data swapping modes. The 3D register set begins at the next 16-Kbyte above the 2D register set (for a total of 32 Kbytes) and is organized as shown in Table 3-26.

Table 3-26. CL-GD5464 Register Apertures

MMI/O Offset	Contents	Format
3D Register Set		
7000h	CL-GD5464 4-Kbyte register aperture	Bytes swapped within dword
6000h	CL-GD5464 4-Kbyte register aperture	Bytes swapped within dword (same as above)
5000h	CL-GD5464 4-Kbyte register aperture	Bytes swapped within words
4000h	CL-GD5464 4-Kbyte register aperture	No swapping (default)
2D Register Set		
3000h	CL-GD5462 4-Kbyte aperture	Bytes swapped within dword
2000h	CL-GD5462 4-Kbyte aperture	Bytes swapped within dword (same as above)
1000h	CL-GD5462 4-Kbyte aperture	Bytes swapped within words
0000h	CL-GD5462 4-Kbyte aperture	No swapping

NOTE: In this manual, the address of the register in the first aperture is used generically to represent register locations.

3.4.2.2 I/O-Mapped Registers

There are registers accessible using normal I/O.

These are the VGA Core registers and Extended I/O registers, described in the *Laguna VisualMedia™ Accelerators Family — CL-GD546X Volume I (Hardware Reference Manual, Second Edition, September 1996)*. The I/O-Mapped registers have fixed addresses that are nearly all standard VGA.

A few registers are accessible both in the memory and I/O space. Most of these are in the CRT Controller and each have addresses in the appropriate columns of the summary tables in each chapter.

3.4.3 System Memory Objects View from the CL-GD5464

3.4.3.1 3D General System Memory Objects

There are three memory objects that can be in system memory (logically four, since Z and color buffer are combined). These are Z-buffer, color buffer, texture buffer, and display list buffer. The display list can only be in system memory.

Each of these objects must be in locked memory for the CL-GD5464 PCI bus master to reliably locate them. They can either be located with a fixed, contiguous physical memory region of up to 4 Mbytes or in a non-contiguous virtual memory space of up to 4 Mbytes. The CL-GD5464 provides the translation from virtual-to-physical addresses through a software-created translation table in locked physical memory. These two cases are distinguished by a control within CL-GD5464 as well as the indication as to whether the objects are in system memory or in frame buffer memory.

3.4.3.2 Virtual Memory Translation

The 3D engine's XY to linear conversion generates a 22-bit byte address ('A') corresponding to the XY address. Some memory objects such as 'Z', texture, and color buffers have offsets that are added to X and Y prior to this conversion. This address is illustrated below.

Table 3-27. Linear Address

	31 22	**21 12**	**11 0**
Field Name	A0	A1	A2
	Bits 31-22 of virtual address (these bits are ignored)	Bits 21:12 of virtual address	Bits 11:0 of virtual address

The ***virtual memory address*** ('**A'**) is treated in one of two ways. For virtual memory translation (bit 0 of the BASE_ADDRESS_XX register for that object is '1'), bits 31:12 of the BASE_ADDRESS_XX register ('**B'**) point to a physical memory location for a 4-Kbyte virtual translation table for that system memory object.The Base Address ('B') for the system memory object is illustrated below.

Table 3-28. Base Address

	31 12	**11 1**	**0**
Field Name	B0	B1	B2
	Bits 31:12 of system memory base address (points to the virtual address translation table for this object — on 4-Kbyte boundary).	Reserved in virtual mode. Write 0s	0 = Physical 1 = Virtual

Bits 21:12 [***A1***] of the ***virtual memory address*** [***A***] index one of 1024 dword entries in the virtual memory translation table. The 4-Kbyte translation table in physically locked down system memory

contains 1-Kbyte, 32-bit translation entries. (This yields a 4-Mbyte maximum size for each system memory object). ***Translation table entries*** (‘**C**’) have the following format illustrated below.

Table 3-29. Translation Table Entry

	31 12	**11 1**	**0**
Field Name	C0	C1	C2
	Bits 31:12 of physical address	0... 0	1 = page present 1 = page not present

Each 32-bit ***translation table entry*** defines the upper 20 bits of the final physical address that corresponds to the original virtual address. The translations for four entries in the table (a set for each object) are cached in the CL-GD5464 to reduce translation table fetches for high-locality accesses. The ***final physical address*** corresponding to the original XY address is illustrated below.

Table 3-30. Final Physical Address

	31 12	**11 2**	**1 0**
Field Name	D0	D1	D2
	Bits 31:12 of C (field C0)	Bits 11:2 of A (field A2)	00

Bits 11:2, of the virtual address, are concatenated to the upper 20 bits just looked up in the table to form the physical address used by the CL-GD5464 to fetch the indicated data.

3.4.3.3 Physical Memory Addressing

The engine’s XY to linear conversion generates a 22-bit byte address (‘A’) corresponding to the XY address. (Some memory objects such as Z, texture, and color buffers have offsets that are added to X and Y prior to this conversion. The virtual memory address (‘A’), resulting from XY linear conversion (23-bit byte address), is illustrated below.)

Table 3-31. Linear Address

	31 22	**21 12**	**110**
Field Name	A0	A1	A2
	Bits 31:22 of virtual address (these bits are ignored)	Bits 21:12 of virtual address	Bits 11:0 of virtual address

For physical memory translation (bit 0 of the BASE_ADDRESS_XX register for that object is 0), field ***A1*** of the resulting ***virtual memory address*** [**A**] is added to field ***B0*** (bits 31-12) of the BASE_ADDRESS_XX register [**B**]. This scheme assumes an up to 4 Mbyte physically contiguous locked down region is being used in system memory for the object. The Base Address B for the system memory object is illustrated below.

Table 3-32. Base Address

	3112	**11 2**	**1 0**
Field Name	B0	B1	B2
	Bits 31:12 of system memory base address (points to the starting physical address for this object — on 4-Kbyte boundary).	Base address physical range (see BASE_ADDR_XX)	0 = Physical 1 = Virtual

The result ***C0*** is concatenated to field ***A2*** of the original virtual memory address to form the final physical address. The ***final physical address***, corresponding to the original XY address, is illustrated below. The final physical address is illustrated below.

Table 3-33. Final Physical Address

	31 12	**11 2**	**1 0**
Field Name	C0	C1	C2
	Sum of fields A0 and field B0	Copied from field A2	00

3.4.3.4 3D Display List Memory

The display list memory object behaves exactly as described above. It uses 4400h, the INSTRUCTION_PTR_3D register, as its equivalent to the BASE_ADDRESS_XX registers.

3.4.3.5 Z-Buffer and Color Map Buffers Mixed in System Memory

TBD

3.4.3.6 Texture Memory Format in System Memory

TBD

3.4.4 READ_3D_REGISTER — Multiple Commands View of System Memory

TBD

3.5 CL-GD5464 3D Instruction Set

The CL-GD5434 3D engine is a stored-program computer with its own instruction set. This section covers the instructions, including the formats and field definitions. In Section 3.5.3, each instruction is described in detail.

Many instructions have field definitions that are common. An example is the EVENT_MASK in bits 10:0 of TEST and WAIT instructions. These common fields are defined in Section 3.5.2 and then referenced in the individual instruction definitions.

3.5.1 Instruction Summary

Each instruction on the CL-GD5464 is precisely 32-bits long and must reside in system memory on a DWORD boundary. Instructions are often followed by a list of parameters. For example, DRAW_POINT is followed by parameters that define the location of the point in display memory space (X, Y, Z) and color space (R, G, B).

Instructions can be loaded into the CL-GD5464 by the host (see Section 3.2.1 and Section 3.2.2), but are more often fetched by the CL-GD5464 itself using the bus master capabilities of the PCI host interface (display list programming). In the cases where it makes a difference in the description, this section assumes display list programming.

3.5.1.1 Drawing Instructions

There are three drawing instructions. The format of the drawing instructions is shown in Table 3-34.

Table 3-34. Drawing Instruction Format Summary

Field	Bits	Description
OP_CODE	31:27	Specifies instruction
STALL	26	Stall control for all opcodes
INSTR_MODIFIER	25:12	Controls details of instruction execution (seeTable 3-41)
ADDR	11:6	Destination register for first parameter, usually '0'
COUNT	5:0	Number of parameter words following the instruction word (0 to 63)

The opcodes for the drawing instructions are summarized in Table 3-35.

Table 3-35. Drawing Instruction Opcodes

Instruction	Opcode (Binary)	Hex Value (32-bit Template)
DRAW_POINT	00000	0000 0000h
DRAW_LINE	00001	0800 0000h
DRAW_POLY	00010	1000 0000h

The number and order of parameters used by the drawing instruction varies according to the instruction and the modifier bits. Typically, DRAW_POINT uses fewer parameters than DRAW_LINE, which uses fewer parameters than DRAW_POLY. The description of each DRAW instruction includes a table specifying the order of the parameters, and the parameter(s) that are skipped based on the modifier bits.

The ADDR field specifies the register that the first parameter is to be loaded into. Since the first parameter is nearly always X_3D, this field is nearly always programmed to ‘0’. The COUNT field specifies the number of parameters. This can be as few as three for a DRAW_POINT to over 30 for a DRAW_POLY.

3.5.1.2 Transfer Instructions

There are four instruction used to transfer control within a display list. The format of the transfer instructions is shown in Table 3-36.

Table 3-36. Transfer Instruction Format Summary

Field	Bits	Description
OP_CODE	31:27	Specifies instruction
STALL	26	Stall control for all opcodes
(Reserved)	25:22	Must be zero
OFFSET_ADDR	21:2	Destination address
(Reserved)	1:0	Must be zero

The opcodes for the transfer instructions are summarized in Table 3-37.

Table 3-37. Transfer Instruction Opcodes

Instruction	Opcode (binary)	Hex Value (32-bit Template)	Note
BRANCH	00111	3800 0000h	Unconditional
C_BRANCH	01000	4000 0000h	Transfer if condition true
NC_BRANCH	01001	4800 0000h	Transfer if condition false
CALL	01010	5000 0000h	Unconditional, store return

These four instructions conditionally or unconditionally transfer control within a display list. If the 3D engine is not in Display List mode, execution of any of these instructions puts it into Display List mode.

The two conditional branch instructions test a single-condition bit, which must have been previously set or reset with a TEST instruction. The actual condition tested is specified in the TEST instruction.

The CALL instruction stores the offset of the next instruction so that it can be restored with a RETURN instruction. The CL-GD5464 supports a single level of subroutine. If a CALL is executed inside a subroutine, the original return address is lost.

The OFFSET field is added to the PF_BASE_ADDR_3D register to obtain the target address. Instructions are always on DWORD boundaries; the least-significant two bits of the OFFSET_ADDR and PF_BASE_ADDR_3D must always be '0'.

3.5.1.3 Control Instructions

There are nine control instructions with various formats. The formats are shown in Table 3-38 through Table 3-40.

Table 3-38. TEST/WAIT Instruction Format Summary

Field	Bits	Description
OP_CODE	31:27	Specifies instruction
STALL	26	Stall control for all opcodes
AND_OR	25	Controls how tests are combined
NOT	24	Inverts the sense of the test
(Reserved)	23:11	Must be zero
EVENT_MASK	10:0	Conditions are individually enabled

The purpose of the TEST instruction is to set or reset the condition flag for a subsequent condition branch instruction. The purpose of the WAIT instruction is to wait for an event or combination of events to be true or false before continuing the instruction stream. The WAIT instruction does not change the condition flag.

The AND_OR bit and NOT bit control how conditions are combined. The details are included in the instruction descriptions.

The EVENT_MASK consists of individual bits for a number of events or conditions. They are enumerated in Table 3-42. Since there is a bit for each event or condition, multiple events, or conditions can be combined in a single TEST or WAIT instruction. The AND_OR and NOT bits control how they are combined.

Table 3-39. CONTROL 'I' Instruction Format Summary

Field	Bits	Description	Note
OP_CODE	31:27	Specifies instruction	
STALL	26	Stall control for all opcodes	
SUB_OPCODE	25:22	Specifies one of 16 operations	Six are used
(Reserved)	21:0	Must be zero	

Six CONTROL 'I' instructions are available. They are distinguished by a four-bit SUB_OPCODE, summarized in Table 3-40.

Table 3-40. CONTROL 'I' Instructions

Instruction	SUB_OPCODE	32-bit Instruction	Function
IDLE	0000b	6800 0000h	Idle prefetch unit (return to Coprocessor mode).
IDLE_INT	0001b	6840 0000h	Idle prefetch unit (return to Coprocessor mode) and set host interrupt request.
NOP	0010	6880 0000h	No operation. Can be used with STALL bit to provide draw engine and prefetch engine stall.
RETURN	0011b	68C0 0000h	Return from subroutine. Transfers control to the instruction following the last CALL.
RETURN_INT	0100b	6900 0000h	Return from interrupt subroutine.
CLEAR	1011b	6AC0 0000h	Clear Execution Engine registers (4000h to 40FCh) to zero.

3.5.1.4 Read/Write Register Instructions

There are five instructions used to read or write registers in the CL-GD5464. These each have a different format; the formats of these instructions are given in the respective instruction descriptions.

READ_DEV_REGS can be used to read an arbitrary register from any unit in the CL-GD5464. The register contents are stored in system memory at the offset specified in PF_DEST_ADDR_3D. On the CL-GD5464, READ_DEV_REGS is limited to a single register for each instruction.

The register to be read is specified in two fields. The MODULE_SELECT field indicates the internal module containing the register (see Table 3-43). The ADDR field specifies, within the module, the register to be read.

WRITE_DEV_REGS is the analog of READ_DEV_REGS to write an arbitrary register. The parameter(s) to be written follows the instruction in the display list. On the CL-GD5464, WRITE_DEV_REGS is limited to a single register unless 2D Engine registers 480h through 4FCh are the target.

The register to be written is specified in two fields. The MODULE_SELECT field indicates the internal module containing the register (see Table 3-43). The ADDR field specifies, within the module, the register to be written.

WRITE_REGISTER is used to write a contiguous set of the CL-GD5464 3D registers. The instruction contains two fields that select the first register that is to be written and a count specifying the number of registers to be written. The parameters that are to be written follow the instruction in the display list. This instruction can also be used in Coprocessor Direct mode.

WRITE_DEST_ADDR is used to write an offset to the PF_DEST_ADDR_3D register. This offset is subsequently used by READ_DEV_REGS.

WRITE_PFCTRL_REG is used to write directly to the PF_CTL_3D register at 4404h or the PF_FB_SEG_3D register at 440Ch.

3.5.2 Instruction Field Tables

This section defines the common fields in the instructions. These descriptions are in no particular order. They are referenced in the descriptions of the individual instructions that use them.

3.5.2.1 STALL

The STALL bit is in all the CL-GD5464 instructions. It is always bit 26. If the STALL bit is set, all DRAW instructions that have been issued are completed before beginning the current instruction (the one with the STALL bit set).

3.5.2.2 Drawing Instruction INSTR_MODIFIER Field

These bits are used in conjunction with the parameters to define just how the drawing instruction is to be executed. Many of these bits can be used in combination to obtain the desired effect. See the sample programs.

Table 3-41. Drawing Instructions INSTR_MODIFIER Field

INSTR_MODIFIER	Bits	Description
MODIFIER_EXP	25	Next word contains additional modifiers (for further expansion). This bit must be '0' on the CL-GD5464.
INITIAL_WIDTH	24	Initial triangle span widths required (for Area 1 and Area 2). Allows drawing polygons that have an initial width that is non-zero.
FETCH_COLOR	23	Fetch existing color buffer pixels. Turn destination read on. This is used for color range compare and mask as well as alpha blending when the blending co-efficient is stored with the data in the frame buffer.
ALPHA_LOAD	22	Load alpha registers A_3D, DA_MAIN_3D and DA_ORTHO_3D. Used for fog or light gradient write.
DITHER	21	Dither polygon using PATTERN_RAM as dither pattern. This improves color quality during lighting and non-3D shading process.
PATTERN	20	Pattern polygon using PATTERN_RAM
STIPPLE	19	Stipple polygon for transparency using PATTERN_RAM
LIGHTING	18	Global enable for lighting
TEXTURE_MODE	17:16	00: Texture mapping off 01: Reserved 10: Linear texture mapping 11: Perspective-corrected texture mapping
RESERVED	15:14	Reserved for expansion
Z_ON	13	Z depth function on: 3D operation
GOURAUD	12	Interpolate colors. The delta color parameters are used to obtain color gradients. Must be '0' for DRAW_POINT.

3.5.2.3 TEST/WAIT Instruction EVENT_MASK

This field is individual bits rather than encoded values. This allows multiple conditions to be tested in a single instruction.

Table 3-42. TEST/WAIT Instruction EVENT_MASK

Bit	Hex Value	Event Name	Description	Reset By:
10	0x400	DISPLAY_LIST_SWITCH	Set by system software to indicate that display can be switched to a new buffer	Software
9	0x200	COMMAND_FIFO_NOT_EMPTY	2D/3D command FIFO is not empty	BLT engine
8	0x100	BLT_ENGINE_BUSY	Neither BLT_BUSY nor BLT_READY are true	BLT engine
7	0x080	HOSTXY_UNIT_BUSY	HostXY unit is busy	XY unit
6	0x040	EXECUTION_ENGINE_BUSY	Execution engine is busy	Execution engine
5	0x020	POLY_ENGINE_BUSY	3D engine is busy	Polygon engine
4	0x010	Z_BUFFER_COMPARE	Z-compare produces true result	
3	0x008	CRT_DISPLAY_BUFFER_SWITCH	CRT Controller switches banks; used for double buffer of display screen	Next CRT VSYNC
2	0x004	CRT_LINE_COMPARE	CRT vertical counter equals LINE_COMPARE	Next CRT VSYNC
1	0x002	CRT_EVSYNC	CRT vertical counter equals CRT VSYNC END	Next CRT VSYNC
0	0x001	CRT_VSYNC	CRT vertical counter equals CRT VSYNC START	Next CRT VSYNC

3.5.2.4 READ/WRITE_DEV_REGS Instruction MODULE_SELECT

This field specifies the internal module for the READ_DEV_REGS and WRITE_DEV_REGS instructions.

Table 3-43. READ/WRITE_DEV_REGS Instruction MODULE_SELECT

Module	Value	Description
VGAMEM	00 000b	VGA register set (I/O, palette, video CRT, cursor)
VGAFB	00 001b	VGA frame buffer memory (A0000h–BFFFFh)
VPORT	00 010b	VPORT Control registers
LPB	00 011b	Local peripheral bus (general-purpose I/O)
MISC	00 100b	Miscellaneous (Rambus® and serial)
ENG2D	00 101b	2D engine registers
HD	00 110b	2D engine host data port
FB	00 111b	Direct frame buffer memory
ROM	01 000b	ROM memory (read-only)
ENG3D	01 001b	3D engine register set (offset 4000h–41FCh)
HOST_XY	01 010b	3D engine host XY register set (offset 4200–42FCh)
HDATA_3D	01 011h	3D (command) data port (offset 4800h–4BFCh)

3.5.2.5 DRAW Instructions Register Skip Controls

For each of the three DRAW instructions, a number of conditions can cause registers to be skipped as the parameters are loaded. For example, if mapped color is being used, there is no requirement to load the blue or green color parameters. Table 3-44 summarizes the conditions and shows which parameters are skipped for each.

Table 3-44. Register Skip Conditions

Condition	Description	Parameters Skipped	Causes Condition
Mapped color	The CL-GD5464 is programmed for CLUT mapped color. A single color value is stored per pixel.	Greens, Blues	CONTROL0_3D[2:0] = 000b
Shading disabled	Color is static	Delta colors	Instruction [12] = 0
Z-buffer off	Z-buffer functions disabled	Z, Delta Zs	Instruction [13] = 0
No initial width	Polygon has vertex at top	WIDTH1, WIDTH2	Instruction [24] = 0
Texture off	No texture mapping	U, V, DU, DV	Instruction [17:16] = 0x
Perspective texture off	Linear texture mapping	D2U, D2V, D_ADD	Instruction [17:16] = 10
ALPHA_LOAD Off	No alpha blending	A, DA	Instruction [22] = 0

3.5.3 Instruction Listings

The following sections describe the instructions in detail. The descriptions are ordered alphabetically.

Table 3-45. Instruction Descriptions

Instruction	Section	Page
BRANCH	3.5.3.1	page 49
CALL	3.5.3.2	page 50
C_BRANCH	3.5.3.3	page 51
CLEAR	3.5.3.4	page 52
DRAW_LINE	3.5.3.5	page 53
DRAW_POINT	3.5.3.6	page 55
DRAW_POLYGON	3.5.3.7	page 56
IDLE	3.5.3.8	page 59
IDLE_INT	3.5.3.9	page 60
INTERRUPT ENABLE CONTROL	3.5.3.10	page 61
NC_BRANCH	3.5.3.11	page 62
NOP	3.5.3.12	page 63
READ_DEV_REGS	3.5.3.13	page 64
RETURN	3.5.3.14	page 65
RETURN_INT	3.5.3.15	page 66
TEST	3.5.3.16	page 67
WAIT	3.5.3.17	page 68
WRITE_DEST_ADDR	3.5.3.18	page 69
WRITE_DEV_REGS	3.5.3.19	page 70
WRITE_PFCTRL_REG	3.5.3.20	page 71
WRITE_REGISTER	3.5.3.21	page 72

3.5.3.1 BRANCH

The 3D engine transfers control within the display list. If it is not already in Display List mode, it enters Display List mode.

Table 3-46. BRANCH Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	00111b	–
STALL	26	0/1	Section 3.5.2.1
(Reserved)	25:22	0	–
OFFSET_ADDR	21:2	–	–
(Reserved)	1:0	0	–

Functional Description

The BRANCH instruction transfers control to the OFFSET_ADDR. This is an unconditional transfer. No return address is saved. If the 3D engine is not already in Display List mode, execution of this instruction causes it to enter Display List mode.

The OFFSET_ADDR field is added to PF_BASE_ADDR_3D to determine the virtual address in system memory of the target instruction.

3.5.3.2 CALL

The 3D engine transfers control within the display list. If it is not already in Display List mode, it enters Display List mode. The return address is saved.

Table 3-47. CALL Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	01010b	–
STALL	26	0/1	Section 3.5.2.1
(Reserved)	25:22	0	–
OFFSET_ADDR	21:2	–	–
(Reserved)	1:0	0	–

Functional Description

The CALL instruction transfers control to the OFFSET_ADDR. This is an unconditional transfer. The return address is saved. If the 3D engine is not already in Display List mode, execution of this instruction causes it to enter Display List mode.

The OFFSET_ADDR field is added to PF_BASE-ADDR_3D to determine the virtual address in system memory of the target instruction.

The CL-GD5464 stores a single return address. If a CALL is executed within a subroutine, the first return address is overwritten.

3.5.3.3 C_BRANCH

The 3D engine conditionally transfers control within the display list. If it is not already in Display List mode, it enters Display List mode.

Table 3-48. C_BRANCH Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	01000b	–
STALL	26	0/1	Section 3.5.2.1
(Reserved)	25:22	0	–
OFFSET_ADDR	21:2	–	–
(Reserved)	1:0	0	–

Functional Description

The C_BRANCH instruction transfers control to the OFFSET_ADDR if the condition flag is TRUE. No return address is saved. If the 3D engine is not already in Display List mode, execution of this instruction causes it to enter Display List mode.

The OFFSET_ADDR field is added to PF_BASE-ADDR_3D to determine the virtual address in system memory of the target instruction.

C_BRANCH is typically preceded with a TEST instruction that sets the condition flag to TRUE or FALSE.

3.5.3.4 CLEAR

The 3D Engine Parameter registers are set to '0'.

Table 3-49. CLEAR Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	01101b	–
STALL	26	0/1	Section 3.5.2.1
SUB_OPCODE	25:22	0101b	–
(Reserved)	21:0	0	–

Functional Description

The CLEAR instruction writes 0s to registers 0x4000 through 0x40FC.

3.5.3.5 DRAW_LINE

This instruction draws a line as specified in the instruction modifiers and the parameter list.

Table 3-50. DRAW_LINE Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	00001b	–
STALL	26	0/1	Section 3.5.2.1
INSTR_MODIFIER	25:12	–	Section 3.5.2.2
ADDR	11:6	(0)	–
COUNT	5:0	–	–

Functional Description

The DRAW_LINE instruction fetches and loads parameters into 3D Engine registers as specified in the COUNT field and the INSTR_MODIFIER field. It then draws a line.

The ADDR field specifies the first parameter to be loaded. This is essentially always programmed to zero so that X_3D is the first parameter.

The COUNT field specifies the number of parameters that follow the instruction. This can vary from six to over 20. Table 3-52 shows the order of the parameter and the ones that skipped if specified conditions are extant. The conditions are covered in Section 3.5.2.5.

Lines can be either X-major or Y-major (see Section 3.3.1). Table 3-51 summaries the registers the two increments must be programmed into. For Y-major lines, the Y-increment must always be positive (Y-major lines must be drawn from lower addresses to higher addresses, just as polygons are always drawn from the top down).

Table 3-51. Increments for DRAW_LINE

Line	X-Increment		Y-Increment	
X-Major	DX_MAIN_3D: 0x4014	± 1.0	DWIDTH2-3D: 0x4028	(signed value)
Y-Major	DX_MAIN_3D: 0x4014	(signed value)	DWIDTH2_3D: 0x4028	+ 1.0

Table 3-52. DRAW_LINE Parameter Order

Addr	Name	Mapped Color	Shading Disabled	Z_ON Off	Texture Mapping Off	Perspective Mapping Off	ALPHA_ LOAD Off
Main Drawing Parameters							
0x4000	X_3D						
0x4004	Y_3D						
0x4008	R_3D						
0x400C	G_3D	Skip					
0x4010	B_3D	Skip					
0x4014	DX_MAIN_3D						
0x4018	Y_COUNT_3D						
0x401C	WIDTH1_3D						
0x402C	DR_MAIN_3D		Skip				
0x4030	DG_MAIN_3D		Skip				
0x4034	DB_MAIN_3D		Skip				
0x4044	Z_3D			Skip			
0x4048	DZ_MAIN_3D			Skip			
Texture Map Parameters							
0x4050	V_3D				Skip		
0x4054	U_3D				Skip		
0x4058	DV_MAIN_3D				Skip		
0x405C	DU_MAIN_3D				Skip		
0x4068	D2V_MAIN_3D				Skip	Skip	
0x406C	D2U_MAIN_3D				Skip	Skip	
Alpha Blending Parameters							
0x40C0	A_3D						Skip
0x40C4	DA_MAIN_3D						Skip

3.5.3.6 DRAW_POINT

This instruction draws a point as specified in the instruction modifiers and the parameter list.

Table 3-53. DRAW_LINE Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	00000b	–
STALL	26	0/1	Section 3.5.2.1
INSTR_MODIFIER	25:12	–	Section 3.5.2.2
ADDR	11:6	(0)	–
COUNT	5:0	–	–

Functional Description

The DRAW_POINT instruction fetches and loads parameters into 3D Engine registers as specified in the COUNT field and the INSTR_MODIFIER field. It then draws a single point.

The ADDR field specifies the first parameter to be loaded. This is essentially always programmed to zero so that X_3D is the first parameter.

The COUNT field specifies the number of parameters that follow the instruction. This can vary from three to eight. Table 3-54 shows the order of the parameter and the ones that are skipped if specified conditions are extant. The conditions are covered in Section 3.5.2.5.

Table 3-54. DRAW_POINT Parameter Order

Addr	Name	Mapped Color	Z_ON Off	Texture Mapping Off	ALPHA_ LOAD Off
Main Drawing Parameters					
0x4000	X_3D				
0x4004	Y_3D				
0x4008	R_3D				
0x400C	G_3D	Skip			
0x4010	B_3D	Skip			
0x4044	Z_3D		Skip		
Texture Map Parameters					
0x4050	V_3D			Skip	
0x4054	U_3D			Skip	

3.5.3.7 DRAW_POLYGON

This instruction draws a polygon as specified in the instruction modifiers and the parameter list.

Table 3-55. DRAW_LINE Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	00010b	–
STALL	26	0/1	Section 3.5.2.1
INSTR_MODIFIER	25:12	–	Section 3.5.2.2
ADDR	11:6	(0)	–
COUNT	5:0	–	–

Functional Description

The DRAW_POLYGON instruction fetches and loads parameters into 3D Engine registers as specified in the COUNT field and the INSTR_MODIFIER field. It then draws a polygon.

The 3D engine interpolates the position of pixels unto an X, Y, Z grid located in either local memory or system memory. See Section 3.3. The interpolated positions and transparencies are dependent on the X, Y, and Z parameters supplied to the 3D engine for each polygon. Colors are also applied during this interpolation process from multiple sources:

- From the color ramp interpolators for Gouraud shading
- From constant values located in the color registers
- From texture maps located in local or system memory

As the color sources are applied to the 3D engine, operators are applied to control the lighting, fogging, and blending. Texture mapping operators are also applied to filter, blend, copy, and 'decal' texels as the traverse the CL-GD5464 3D engine data path.

The ADDR field specifies which is the first parameter to be loaded. This is essentially always programmed to zero so that X_3D is the first parameter.

The COUNT field specifies the number of parameters that follow the instruction. This can vary from seven to over 30. Table 3-56 shows the order of the parameter and the ones that are skipped if specified conditions are extant. The conditions are covered in Section 3.5.2.5.

Table 3-56. DRAW_POLYGON Parameter Order

Address	Name	Mapped Color	Shading Disabled	Z_ON Off	Initial Width Off	Texture Off	Perspective Off	ALPHA_ LOAD Off
Main Drawing Parameter								
0x4000	X_3D							
0x4004	Y_3D							
0x4008	R_3D							

Table 3-56. DRAW_POLYGON Parameter Order *(cont.)*

Address	Name	Mapped Color	Shading Disabled	Z_ON Off	Initial Width Off	Texture Off	Perspective Off	ALPHA_ LOAD Off
0x400C	G_3D	Skip						
0x4010	B_3D	Skip						
0x4014	DX_MAIN_3D							
0x4018	Y_COUNT_3D							
0x401C	WIDTH1_3D				Skip			
0x4020	WIDTH2_3D				Skip			
0x4024	DWIDTH1_3D							
0x4028	DWIDTH2_3D							
0x402C	DR_MAIN_3D		Skip					
0x4030	DG_MAIN_3D	Skip	Skip					
0x4034	DB_MAIN_3D	Skip	Skip					
0x4038	DR_ORTHO_3D		Skip					
0x403C	DG_ORTHO_3D		Skip					
0x4040	DB-ORTHO_3D		Skip					
0x4044	Z_3D			Skip				
0x4048	DZ_MAIN_3D			Skip				
0x404C	DZ_ORTHO_3D			Skip				
Texture Map Parameter								
0x4050	V_3D					Skip		
0x4054	U_3D					Skip		
0x4058	DV_MAIN_3D					Skip		
0x405C	DU_MAIN_3D					Skip		
0x4060	DV_ORTHO_3D					Skip		
0x4064	DU_ORTHO_3D					Skip		
0x4068	D2V_MAIN_3D					Skip	Skip	
0x406C	D2U_MAIN_3D					Skip	Skip	
0x4070	D2V_ORTHO_ 3D					Skip	Skip	
0x4074	D2U_ORTHO_ 3D					Skip	Skip	

Table 3-56. DRAW_POLYGON Parameter Order *(cont.)*

Address	Name	Mapped Color	Shading Disabled	Z_ON Off	Initial Width Off	Texture Off	Perspective Off	ALPHA_ LOAD Off
0x4078	DV_ORTHO_ ADD_3D					Skip	Skip	
0x407C	DU_ORTHO_ ADD_3D					Skip	Skip	
Alpha Blending Parameters								
0x40C0	A_3D							Skip
0x40C4	DA_MAIN_3D							Skip
0x4048	DA_ORTHO_3D							Skip

3.5.3.8 IDLE

The 3D engine enters the Idle state (Coprocessor mode).

Table 3-57. IDLE Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	01101b	–
STALL	26	0/1	Section 3.5.2.1
SUB_OPCODE	25:22	0000b	–
(Reserved)	21:0	0	–

Functional Description

The IDLE instruction places the CL-GD5464 into the idle state. It no longer processes the display list. A BRANCH, C_BRANCH, NC_BRANCH, or CALL can be used to re-initiate display list processing.

In the Idle state, all internal registers are exposed for direct control by the host.

3.5.3.9 IDLE_INT

The 3D engine enters the idle state (Coprocessor mode) and generates an interrupt to the host (if enabled).

Table 3-58. IDLE_INT Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	01101b	–
STALL	26	0/1	Section 3.5.2.1
SUB_OPCODE	25:22	0001b	–
(Reserved)	21:0	0	–

Functional Description

When executed in Display List mode, the IDLE_INT instruction ceases display list processing and generates an interrupt on INTA#. RETURN_INT can be used to resume display list processing.

IDLE_INT is typically used to signal the end of a display list, or that the engine has reached a point where guidance from the host is required.

3.5.3.10 Interrupt Enable Control

Interrupt enable bits are set or cleared according the SET_CLR field and the INT_MASK field.

Table 3-59. IDLE_INT Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	01111b	–
STALL	26	0/1	Section 3.5.2.1
SET_CLR	25	0/1	0 = Clear, 1 = Set
(Reserved)	24:5	0	–
INT_MASK	4:0	-	Table 3-60

Functional Description

The INTERRUPT instruction either sets or clears flip-flops that enable events to set bits in the PF_STATUS_3D register. The flip-flops are chosen according to the SET_CLR field and the INT_MASK field.

The SET_CLR field controls whether bits are to be set to '1' or cleared to '0'.

The INT_MASK field controls which interrupt enables are to be set or cleared. Since this field is bit-sensitive (rather than encoded), multiple bits can be set or cleared at once.

Table 3-60. INTERRUPT Instruction INT_MASK Field

INT_MASK	Interrupt Name	PF_STATUS_3D	Reset By
1 0000b	Z_COLLISION	Bit 4	Read of STATUS_3D (0x413C)
0 1000b	DISPLAY_BUFFER_SWITCH	Bit 3	Next CRT VSYNC
0 0100b	CRT_LINE_COMPARE	Bit 2	Next CRT VSYNC
0 0010b	CRT_EVSYNC	Bit 1	Next CRT VSYNC
0 0001b	CRT_VSYNC	Bit 0	Next CRT VSYNC

3.5.3.11 NC_BRANCH

The 3D engine conditionally transfers control within the display list. If it is not already in Display List mode, it enters Display List mode.

Table 3-61. NC_BRANCH Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	01001b	–
STALL	26	0/1	Section 3.5.2.1
(Reserved)	25:22	0	–
OFFSET_ADDR	21:2	–	–
(Reserved)	1:0	0	–

Functional Description

The NC_BRANCH instruction transfers control to the OFFSET_ADDR if the condition flag is FALSE. No return address is saved. If the 3D engine is not already in Display List mode, execution of this instruction causes it to enter Display List mode.

The OFFSET_ADDR field is added to PF_BASE-ADDR_3D to determine the virtual address in system memory of the next instruction.

NC_BRANCH is typically preceded with a TEST instruction that sets the condition flag to TRUE or FALSE.

3.5.3.12 NOP

The 3D engine advances the instruction address pointer.

Table 3-62. NOP Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	01101b	–
STALL	26	0/1	Section 3.5.2.1
SUB_OPCODE	25:22	0010b	–
(Reserved)	21:0	0	–

Functional Description

The IDLE instruction does nothing except to advance the instruction pointer and possibly execute a STALL. This can be used to ensure the 3D engine has finished drawing and the pre-fetch pipeline is clear.

3.5.3.13 READ_DEV_REGS

The indicated register is read. The results are stored in system memory at the address specified in PF_DEST_ADDR_3D.

Table 3-63. READ_DEV_REGS Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	00110b	–
STALL	26	0/1	Section 3.5.2.1
MODULE_SELECT	25:21	–	Table 3-43
(Reserved)	20:17	0	–
ADDR	16:6	–	–
COUNT	5:0	1	–

Functional Description

The READ_DEV_REGS instruction fetches the contents of the specified register and places the contents into the location in system memory specified in PF_DEST_ADDR_3D. PF_DEST_ADDR_3D is loaded with the execution of a WRITE_DEST_ADDR instruction.

This instruction is used to write the CL-GD5464 status to system memory while processing a display list. This instruction waits for the 3D engine to be idle before it is executed. This allows the instruction stream processing to communicate its status to the system software, thus providing a means of synchronization between 3D operations and 2D operations.

The MODULE_SELECT field indicates which module in the CL-GD5464 contains the register to be read. This field in defined in Table 3-43.

The ADDR field indicates the register within the module is to be read.

COUNT must be programmed to ‘1’.

3.5.3.14 RETURN

The 3D engine returns from a subroutine.

Table 3-64. RETURN Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	01101b	–
STALL	26	0/1	Section 3.5.2.1
SUB_OPCODE	25:22	0011b	–
(Reserved)	21:0	0	–

Functional Description

The RETURN instruction restores the state information saved by a CALL instruction and resumes instruction stream processing at the instruction following the CALL.

The CL-GD5464 supports only a single-level subroutine.

3.5.3.15 RETURN_INT

The 3D engine restores the state information saved by an INT and resumes display list processing at the location following the INT.

Table 3-65. RETURN_INT Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	01101b	–
STALL	26	0/1	Section 3.5.2.1
SUB_OPCODE	25:22	0100b	–
EI	21	0/1	–
DA	20	0/1	–
CF	19	0/1	–
(Reserved)	18:0	0	–

Functional Description

The RETURN_INT instruction restores the instruction state saved by an INT and resumes display list processing.

Table 3-66 summarizes the control bits in 21:19.

Table 3-66. State Information Restoration Control for RETURN_INT

RETURN_INT Field	Field	Specific Field(s) Effected
EI (bit 21)	Interrupt enables	
DA (bit 20)	Destination address	
CF (bit 19)	Condition flag	

3.5.3.16 TEST

The 3D engine tests one or more events and sets the condition flag based on the outcome.

Table 3-67. TEST Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	10000	–
STALL	26	0/1	Section 3.5.2.1
AND_OR	25	0/1	–
NOT	24	0/1	–
(Reserved)	23:11	0	–
EVENT_MASK	10:0	-	Table 3-42

Functional Description

The TEST instruction tests one or more events as specified in the EVENT_MASK and sets or resets the condition flag based on the outcome.

The AND_OR bit and NOT bit control how the events are combined. This is summarized in Table 3-68.

Table 3-68. TEST Instruction Control Bits

AND_OR	NOT	Number of ‘1’ bits in EVENT_MASK	Instruction Name	C-Flag is set if
X	0	1	TEST	Single event is TRUE
1	0	2 or more	TEST_AND	All events are TRUE
0	0	2 or more	TEST_OR	Any event is TRUE
X	1	1	NTEST	Single event is FALSE
1	1	2 or more	NTEST_AND	All events are FALSE
0	1	2 or more	NTEST_OR	Any event is FALSE

3.5.3.17 WAIT

The 3D engine tests one or more events and waits before continuing with the display list.

Table 3-69. WAIT Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	01110b	–
STALL	26	0/1	Section 3.5.2.1
AND_OR	25	0/1	–
NOT	24	0/1	–
(Reserved)	23:11	0	–
EVENT_MASK	10:0	–	Table 3-42

Functional Description

The WAIT instruction tests one or more events as specified in the EVENT_MASK and waits before continuing with the display list. The WAIT instruction does not change the condition flag.

The AND_OR bit and NOT bit control how the events are combined. This is summarized in Table 3-70.

Table 3-70. WAIT Instruction Control Bits

AND_OR	NOT	Number of ‘1’ bits in EVENT_MASK	Instruction Name	Waits for
X	0	1	TEST	Single event to be TRUE
1	0	2 or more	TEST_AND	All events to be TRUE
0	0	2 or more	TEST_OR	Any event to be TRUE
X	1	1	NTEST	Single event to be FALSE
1	1	2 or more	NTEST_AND	All events to be FALSE
0	1	2 or more	NTEST_OR	Any event to be FALSE

3.5.3.18 WRITE_DEST_ADDR

The 3D engine writes the OFFSET_ADDR value to PF_DEST_ADDR_3D.

Table 3-71. WRITE_DEST_ADDR Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	01011b	–
STALL	26	0/1	Section 3.5.2.1
(Reserved)	25:22	0	–
OFFSET_ADDR	21:2	–	–
(Reserved)	1	0	–
INCREMENT	0	0/1	–

Functional Description

The WRITE_REGISTER instruction transfers the OFFSET_ADDR field to PF_DEST_ADDR_3D. Bit 0 of the instruction is transferred to bit 0 (the INCREMENT bit) of PF_DEST_ADDR_3D. The PF_DEST_ADDR_3D register specifies the offset for READ_DEV_REG.

3.5.3.19 WRITE_DEV_REGS

The indicated registers are written with words following the instruction word.

Table 3-72. WRITE_DEV_REGS Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	00101b	–
STALL	26	0/1	Section 3.5.2.1
MODULE_SELECT	25:21	–	Table 3-43
BYTE_ENABLES	20:17	0	–
ADDR	16:6	–	–
COUNT	5:0	1	–

Functional Description

The WRITE_DEV_REGS instruction fetches the word(s) following the instruction and writes them into the indicated register(s).

This instruction waits for the 3D engine to be idle before it is executed. This allows the instruction stream processing to communicate its status to the system software, thus providing a means of synchronization between 3D operations and 2D operations.

The MODULE_SELECT field indicates the module in the CL-GD5464 containing the register to be written. This field in defined in Table 3-43.

The BYTE_ENABLES field indicates which bytes are to be written. Bit 20 corresponds to the most-significant byte; bit 17 corresponds to the least-significant byte. Typically, this field is all '1's except for some single-register operations.

The ADDR field indicates the first (or only) register within the module to be written.

The COUNT field indicates the number of registers to be written.

The WRITE_DEV_REGS is used to access other devices with the CL-GD5464 while processing a display list. For example, this allows texture map palettes to be loaded into the TLUT after a texture-mapped polygon is completed.

3.5.3.20 WRITE_PFCTRL_REG

The 3D engine writes the PF_CTL_DATA field to either the PF_CTL_3D register or the PF_FB_SEL_3D register.

Table 3-73. WRITE_PFCTRL_REG Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	10001b	–
STALL	26	0/1	Section 3.5.2.1
RSEL	25	0/1	–
(Reserved)	24:16	0	–
PF_CTL_DATA	15:0	–	–

Functional Description

The WRITE_PFCTRL_REG instruction transfers the PF_CTL_DATA field to the least-significant 16 bits of either PF_CTL_3D or PF_FB_SEG_3D.

The PF_CTL_3D register is not otherwise accessible from display list processing. Care must be exercised if this register is changed during engine execution. Place a NOP with STALL (that waits until the engine is idle) immediately in front of this instruction in the display list.

RSEL specifies which of the two object registers is to be loaded, as shown in Table 3-74.

Table 3-74. WRITE_PFCTRL_REG Instruction RSEL Field

RSEL	Destination Register	Address	Bits Used
0	PF_CRL_3D	0x4404	15:0
1	PF_FB_SEG_3D	0x440C	10:0

3.5.3.21 WRITE_REGISTER

The 3D engine writes one or more contiguous registers. This instruction can be used in coprocessor mode as well as Display List mode.

Table 3-75. WRITE_REGISTER Instruction Format

Field	Bits	Value	Reference
OP_CODE	31:27	00011b	–
STALL	26	0/1	Section 3.5.2.1
(Reserved)	25:13	0	–
C	12	0/1	–
ADDR	11:6	–	–
COUNT	5:0	–	–

Functional Description

The WRITE_REGISTER instruction transfers the next COUNT DWORDs in the display to 3D registers beginning with the register specified in the C field and ADDR field. This instruction can take advantage of the PCI burst write sequence.

If the COUNT field is ‘0’, no registers are written.

The C bit indicates whether the ADDR field is an offset from 0x4000h or 0x4100. See Table 3-76.

Table 3-76. WRITE_REGISTER Instruction C Bit

C (Bit 12)	Base Address	Registers
0	0x4000	Drawing
1	0x4100	Control

3.6 3D Register Header Files

The following four header files are used by the programming examples for the CL-GD5464. Included are register definitions, instruction opcodes, structure definitions, and other generally useful information. The # defines that could be used as a basic for programming the CL-GD5464. The files ‘l3types.h’ and ‘modemon.h’ are support header files, included for completeness.

3.6.1 trm.h

```
/*********************************************************************
*
*   Module:     trm.h
*
*   Revision:   1.00
*
*   Date:       August 30, 1996
*
*   Author:     Cirrus Logic Austin Design Center
*
**********************************************************************
*
*   Module Description:
*
*      Register, instruction, instruction modifiers and other
*      defines used by this library
*
**********************************************************************
*
*   Changes:
*
*    DATE     REVISION  DESCRIPTION                       AUTHOR
*  --------   --------  ----------------------------------------------
**********************************************************************/

#ifndef _TRM_H_
#define _TRM_H_

#include <string.h>                 // memset()
#include "l3types.h"                // DWORD, etc
#include "l3struct.h"               // LL_Texture, etc

// Set a register to a given value
//
#define SETREG(reg, value)                                        \
    (*(DWORD *)(LL_State.pRegs + reg) = (DWORD) value)

// Clear a range of registers
//
#define CLEAR_RANGE(StartReg, EndReg)                             \
   memset( (void *)(LL_State.pRegs + (StartReg)), 0, ((EndReg) - (StartReg)+1) )

// Laguna 3D instruction set

#define  DRAW_POINT                     0x00000000
#define  DRAW_LINE                      0x08000000
#define  DRAW_POLY                      0x10000000
#define  WRITE_REGISTER                 0x18000000
#define  READ_REGISTER                  0x20000000
#define  WRITE_DEV_REGS                 0x28000000
#define  READ_DEV_REGS                  0x30000000
#define  BRANCH                         0x38000000
#define  C_BRANCH                       0x40000000
#define  NC_BRANCH                      0x48000000
#define  CALL                           0x50000000
#define  WRITE_DEST_ADDR                0x58000000
#define  INSTR_EXT                      0x68000000
#define  WAIT                           0x72000000
#define  WAIT_AND                       0x72000000
#define  NWAIT_AND                      0x73000000
#define  WAIT_OR                        0x70000000
#define  NWAIT_OR                       0x71000000
#define  CLEAR_INT                      0x78000000
#define  SET_INT                        0x7A000000
#define  TEST                           0x80000000
#define  TEST_AND                       0x82000000
#define  NTEST_AND                      0x83000000
#define  TEST_OR                        0x80000000
#define  NTEST_OR                       0x81000000
#define  WRITE_PREFETCH_CONTROL         0x88000000

// instruction modifier set for drawing instructions

#define  STALL                          0x04000000
#define  GOURAUD                        0x00001000
#define  Z_ON                           0x00002000
#define  COLOR_OFF                      0x00008000
```

```
#define  TEXTURE_LINEAR                0x00020000
#define  TEXTURE_PERSPECTIVE           0x00030000
#define  LIGHTING                      0x00040000
#define  STIPPLE                       0x00080000
#define  PATTERN                       0x00100000
#define  DITHER                        0x00200000
#define  ALPHA                         0x00400000
#define  FETCH_COLOR                   0x00800000
#define  WARP_MODE                     0x01000000
#define  MODIFIER_EXPANSION            0x02000000

// instruction extension opcodes

#define  IDLE                          0x00000000
#define  IDLE_INT                      0x00400000
#define  NOP                           0x00800000
#define  RETURN                        0x00c00000
#define  INT_RETURN                    0x01000000
#define  CLEAR                         0x01400000

// core 3D registers - non byte-swapping apperture beginning at 0x4000

#define X_3D                           0x4000
#define Y_3D                           0x4004
#define R_3D                           0x4008
#define G_3D                           0x400c
#define B_3D                           0x4010
#define DX_MAIN_3D                     0x4014
#define Y_COUNT_3D                     0x4018
#define WIDTH1_3D                      0x401c
#define WIDTH2_3D                      0x4020
#define DWIDTH1_3D                     0x4024
#define DWIDTH2_3D                     0x4028
#define DR_MAIN_3D                     0x402c
#define DG_MAIN_3D                     0x4030
#define DB_MAIN_3D                     0x4034
#define DR_ORTHO_3D                    0x4038
#define DG_ORTHO_3D                    0x403c
#define DB_ORTHO_3D                    0x4040
#define Z_3D                           0x4044
#define DZ_MAIN_3D                     0x4048
#define DZ_ORTHO_3D                    0x404c
```

```
#define V_3D                        0x4050
#define U_3D                        0x4054
#define DV_MAIN_3D                  0x4058
#define DU_MAIN_3D                  0x405c
#define DV_ORTHO_3D                 0x4060
#define DU_ORTHO_3D                 0x4064
#define D2V_MAIN_3D                 0x4068
#define D2U_MAIN_3D                 0x406c
#define D2V_ORTHO_3D                0x4070
#define D2U_ORTHO_3D                0x4074
#define DV_ORTHO_ADD_3D             0x4078
#define DU_ORTHO_ADD_3D             0x407c

#define A_3D                        0x40C0
#define DA_MAIN_3D                  0x40C4
#define DA_ORTHO_3D                 0x40C8

#define OPCODE_3D                   0x40fc

// 3D Control registers

#define CONTROL_MASK_3D             0x4100
#define CONTROL0_3D                 0x4104
#define COLOR_MIN_BOUNDS_3D         0x4108
#define COLOR_MAX_BOUNDS_3D         0x410c
#define CONTROL1_3D                 0x4110
#define BASE0_ADDR_3D               0x4114
#define BASE1_ADDR_3D               0x4118

#define TX_CTL0_3D                  0x4120
#define TX_XYBASE_3D                0x4124
#define TX_CTL1_3D                  0x4128
#define TX_CTL2_3D                  0x412C
#define COLOR0_3D                   0x4130
#define COLOR1_3D                   0x4134
#define Z_COLLIDE_3D                0x4138
#define STATUS0_3D                  0x413C
#define PATTERN_RAM_0_3D            0x4140
#define PATTERN_RAM_1_3D            0x4144
#define PATTERN_RAM_2_3D            0x4148
#define PATTERN_RAM_3_3D            0x414c
#define PATTERN_RAM_4_3D            0x4150
```

```
#define PATTERN_RAM_5_3D                0x4154
#define PATTERN_RAM_6_3D                0x4158
#define PATTERN_RAM_7_3D                0x415c
#define X_CLIP_3D                       0x4160
#define Y_CLIP_3D                       0x4164
#define TEX_SRAM_CTRL_3D                0x4168

// host_XY unit registers

#define HXY_BASE0_ADDRESS_PTR_3D        0x4200
#define HXY_BASE0_START_XY_3D           0x4204
#define HXY_BASE0_EXTENT_XY_3D          0x4208
#define HXY_BASE1_ADDRESS_PTR_3D        0x4210
#define HXY_BASE1_OFFSET0_3D            0x4214
#define HXY_BASE1_OFFSET1_3D            0x4218
#define HXY_BASE1_LENGTH_3D             0x421C
#define HXY_BASE2_ADDRESS_PTR_3D        0x4220
#define HXY_HOST_CTRL_3D                0x4240
#define MAILBOX0_3D                     0x4260
#define MAILBOX1_3D                     0x4264
#define MAILBOX2_3D                     0x4268
#define MAILBOX3_3D                     0x426C

// 3D prefetch unit registers

#define PF_BASE_ADDR_3D                 0x4400
#define PF_CTRL_3D                      0x4404
#define PF_FB_SEG_3D                    0x440C
#define PF_DEST_ADDR_3D                 0x4408
#define PF_INST_ADDR_3D                 0x4420
#define PF_STATUS_3D                    0x4424
#define HOST_MASTER_CTRL_3D             0x4440
#define PF_INST_3D                      0x4480

//*******************************************************************
//
// Library initialization defines:
//
// The following are examples of various useful defines that pertain
// to Laguna 3d. They are not necessarily used in this programming
// example
//
```

```
//*********************************************************************

// Defines for pixel modes (Control0 register)

#define PIXEL_MODE_INDEXED  0
#define PIXEL_MODE_332      1
#define PIXEL_MODE_565      2
#define PIXEL_MODE_555      3
#define PIXEL_MODE_A888     4
#define PIXEL_MODE_Z888     5

// Z Compare modes

#define LL_Z_WRITE_GREATER_EQUAL   0x00000000  // True if new >= old
#define LL_Z_WRITE_GREATER         0x00000001  // True if new >  old
#define LL_Z_WRITE_LESS_EQUAL      0x00000002  // True if new <= old
#define LL_Z_WRITE_LESS            0x00000003  // True if new <  old
#define LL_Z_WRITE_NOT_EQUAL       0x00000004  // True if new <> old
#define LL_Z_WRITE_EQUAL           0x00000005  // True if new =  old

// Functional Z modes

#define LL_Z_MODE_NORMAL           0x00000000  // Normal operation
#define LL_Z_MODE_MASK             0x00000001  // Z not written
#define LL_Z_MODE_ALWAYS           0x00000002  // Z, color always wrt
#define LL_Z_MODE_ONLY             0x00000003  // Color not written
#define LL_Z_MODE_HIT              0x00000004  // collision dtct only

// Color compare controls

#define LL_COLOR_SATURATE_ENABLE   0x00000040  // for indexed mode
#define LL_COLOR_SATURATE_DISABLE  0x00000000  // (default)
#define LL_COLOR_COMPARE_INCLUSIVE 0x00000400  // tc modes
#define LL_COLOR_COMPARE_EXCLUSIVE 0x00000000  // tc modes (default)
#define LL_COLOR_COMPARE_BLUE      0x00000200  // blue (default off)
#define LL_COLOR_COMPARE_GREEN     0x00000100  // green (default off)
#define LL_COLOR_COMPARE_RED       0x00000080  // red (default off)

//*********************************************************************
//
//  Lighting source: Selects the value for the lighting multiplier
//      - interpolated light from the polygon engine
//          load lighting values as r,g,b components
//          also LL_GOURAUD must be set in the flags
//          this mode uses Polyengine color registers
//      - interpolated light from the alpha interpolator
//          load lighting values as alpha components
//          this mode uses LA-interpolators
//      - constant light from the COLOR1 register
//
//**********************************************************************

#define LL_LIGHTING_INTERP_RGB     0x00000000  // Using poly engine
#define LL_LIGHTING_INTERP_ALPHA   0x00000001  // Using LA interp.
#define LL_LIGHTING_CONST          0x00000002  // Constant light

//**********************************************************************
//
//  Alpha mode:  Magnitude of alpha blending will be taken from
//      - constant alpha, use LL_SetConstantAlpha(src/new,dest/old)
//          this mode uses LA-interpolators
//      - interpolated, variable alpha from LA-interpolators
//          this mode also uses LA-interpolators
//      - alpha field from the frame buffer
//
//**********************************************************************
#define LL_ALPHA_CONST             0x00000000  // Constant alpha
#define LL_ALPHA_INTERP            0x00000002  // Using LA interp.
#define LL_ALPHA_FRAME             0x00000003  // Using frame values

//**********************************************************************
//
//  Alpha destination: Selects where the second color input to the
//      alpha multiplier comes from
//      - color from the frame buffer ("normal" alpha blending)
//      - constant color (also called fog) from COLOR0 register
//      - interpolated, shaded color from the polygon engine (also fog)
//          also LL_GOURAUD must be set in the flags
//          this mode uses Polyengine color registers
//
//  Fog: Use aliases LL_FOG_CONST and LL_FOG_INTERP to avoid fetching
//      colors from the frame and to set the fog color.
//
```

```
//*******************************************************************/
#define LL_ALPHA_DEST_FRAME        0x00000000   // Using frame color
#define LL_ALPHA_DEST_CONST        0x00000001   // Constant color
#define LL_ALPHA_DEST_INTERP       0x00000002   // Using poly engine

/*********************************************************************
*
*   Buffer identification numbers and Z stride info.
*
*   Used with LL_InitBuffers()
*
*********************************************************************/
#define LL_ID_BUFFER_A      0       /* ID of the primary buffer      */
#define LL_ID_BUFFER_B      1       /* ID of the secondary buffer    */
#define LL_ID_BUFFER_Z      2       /* ID of the Z buffer in RDRAM   */

// function prototypes internal to this library

int  LL_Init();
int  LL_InitBuffers();
int  LL_Terminate();
LL_Wait();
void LL_print_state();
void LL_print_regs();

LL_SetDisplayBuffer(TBuffer  *pBuf);            // useful for debug

// display list programming examples

void dl_point_set0(DWORD *pdwNext);             // points
void dl_point_set1(DWORD *pdwNext);
void dl_point_set2(DWORD *pdwNext);

void dl_line_set0(DWORD *pdwNext);              // lines
void dl_line_set1(DWORD *pdwNext);

void dl_poly_set0(DWORD *pdwNext);              // polys
void dl_poly_set1(DWORD *pdwNext);
void dl_poly_set2(DWORD *pdwNext);
void dl_poly_set3(DWORD *pdwNext);
void dl_poly_set3(DWORD *pdwNext);
```

```
// functions external to this library

extern DWORD  AllocSystemMemory( DWORD dwSize );
extern void   FreeSystemMemory( DWORD hHandle );
extern DWORD  GetLinearAddress( DWORD hHandle );
extern DWORD  GetPhysicalAddress( DWORD hHandle );
extern DWORD *GetRegisterApperture();
extern BYTE  *GetLagunaApperture( int base );
extern int    GetPrivateProfileString(char *, char *, char *, char *, int, char
*);

#endif // _TRM_H_
```

3.6.2 l3struct.h

```
/**********************************************************************
*
*   Module:     l3struct.h
*
*   Revision:   1.00
*
*   Date:       August 30, 1996
*
*   Author:     Cirrus Logic Austin Design Center
*
***********************************************************************
*
*   Module Description:
*
*       Various structures that support this small library
*
***********************************************************************
*
*   Changes:
*
*    DATE     REVISION  DESCRIPTION                       AUTHOR
*  --------   --------  ---------------------------------------------
***********************************************************************/

#ifndef _L3STRUCT_H_
#define _L3STRUCT_H_

/**********************************************************************
*
*   LL_Rect structure defines a general rectangular region
*
***********************************************************************/
typedef struct
{
    DWORD left;                     // x1
    DWORD top;                      // y1
    DWORD right;                    // x2
    DWORD bottom;                   // y2

} LL_Rect;
```

```
/*********************************************************************
*
*   LL_Color structure defines color by its components or index
*
*********************************************************************/
typedef struct
{
    union
    {
        struct                      // If in true color mode,
        {
            BYTE r;                 // Red component
            BYTE g;                 // Green component
            BYTE b;                 // Blue component
        };
        BYTE index;                 // Index if in 8bpp indexed mode
    };

} LL_Color;


/*********************************************************************
*
*   LL_Pattern structure holds the pattern to be stored in the
*   PATTERN_RAM registers.  These values are used for pattern,
*   dither or stipple (only one at a time).
*
*********************************************************************/
typedef struct                      // pattern holding structure
{
    DWORD pat[ 8 ];                 // 8 word pattern

} LL_Pattern;


/*********************************************************************
*
*   Buffer information structure (buffers A, B, Z, ...)
*
*********************************************************************/
typedef struct
```

```
{
    // for this example, all buffers are in RDRAM

    DWORD dwFlags;                  // Buffer flags
    DWORD dwAddress;                // Buffer start byte address (abs linear)
    DWORD dwPitchBytes;             // Pitch of a buffer in bytes
    LL_Rect Extent;                 // Buffer location offsets (video)

    // these fields are not used, but are examples of possible fields
    // for buffers in system memory

    DWORD dwPhyAdr;                 // Buffer physical address (system)
    DWORD dwPitchCode;              // Pitch code of a buffer (system)
    DWORD hMem;                     // Internal memory handle (system)
} TBuffer;


/**********************************************************************
*
*   LL_Texture structure defines a texture map
*
**********************************************************************/
typedef struct
{
    DWORD * dwAddress;              // Pointer to texture storage location
    LL_Color * ColPalette;          // Pointer to palette if indexed
    BYTE  bMem;                     // Index to the texture memory block
    DWORD dwFlags;                  // Flags for the texture
    WORD  wWidth;                   // Texture X dimension in texels
    WORD  wHeight;                  // Texture Y dimension in texels
    BYTE  bSizeMask;                // Encoded size 0=16,... Y[7:4],X[3:0]
    BYTE  bType;                    // Texture type
    BYTE  fIndexed;                 // True for indexed textures
    BYTE  bLookupOffset;            // Palette lookup offset (indexed only)
    BYTE  bBpp;                     // Bits per pixel
    BYTE  bID;                      // Texture ID for the placement module
    WORD  wXloc;                    // X offset location in bytes
    WORD  wYloc;                    // Y offset location in lines
    DWORD dwUsed;                   // Usage count (for priorities)

} LL_Texture;
```

```
/*********************************************************************
*
*   TDisplayList structure defines a display list.
*
*********************************************************************/
typedef struct
{
    // pdwNext points to the next available location within this
    // display list to fill in the Laguna instruction.
    // It is used for parametarization routines that postincrement
    // this variable.
    //
    DWORD *pdwNext;

    // Memory handle for this display list as optained from the
    // internal memory allocation function
    //
    DWORD hMem;

    // Linear address of the display list
    //
    DWORD *pdwLinPtr;

    // Physical address for a display list is next; it can also
    // be the address to the page table.  This address has the
    // appropriate format to be stored in the BASE* class registers
    //
    DWORD dwPhyPtr;

    // The length of a display list in bytes
    //
    DWORD dwLen;

    // Safety margin for building the display list
    //
    DWORD dwMargin;

} TDisplayList;


/*********************************************************************
```

```
*
*   Control0_3d register bitfields
*
*********************************************************************/
typedef struct
{
    DWORD Pixel_Mode            : 3;  // Color frame buffer drawing mode
    DWORD Res1                  : 1;  // Reserved
    DWORD Pixel_Mask_Enable     : 1;  // Enables pixel masking
    DWORD Pixel_Mask_Polarity   : 1;  // Polarity of the pixel masks
    DWORD Color_Saturate_En     : 1;  // Enables saturation in indexed mode
    DWORD Red_Color_Compare_En  : 1;  // Enables compare to bounds for red
    DWORD Green_Color_Compare_En: 1;  // Enables compare to bounds for green
    DWORD Blue_Color_Compare_En : 1;  // Enables compare to bounds for blue
    DWORD Color_Compare_Mode    : 1;  // Mask inclusive/exclusive to bounds
    DWORD Alpha_Mode            : 2;  // Selects alpha blending mode
    DWORD Alpha_Dest_Color_Sel  : 2;  // Selects the DEST_RGB input to alpha
    DWORD Alpha_Blending_Enable : 1;  // Enables alpha blending
    DWORD Z_Stride_Control      : 1;  // 16/8 bit Z depth
    DWORD Res2                  : 3;  // Reserved
    DWORD Z_Compare_Mode        : 4;  // Different Z compare function
    DWORD Z_Collision_Detect_En : 1;  // Enables Z collision detection
    DWORD Light_Src_Sel         : 2;  // Selects the lighting source input
    DWORD Res3                  : 1;  // Reserved
    DWORD Z_Mode                : 3;  // Controls Z and color update method
    DWORD Res4                  : 1;  // Reserved

} TControl0Reg;


/********************************************************************
*
*   TSystem structure defines possible cached state information
*
*********************************************************************/
typedef struct
{
    // Laguna 3D registers software cache.  These registers are
    // cached here in order to avoid unnecessary setup with the
    // possibly same values.  Every time when one of these registers
    // need to be set, the content of the cache is compared and the
    // register is set only if it differs from the new value.
```

```
//
DWORD dwDA_MAIN;                 // Current value of DA_MAIN_3D reg
DWORD dwDA_ORTHO;                // Current value of DA_ORTHO_3D reg

union {
    TControl0Reg Control0;       // Control 0 register shadow
    DWORD dwControl0;
};

DWORD dwColor_Min_Bounds;        // Color compare min bounds
DWORD dwColor_Max_Bounds;        // Color compare max bounds
DWORD dwBase0;
DWORD dwBase1;
DWORD dwTxControl0;
DWORD dwTxXYBase;
DWORD dwColor0;                  // Current value of COLOR_REG0_3D reg
DWORD dwColor1;                  // Current value of COLOR_REG1_3D reg
DWORD dwHXY_Base1_Address_Ptr;   // State of the host access base reg 1
DWORD dwHXY_Base1_Offset0;       // State of the Offset base 1 reg
DWORD dwHXYHostControl;
DWORD dwFetchColor;              // Equals FETCH_COLOR for color compares
DWORD dwAlphaConstSource;        // Constant source alpha (9:16)
DWORD dwAlphaConstDest;          // Constant destination alpha (9:16)

// Buffer management
TBuffer BufRender;               // current rendering buffer structure
TBuffer BufTextures;             // backup buffer (textures) structure
TBuffer BufZ;                    // Z buffer structure

// Display lists management
TDisplayList DL;                 // Current display list to build

// Non-register state information

DWORD dwFlags;                   // pixel depth flags
BYTE  *pRegs;                    // 5464 Register apperture
BYTE  *pFrame;                   // 5464 Frame apperture
WORD  wPCI_Interrupt;            // Laguna PCI interrupt number
WORD  wPCI_Slot;                 // Laguna PCI slot

DWORD dwVRAM;                    // Amount of video RAM in bytes
DWORD pitch;                     // Screen pitch in bytes
```

```
    WORD  wHoriz;                   // Display width in pixels
    WORD  wVert;                    // Display height in pixels
    WORD  wBpp;                     // Pixel depth (8, 16,...)

    BYTE  fSingleRead;              // Single read flag as opposed to burst
    BYTE  fSingleWrite;             // Single write flag as opposed to burst
    DWORD dwLatencyTimer;           // Latency timer

    BYTE  fIndexed;                 // True if indexed graphics mode is used

} TSystem;


/**********************************************************************
*
*   LL_DeviceState structure defines possible cached state information
*   that might be exported to a client of a graphics library
*
**********************************************************************/
typedef struct
{
    /* These three fields can be set before calling the LL_InitLib function  */

    DWORD dwFlags;                 /* Init flags                            */
    DWORD dwDisplayListLen;        /* Size of the display lists (in bytes)  */
    DWORD dwSystemTexturesLen;     /* Size of the system textures (in bytes)*/

    /* These variables can be used by the software */

    DWORD *pRegs;                  /* Laguna regs, ptr to mem mappped I/O   */
    BYTE  *pFrame;                 /* Frame, pointer to the a frame buffer  */
    DWORD dwVRAM;                  /* Video memory on the card (in bytes)   */
    WORD  wHoriz;                  /* Current horizontal resolution         */
    WORD  wVert;                   /* Current vertical resolution           */
    WORD  wBpp;                    /* Current pixel depth                   */

} LL_DeviceState;

#endif // _L3STRUCT_H_
```

3.6.3 l3types.h

```
/**********************************************************************
*
*   Module:     l3types.h          Generic Type Header Module
*
*   Revision:   1.00
*
*   Date:       April 14, 1994
*
*   Author:     Cirrus Logic Austin Design Center
*
***********************************************************************
*
*   Module Description:
*
*       This module contains generic type declarations
*
***********************************************************************
*
*   Changes:
*
*    DATE     REVISION  DESCRIPTION                           AUTHOR
*  --------   --------  --------------------------------------------
*  04/14/95     1.00    Original                            Randy Spurlock
*  09/26/95     1.01    Add few new defines                 Goran Devic
*  02/15/96     1.02    Fit to L3d library format           Goran Devic
*  --------   --------  --------------------------------------------
***********************************************************************/

#ifndef _L3TYPES_H_
#define _L3TYPES_H_

/**********************************************************************
*   Type Definitions
***********************************************************************/

typedef int BOOL;                  /* Define a boolean as an integer  */
typedef unsigned char BYTE;        /* Define a byte data type         */
typedef unsigned short int WORD;  /* Define a word data type         */
typedef unsigned long DWORD;       /* Define a double word data type  */

#endif // _L3TYPES_H_
```

3.6.4 modemon.h

```
/*****************************************************************************
*
*
*   Module:     modemon.h       Mode/Monitor Functions Header Module
*
*   Revision:   1.00
*
*   Date:       April 8, 1994
*
*   Author:     Cirrus Logic Austin Design Center
*
******************************************************************************
*
*
*   Module Description:
*
*       This module contains the type declarations and function
*   prototypes for the mode/monitor functions.
*
******************************************************************************
*
*
*   Changes:
*
*    DATE     REVISION  DESCRIPTION                             AUTHOR
*  --------   --------  --------------------------------------------------
******************************************************************************
/

#ifndef _MODEMON_H_
#define _MODEMON_H_

/*****************************************************************************
*
*   Type Definitions and Structures
******************************************************************************
/
typedef struct tagRange                /* Range structure                   */
{
    union tagMinimum                   /* Minimum value for the range       */
    {
        int     nMin;
```

```
            long    lMin;
            float   fMin;
        } Minimum;
        union tagMaximum                    /* Maximum value for the range       */
        {
            int     nMax;
            long    lMax;
            float   fMax;
        } Maximum;
    } Range;

    typedef struct tagMonListHeader         /* Monitor list header structure     */
    {
        int         nMonitor;               /* Number of monitors in the list    */
    } MonListHeader;

    typedef struct tagMonListEntry          /* Monitor list entry structure      */
    {
        char        *pszName;               /* Pointer to monitor name string    */
        char        *pszDesc;               /* Pointer to monitor description    */
    } MonListEntry;

    typedef struct tagMonList               /* Monitor list structure            */
    {
        MonListHeader       MonHeader;      /* Monitor list header               */
        MonListEntry        MonEntry[1];    /* Start of the monitor list entries */
    } MonList;

    typedef struct tagMonInfoHeader         /* Monitor info. header structure    */
    {
        int         nMode;                  /* Number of monitor modes in list   */
    } MonInfoHeader;

    typedef struct tagMonInfoEntry          /* Monitor info. entry structure     */
    {
        char        *pszName;               /* Pointer to monitor mode name      */
        Range       rHoriz;                 /* Horizontal range values           */
        Range       rVert;                  /* Vertical range values             */
        int         nSync;                  /* Horiz./Vert. sync. polarities     */
        int         nResX;                  /* Maximum suggested X resolution    */
        int         nResY;                  /* Maximum suggested Y resolution    */
    } MonInfoEntry;
```

```
typedef struct tagMonInfo              /* Monitor information structure     */
{
    MonInfoHeader       MonHeader;      /* Monitor information header        */
    MonInfoEntry        MonEntry[1];    /* Start of the monitor entries      */
} MonInfo;

typedef struct tagModeInfoEntry        /* Mode information entry structure  */
{
    char        *pszName;              /* Pointer to mode name string       */
    float       fHsync;                /* Horizontal sync. frequency value  */
    float       fVsync;                /* Vertical sync. frequency value    */
    int         nResX;                 /* Horizontal (X) resolution value   */
    int         nResY;                 /* Vertical (Y) resolution value     */
    int         nBPP;                  /* Pixel depth (Bits/Pixel)          */
    int         nMemory;               /* Memory size (Kbytes)              */
    int         nPitch;                /* Pitch value (Bytes)               */
    unsigned int nAttr;                /* Mode attribute value              */
} ModeInfoEntry;

typedef struct tagModeListHeader       /* Mode list header structure        */
{
    int         nMode;                 /* Number of modes in the list       */
} ModeListHeader;

typedef struct tagModeListEntry        /* Mode list entry structure         */
{
    ModeInfoEntry ModeEntry;           /* Mode information entry            */
    MonInfoEntry *pMonEntry;           /* Monitor mode index value          */
} ModeListEntry;

typedef struct tagModeList             /* Mode list structure               */
{
    ModeListHeader      ModeHeader;     /* Mode list header                  */
    ModeListEntry       ModeEntry[1];   /* Start of the mode list entries    */
} ModeList;

typedef struct tagModeInfo             /* Mode information structure        */
{
    ModeInfoEntry       ModeEntry;      /* Mode information entry            */
} ModeInfo;
```

```
/*****************************************************************************
*
*   Function Prototypes
*****************************************************************************
/
MonList *GetMonitorList(void);
MonInfo *GetMonitorInfo(char *);
ModeList *GetModeList(MonInfo *, char *);
ModeInfo *GetModeInfo(char *, char *);
BYTE *GetModeTable(char *, char *);
void SetMode(BYTE *, BYTE *);

#endif // _MODEMON_H_
```

3.7 Programming Examples

3.7.1 CL-GD5464 Setup

The following source code set up the CL-GD5464 for operation, and shows an example of executing a simple display list and exiting. The code uses some external routines that are provided and some library routines that are not provided.

```
/**********************************************************************
*
*   Module:     trm.c
*
*   Revision:   1.00
*
*   Date:       August 30, 1996
*
*   Author:     Cirrus Logic Austin Design Center
*
**********************************************************************
*
*   Module Description:
*
*       Set up for Laguna 3D 5464 part
*
**********************************************************************
*
*   Changes:
*
*    DATE     REVISION  DESCRIPTION                       AUTHOR
*  --------   --------  ------------------------------------------
**********************************************************************/

#include <stdlib.h>                 // Include standard library
#include <stdio.h>                  // Include standard input/output
#include <conio.h>                  // getch
#include <i86.h>                    // Include x86 specific library

#include "trm.h"                    // self
#include "l3struct.h"               // laguna structures
#include "l3types.h"                // laguna types
#include "modemon.h"                // ModeInfo

// global variables
```

```
TSystem     LL_State;               // main global Laguna device state
ModeInfo   *pModeInfo;              // Pointer to mode information
char       *pModeTable;             // Pointer to mode table

// useful local definitions

#define ERROR              1
#define DISPLAY_LIST_SIZE  1048576  // 1 MB for testing purposes
#define TEXTURE_LENGTH     1048576  // 1 MB for host textures
#define DL_START_OFFSET    20       // 5 dwords offset for display list
#define EBIOS_CLGD5464     0x64     // BIOS Laguna 3D signature
#define EBIOS_CLGD5464B    0x61     // BIOS Laguna 3D alternate signature

/*********************************************************************
*
*   void LL_Init()
*
*   Set up Laguna 3D for operation
*
*********************************************************************/

int LL_Init()                       // set up
{
    int         rc=0;               // return code
    union REGS  r;                  // i86 registers
    char        sBuf[128];          // Temp buffer for initialization info
    char        sMode[128];         // Temp buffer for mode name
    DWORD       z_buf_addr;
    DWORD      *pTex;

   // INITIALIZE DISPLAY ADAPTER ==========================================
    //
    // The following code section relies on library calls to do such things as
    // reading initialization files, obtaining PCI addresses and setting
    // the graphics mode. These routines are not provided as part of this
    // example but are expected to be provided as an appendix to this document.
    // The routines are:
    //
    // GetPrivateProfileString()
    // GetRegisterApperture()
    // GetLagunaApperture()
    // GetModeInfo()
    // GetModeTable()
    // SetMode()

    // read and store chipset-specific and other settings from .ini files
    //
    GetPrivateProfileString( "SYSTEM", "PCI_MASTER_READ", "SINGLE",
        sBuf, 128, "LAGUNA.INI" );

    if( !strcmp(sBuf, strupr("BURST")) )
        LL_State.fSingleRead = 0;
    else
        LL_State.fSingleRead = 1;

    GetPrivateProfileString( "SYSTEM", "PCI_MASTER_WRITE", "SINGLE",
        sBuf, 128, "LAGUNA.INI" );

    if( !strcmp(sBuf, strupr("BURST")) )
        LL_State.fSingleWrite = 0;
    else
        LL_State.fSingleWrite = 1;

    GetPrivateProfileString( "SYSTEM", "PCI_MASTER_LATENCY_TIMER", "255",
        sBuf, 128, "LAGUNA.INI" );

    sscanf( sBuf, "%d", &LL_State.dwLatencyTimer );

    // First, use Extended BIOS to get the graphics card information
    //
    r.h.ah = 0x12;
    r.h.bl = 0x80;
    int386( 0x10, &r, &r );

    if( (r.w.ax != EBIOS_CLGD5464) &&
        (r.w.ax != EBIOS_CLGD5464B) )

            return( ERROR );

    // Get the amount of the video memory present
    //
    r.h.ah = 0x12;
    r.h.bl = 0x85;
    int386( 0x10, &r, &r );
```

```
LL_State.dwVRAM = r.h.al * 64 * 1024;

// Get the register and frame appertures
//
LL_State.pRegs  = (BYTE *) GetRegisterApperture();
LL_State.pFrame = (BYTE *) GetLagunaApperture(1);

// Get the mode information and set the structure that defines
// the state of the graphics device (DC)
//
GetPrivateProfileString( "MODE", "VIDEO", "MODE_640X480X16_1",
    sMode, 128, "TRM.INI" );

pModeInfo = GetModeInfo( "5462", sMode );
if( pModeInfo == NULL )
{
    printf("error getting mode info\n");
    return( ERROR );
}

// retrieve screen size and pixel depth parameters
//
LL_State.wHoriz = pModeInfo->ModeEntry.nResX;
LL_State.wVert  = pModeInfo->ModeEntry.nResY;
LL_State.wBpp   = pModeInfo->ModeEntry.nBPP;

// We dont need mode structure any more
//
free( pModeInfo );

// now we will actually set the video mode, so get the mode tables
//
pModeTable = GetModeTable( "5462", sMode );
if( pModeInfo == NULL )
{
    printf("error getting mode table\n");
    return( ERROR );
}

// set the video mode and free the structure
//
SetMode( pModeTable, LL_State.pRegs );
```

```
 free( pModeTable );

 // clear all of RDRAM
 //
 memset( LL_State.pFrame, 0, LL_State.dwVRAM );

// calculate the pitch of the given graphics mode: read CR13 (Offset Register)
 //
 LL_State.pitch = *(LL_State.pRegs + 0x4C) & 0xff;

 // read CR1B, which contains Offset Register[8]
 //
 if( *(LL_State.pRegs + 0x6c) & 0x10 )
     LL_State.pitch += 256;

 // read CR1D, which contains Offset Register[9]
 //
 if( *(LL_State.pRegs + 0x74) & 1 )
     LL_State.pitch += 512;

 // pitch is an oct-byte value, convert to bytes
 //
 LL_State.pitch *= 8;

// INITIALIZE REGISTER FILE ===============================================

 CLEAR_RANGE( X_3D, DU_ORTHO_ADD_3D );   // clear 3D interpolators
 CLEAR_RANGE( A_3D, DA_ORTHO_3D );       // clear 3D interpolators
 CLEAR_RANGE( COLOR_MIN_BOUNDS_3D, COLOR_MAX_BOUNDS_3D );

SETREG( WIDTH1_3D, 0x10000 );          // init polyengine reg WIDTH1_3D to 1
 SETREG( CONTROL_MASK_3D, 0 );           // enable writes to all fields
 SETREG( CONTROL1_3D, 0 );               // initialize

 // set Base0 address register:
 //  * Color buffer X offset of 0
 //  * Color buffer location in RDRAM
 //  * Z buffer location in RDRAM
 //  * Textures in RDRAM
 //  * Pattern offset of 0
 //
 SETREG( BASE0_ADDR_3D, 0 );
```

```
// set Base1 address register:
//  * Color buffer Y offset of 0
//  * Z buffer Y offset to 0
//
SETREG( BASE1_ADDR_3D, 0 );

// set texture control register:
//  * Texture U, V masks to 16
//  * Texture U, V wraps
//  * Texel mode temporarily to 0
//  * Texel lookup to no lookup
//  * Texture data is lighting source
//  * Filtering disabled
//  * Texture polarity of type 0
//  * Texture masking diasabled
//  * Texture mask function to Write mask
//  * Address mux to 0
//  * CLUT offset to 0
//
SETREG( TX_CTL0_3D,       0);

SETREG( TX_XYBASE_3D,     0 );          // Set texture base reg and cache
SETREG( TX_CTL1_3D,       0 );          // Set tex color bounds
SETREG( TX_CTL2_3D,       0 );          // Set tex color bounds
SETREG( COLOR0_3D,        0 );          // Set color 0 reg/cache
SETREG( COLOR0_3D,        0 );          // Set color 1 reg/cache
SETREG( X_CLIP_3D,        0 );          // Reset clipping reg
SETREG( Y_CLIP_3D,        0 );          // Reset clipping reg
SETREG( TEX_SRAM_CTRL_3D, 0 );          // Set a 2D ctrl reg

// INITIALIZE HOST XY UNIT REGISTERS ======================================

// set host xy control register:
//  * HostXY is disabled
//  * Pitch is set to 1024 (100001b)
//
SETREG( HXY_HOST_CTRL_3D, 0x21 << 8);

// intialize host base 0 regs
SETREG( HXY_BASE0_ADDRESS_PTR_3D, 0 );
SETREG( HXY_BASE0_START_XY_3D,    0 );
```

```
    SETREG( HXY_BASE0_EXTENT_XY_3D,   0 );

    // initialize host base 1 regs
    SETREG( HXY_BASE1_ADDRESS_PTR_3D, 0 );
    SETREG( HXY_BASE1_OFFSET0_3D,     0 );
    SETREG( HXY_BASE1_OFFSET1_3D,     0 );
    SETREG( HXY_BASE1_LENGTH_3D,      0 );

    // initialize mailbox registers
    SETREG( MAILBOX0_3D, 0 );
    SETREG( MAILBOX1_3D, 0 );
    SETREG( MAILBOX2_3D, 0 );
    SETREG( MAILBOX3_3D, 0 );

    // INITIALIZE PREFETCH UNIT REGISTERS =====================================

    SETREG( PF_CTRL_3D,      0);            // Disable Prefetch
    SETREG( PF_BASE_ADDR_3D, 0 );           // Set prefetch base reg
    SETREG( PF_INST_3D,      IDLE );        // Write an IDLE instruction
    SETREG( PF_DEST_ADDR_3D, 0 );           // Set prefetch dest address
    SETREG( PF_FB_SEG_3D,    0 );           // Set frame segment reg
    SETREG( PF_STATUS_3D,    0 );           // Reset Display_List_Switch

    // set read/write bursting mode: these values are read from a file in this
    // example so as to be configurable based on the capabilities of a specific
chipset
    //
    SETREG( HOST_MASTER_CTRL_3D, (LL_State.fSingleRead << 1) |
LL_State.fSingleWrite );

    // 3d instruction track disable, fetch on request, enable instruction fetch
    SETREG( PF_CTRL_3D, 0x19);

    // set up control0_3d register for 16 bpp mode, 565, minimal features
    //
    // start with all bits clear: that implies
    //  * Z mode normal, Z-collision detect disabled, 16-bit Z buffer
    //  * Lighting source is poly engine
    //  * Alpha blending disabled
    //  * Color saturation disabled
    //  * Color compares disabled
    //  * Pixel mask disabled
```

```
//
LL_State.dwControl0 = 0;

// now set pixel depth
//
LL_State.Control0.Pixel_Mode = PIXEL_MODE_565;

// write the register
//
SETREG( CONTROL0_3D, LL_State.dwControl0 );

// INITIALIZE BUFFER MANAGEMENT: A, B (textures) and Z ====================

rc = LL_InitBuffers();
if (rc)
    goto exit;

// calculate z buffer y offset
z_buf_addr = LL_State.BufZ.Extent.top;
z_buf_addr /= 32;           // this is a 32-line offset from color buffer
z_buf_addr <<= 21;          // the address is stored in BASE1_ADDR[28:21]

// Set rendering buffer x and y offsets, z buffer y offset
SETREG( BASE0_ADDR_3D, 0);
SETREG( BASE1_ADDR_3D, z_buf_addr);

// INITIALIZE A SINGLE DISPLAY LIST =======================================
//
// This section relies on library routines to allocate and manage memory.
// They are expected to be provided as an appendix to this document.
// The routines are:
//
// AllocSystemMemory()
// GetLinearAddress()
// GetPhysicalAddress()

// allocate 1 MB system memory for the display list
//
if( (LL_State.DL.hMem = AllocSystemMemory( DISPLAY_LIST_SIZE )) == 0 )
    return( ERROR );

// retrieve the linear and physical addresses of the display list
```

```
    //
    LL_State.DL.pdwLinPtr = (DWORD *) GetLinearAddress( LL_State.DL.hMem );
    LL_State.DL.dwPhyPtr  = GetPhysicalAddress( LL_State.DL.hMem );

    // set the length and the display list pointer to point to an offset
    // of 20: 4 dwords are reserved for an interrupt jump table, and an
    // additional dword for a display list semaphore (this is implementation-
    // dependent)
    //
    LL_State.DL.dwLen   = DISPLAY_LIST_SIZE;
    LL_State.DL.pdwNext = LL_State.DL.pdwLinPtr + 5;    // + 20 bytes

    // clear the jump table and a semaphore
    //
    LL_State.DL.pdwNext[0] = IDLE;
    LL_State.DL.pdwNext[1] = IDLE;
    LL_State.DL.pdwNext[2] = IDLE;
    LL_State.DL.pdwNext[3] = IDLE;
    LL_State.DL.pdwNext[4] = 0;

    // temporary fix for non-flushing TLB
    //
    *(DWORD *)((DWORD)LL_State.DL.pdwLinPtr + DISPLAY_LIST_SIZE - 16) = BRANCH
+ 20;

    // write base address of display list to the register that stores this value
    //
    SETREG( PF_BASE_ADDR_3D, LL_State.DL.dwPhyPtr );

    // make the screen white
    //
    memset( LL_State.pFrame, 255, LL_State.wHoriz * LL_State.wVert * 2);

    // create a texture on-the-fly to fill buffer B
    //
    for (pTex = (DWORD *)LL_State.BufTextures.dwAddress;
         pTex <= (DWORD *)LL_State.BufTextures.dwAddress + LL_State.wHoriz *
LL_State.wVert / 2;
         pTex++)
    {
        *pTex = (DWORD) pTex;
    }
```

```
exit:
    return( rc );
}

/**********************************************************************
*
*   void LL_Terminate()
*
*       Shutdown Laguna 3D after operation
*
*
***********************************************************************/

int LL_Terminate()                          // shutdown Laguna 3D
{
    union REGS r;

    LL_Wait();

    // Get the video mode info to reset
    //
    pModeTable = GetModeTable( "5462", "MODE_RESET" );

    if( pModeInfo == NULL )
        return( ERROR );

    // Set the video mode "reset"
    //
    SetMode( pModeTable, LL_State.pRegs );
    free( pModeTable );

    // Finally, set the BIOS video mode 3
    //
    r.w.ax = 0x0003;
    int386( 0x10, &r, &r );

    return( 0 );
}

/**********************************************************************
*
```

```
*   void LL_print_regs()
*
*   Wait for 3D engine to idle by spinning on prefetch status register
*
**********************************************************************/

void LL_Wait()
{
    while ( (*(DWORD *)(LL_State.pRegs + PF_STATUS_3D)) & 0x3e0)
    {
        printf("status: %08x\n", *(DWORD *)(LL_State.pRegs + PF_STATUS_3D));
    };
}

/*********************************************************************
*
*   void main()
*
**********************************************************************/

void main()
{
    int         rc=0;
    BYTE       *pRegs;

    // setup
    rc = LL_Init();
    if (rc)
        goto exit;

    printf("setup: %s\n", rc == 0 ? "OK" : "ERROR");

    // obtain local pointer to Laguna register file
    pRegs = LL_State.pRegs;

    // dump Laguna 5464 state information if desired
    // LL_print_state();

    // dump Laguna 5464 registers if desired
    // LL_print_regs();

    // start a simple display list:
```

```
    // any of the display lists from dlists.c can be inserted here
    //
    dl_poly_set4(LL_State.DL.pdwNext);

    // branch to the display list and execute
    *(DWORD *)(LL_State.pRegs + PF_INST_3D) = BRANCH + DL_START_OFFSET;

    // wait after the display list before we launch another
    LL_Wait();
    getch();

    // peek at our texture buffer
    LL_SetDisplayBuffer(&LL_State.BufTextures);
    getch();

    // peek at the z buffer
    LL_SetDisplayBuffer(&LL_State.BufZ);
    getch();

    // shutdown Laguna
    rc = LL_Terminate();
    printf("shutdown: %s\n", rc == 0 ? "OK" : "ERROR");

exit:
    printf("exit: %x\n", rc);
}
```

3.7.2 Z-Buffered Points

Each of the following code examples require the include and extern statements from the Z-Buffered Points example.

```
/**********************************************************************
*
*   Module:     dlists.c
*
*   Revision:   1.00
*
*   Date:       Sept 3, 1996
*
*   Author:     Cirrus Logic Austin Design Center
*
***********************************************************************
*
*   Module Description:
*
*       Various display lists as TRM examples
*
***********************************************************************
*
*   Changes:
*
*    DATE     REVISION  DESCRIPTION                       AUTHOR
*  --------   --------  ----------------------------------------------
**********************************************************************/

#include <stdio.h>                  // printf
#include "l3types.h"                // laguna types
#include "l3struct.h"               // TSystem

extern TSystem  LL_State;

/**********************************************************************
*
*  dl_point_set0()
*
*  draw three z-buffered points in the top left corner, red, green, blue
*
**********************************************************************/
```

```
void dl_point_set0(DWORD *pdwNext)
{
    *pdwNext++ = 0x18000181;// WRITE_3D_REG   reg: 4018  Y_COUNT_3D
    *pdwNext++ = 0x00000000;// - set y count to 1 for points

    *pdwNext++ = 0x00002006;// DRAW_POINT     im: 0002Z-Buffer
    *pdwNext++ = 0x00100000;//   X_3D           X value:   16.000
    *pdwNext++ = 0x00100000;//   Y_3D           Y value:   16.000
    *pdwNext++ = 0x00ff0000;//   R_3D           R value:  255.000
    *pdwNext++ = 0x00000000;//   G_3D           G value:    0.000
    *pdwNext++ = 0x00000000;//   B_3D           B value:    0.000
    *pdwNext++ = 0x27810000;//   Z_3D           Z val:  10113.000
    *pdwNext++ = 0x00002006;// DRAW_POINT     im: 0002Z-Buffer
    *pdwNext++ = 0x00100000;//   X_3D           X value:   16.000
    *pdwNext++ = 0x00200000;//   Y_3D           Y value:   32.000
    *pdwNext++ = 0x00000000;    //   R_3D           R value:    0.000
    *pdwNext++ = 0x00ff0000;//   G_3D           G value:  255.000
    *pdwNext++ = 0x00000000;//   B_3D           B value:    0.000
    *pdwNext++ = 0x3f540000;//   Z_3D           Z val:  16212.000
    *pdwNext++ = 0x00002006;// DRAW_POINT     im: 0002Z-Buffer
    *pdwNext++ = 0x00200000;//   X_3D           X value:   32.000
    *pdwNext++ = 0x00100000;//   Y_3D           Y value:   16.000
    *pdwNext++ = 0x00000000;//   R_3D           R value:    0.000
    *pdwNext++ = 0x00000000;//   G_3D           G value:    0.000
    *pdwNext++ = 0x00ff0000;//   B_3D           B value:  255.000
    *pdwNext++ = 0x7de10000;//   Z_3D           Z val:  32225.000
    *pdwNext++ = 0x68000000;// IDLE
}
```

3.7.3 Alpha-Blended Points

```
/*********************************************************************
*
*  dl_point_set1()
*
*  draw three mostly red points at top of screen, alpha blend with
*  three mostly green points at the same spot, result is a dull yellow
*
*********************************************************************/

void dl_point_set1(DWORD *pdwNext)
{
    *pdwNext++ = 0x18000181;    // WRITE_3D_REG   reg: 4018  Y_COUNT_3D
    *pdwNext++ = 0x00000000;    // - set y count to 1 for points

    *pdwNext++ = 0x00000005;    // DRAW_POINT     im: 0000
    *pdwNext++ = 0x00000000;    //   X_3D           X value:    0.000
    *pdwNext++ = 0x00000000;    //   Y_3D           Y value:    0.000
    *pdwNext++ = 0x00ff0000;    //   R_3D           R value:  255.000
    *pdwNext++ = 0x00320000;    //   G_3D           G value:   50.000
    *pdwNext++ = 0x00320000;    //   B_3D           B value:   50.000
    *pdwNext++ = 0x00000005;    // DRAW_POINT     im: 0000
    *pdwNext++ = 0x00010000;    //   X_3D           X value:    1.000
    *pdwNext++ = 0x00000000;    //   Y_3D           Y value:    0.000
    *pdwNext++ = 0x00ff0000;    //   R_3D           R value:  255.000
    *pdwNext++ = 0x00320000;    //   G_3D           G value:   50.000
    *pdwNext++ = 0x00320000;    //   B_3D           B value:   50.000
    *pdwNext++ = 0x00000005;    // DRAW_POINT     im: 0000
    *pdwNext++ = 0x00020000;    //   X_3D           X value:    2.000
    *pdwNext++ = 0x00000000;    //   Y_3D           Y value:    0.000
    *pdwNext++ = 0x00ff0000;    //   R_3D           R value:  255.000
    *pdwNext++ = 0x00320000;    //   G_3D           G value:   50.000
    *pdwNext++ = 0x00320000;    //   B_3D           B value:   50.000

    *pdwNext++ = 0x18001041;    // WRITE_3D_REG   reg: 4104  CONTROL0_3D
    *pdwNext++ = 0x04008002;    // - light source = COLOR_REG1_3D
                                // - alpha blending enable
                                // - preserve 16 bpp

    *pdwNext++ = 0x18000c42;    // WRITE_3D_REG   reg: 40c4  DA_MAIN_3D
    *pdwNext++ = 0x00800000;    // - set da_main for constant alpha blending
    *pdwNext++ = 0x00800000;    // - set da_ortho for constant alpha blending
    *pdwNext++ = 0x00800005;    // DRAW_POINT     im: 0800FetchColor
    *pdwNext++ = 0x00000000;    //   X_3D           X value:    0.000
    *pdwNext++ = 0x00000000;    //   Y_3D           Y value:    0.000
    *pdwNext++ = 0x00320000;    //   R_3D           R value:   50.000
    *pdwNext++ = 0x00ff0000;    //   G_3D           G value:  255.000
    *pdwNext++ = 0x00320000;    //   B_3D           B value:   50.000
    *pdwNext++ = 0x00800005;    // DRAW_POINT     im: 0800FetchColor
    *pdwNext++ = 0x00010000;    //   X_3D           X value:    1.000
    *pdwNext++ = 0x00000000;    //   Y_3D           Y value:    0.000
    *pdwNext++ = 0x00320000;    //   R_3D           R value:   50.000
    *pdwNext++ = 0x00ff0000;    //   G_3D           G value:  255.000
    *pdwNext++ = 0x00320000;    //   B_3D           B value:   50.000
    *pdwNext++ = 0x00800005;    // DRAW_POINT     im: 0800FetchColor
    *pdwNext++ = 0x00020000;    //   X_3D           X value:    2.000
    *pdwNext++ = 0x00000000;    //   Y_3D           Y value:    0.000
    *pdwNext++ = 0x00320000;    //   R_3D           R value:   50.000
    *pdwNext++ = 0x00ff0000;    //   G_3D           G value:  255.000
    *pdwNext++ = 0x00320000;    //   B_3D           B value:   50.000
    *pdwNext++ = 0x68000000;// IDLE
}
```

3.7.4 Gouraud-Shaded Lines

```
/**********************************************************************
*
*  dl_line_set0()
*
*  draw three gouraud shaded lines
*
**********************************************************************/

void dl_line_set0(DWORD *pdwNext)
{
    *pdwNext++ = 0x0800100b;    // DRAW_LINE      im: 0001Gouraud
    *pdwNext++ = 0x02628000;    //   X_3D           X value:  610.500
    *pdwNext++ = 0x00db8000;    //   Y_3D           Y value:  219.500
    *pdwNext++ = 0x00930000;    //   R_3D           R value:  147.000
    *pdwNext++ = 0x00060000;    //   G_3D           G value:    6.000
    *pdwNext++ = 0x00c70000;    //   B_3D           B value:  199.000
    *pdwNext++ = 0xffff0000;    //   DX_MAIN_3D     dx main:   -1.000
    *pdwNext++ = 0x0000019c;    //   Y_COUNT_3D     y count1: 0 y count2: 412
    *pdwNext++ = 0x0000a0ed;    //   WIDTH1_3D      width 1:    0.629
    *pdwNext++ = 0x00000d0c;    //   DR_MAIN_3D     dr main:    0.051
    *pdwNext++ = 0x00006901;    //   DG_MAIN_3D     dg main:    0.410
    *pdwNext++ = 0xfffffe22;    //   DB_MAIN_3D     db main:   -1.993
    *pdwNext++ = 0x0800100b;    // DRAW_LINE      im: 0001Gouraud
    *pdwNext++ = 0x007c8000;    //   X_3D           X value:  124.500
    *pdwNext++ = 0x003c8000;    //   Y_3D           Y value:   60.500
    *pdwNext++ = 0x005f0000;    //   R_3D           R value:   95.000
    *pdwNext++ = 0x00cb0000;    //   G_3D           G value:  203.000
    *pdwNext++ = 0x00630000;    //   B_3D           B value:   99.000
    *pdwNext++ = 0x0000cd91;    //   DX_MAIN_3D     dx main:    0.803
    *pdwNext++ = 0x00000042;    //   Y_COUNT_3D     y count1: 0 y count2: 66
    *pdwNext++ = 0x00010000;    //   WIDTH1_3D      width 1:    1.000
    *pdwNext++ = 0x0001745a;    //   DR_MAIN_3D     dr main:    1.454
    *pdwNext++ = 0xfffef07e;    //   DG_MAIN_3D     dg main:   -2.939
    *pdwNext++ = 0xffff07c;    //   DB_MAIN_3D     db main:   -1.939
    *pdwNext++ = 0x0800100b;    // DRAW_LINE      im: 0001Gouraud
    *pdwNext++ = 0x01bc8000;    //   X_3D           X value:  444.500
    *pdwNext++ = 0x00b98000;    //   Y_3D           Y value:  185.500
    *pdwNext++ = 0x00590000;    //   R_3D           R value:   89.000
    *pdwNext++ = 0x00ac0000;    //   G_3D           G value:  172.000
    *pdwNext++ = 0x00740000;    //   B_3D           B value:  116.000
    *pdwNext++ = 0x000064fb;    //   DX_MAIN_3D     dx main:    0.394
    *pdwNext++ = 0x0000006d;    //   Y_COUNT_3D     y count1: 0 y count2: 109
    *pdwNext++ = 0x00010000;    //   WIDTH1_3D      width 1:    1.000
    *pdwNext++ = 0x00010705;    //   DR_MAIN_3D     dr main:    1.027
    *pdwNext++ = 0xffffb4da;    //   DG_MAIN_3D     dg main:   -1.706
    *pdwNext++ = 0x000133a3;    //   DB_MAIN_3D     db main:    1.202
    *pdwNext++ = 0x68000000;    // IDLE
}
```

3.7.5 Gouraud-Shaded, Dithered Polygon

```
/**********************************************************************
*
*  dl_poly_set0()
*
*  draw a single polygon that is gouraud shaded and dithered
*
**********************************************************************/

void dl_poly_set0(DWORD *pdwNext)
{
    *pdwNext++ = 0x1020100f;    // DRAW_POLY      im: 0201Dither Gouraud
    *pdwNext++ = 0x81fb8000;    //   X_3D           X value:  507.500
    *pdwNext++ = 0x01120000;    //   Y_3D           Y value:  274.000
    *pdwNext++ = 0x00ba0000;    //   R_3D           R value:  186.000
    *pdwNext++ = 0x00420000;    //   G_3D           G value:   66.000
    *pdwNext++ = 0x00710000;    //   B_3D           B value:  113.000
    *pdwNext++ = 0xffffda40;    //   DX_MAIN_3D     dx main:   -1.853
    *pdwNext++ = 0x00280015;    //   Y_COUNT_3D     y count1: 40 y count2: 21
    *pdwNext++ = 0x0014fff0;    //   DWIDTH1_3D     dwidth1:   21.999
    *pdwNext++ = 0xfff42773;    //   DWIDTH2_3D     dwidth2:  -12.308
    *pdwNext++ = 0xffff5c55;    //   DR_MAIN_3D     dr main:   -1.361
    *pdwNext++ = 0x00028211;    //   DG_MAIN_3D     dg main:    2.508
    *pdwNext++ = 0x0001714b;    //   DB_MAIN_3D     db main:    1.443
    *pdwNext++ = 0x00001010;    //   DR_ORTHO_3D    dr orth:    0.063
    *pdwNext++ = 0xffffee46;    //   DG_ORTHO_3D    dg orth:   -1.931
    *pdwNext++ = 0x00000f82;    //   DB_ORTHO_3D    db orth:    0.061
    *pdwNext++ = 0x68000000;    // IDLE
}
```

3.7.6 Polygons with Z-Buffering, Flat-Shading, and Other Modifiers

```
/**********************************************************************
*
*  dl_poly_set1()
*
*  draw 2 polygons with various features
*
**********************************************************************/

void dl_poly_set1(DWORD *pdwNext)
{
    // z-buffered, flat-shaded, dithered polygon, load initial width: flat top

    *pdwNext++ = 0x1120200e;    // DRAW_POLY      im: 1202InitialWdth Dither Z-
Buffer
    *pdwNext++ = 0x000a8000;    //   X_3D           X value:   10.500
    *pdwNext++ = 0x000a0000;    //   Y_3D           Y value:   10.000
    *pdwNext++ = 0x00640000;    //   R_3D           R value:  100.000
    *pdwNext++ = 0x00320000;    //   G_3D           G value:   50.000
    *pdwNext++ = 0x00c80000;    //   B_3D           B value:  200.000
    *pdwNext++ = 0x00000005;    //   DX_MAIN_3D     dx main:    0.000
    *pdwNext++ = 0x000000be;    //   Y_COUNT_3D     y count1: 0 y count2: 190
    *pdwNext++ = 0x01230000;    //   WIDTH1_3D      width 1:  291.000
    *pdwNext++ = 0x00000000;    //   WIDTH2_3D      width 2:    0.000
    *pdwNext++ = 0xfffe7953;    //   DWIDTH1_3D     dwidth1:   -2.947
    *pdwNext++ = 0x00000000;    //   DWIDTH2_3D     dwidth2:    0.000
    *pdwNext++ = 0x00640000;    //   Z_3D           Z val:    100.000
    *pdwNext++ = 0x000f38e0;    //   DZ_MAIN_3D     Z main:    15.222
    *pdwNext++ = 0x00031380;    //   DZ_ORTHO_3D    Z orth:     3.076
    *pdwNext++ = 0x180001c1;    // WRITE_3D_REG   reg: 401c  WIDTH1_3D
    *pdwNext++ = 0x00010000;    // - reset initial width back to 1

    // z-buffered, gouraud shaded, dithered polygon

    *pdwNext++ = 0x10203012;    // DRAW_POLY      im: 0203DitherZ-BufferGouraud
    *pdwNext++ = 0x80328000;    //   X_3D           X value:   50.500
    *pdwNext++ = 0x00140000;    //   Y_3D           Y value:   20.000
    *pdwNext++ = 0x00000000;    //   R_3D           R value:    0.000
    *pdwNext++ = 0x00ff0000;    //   G_3D           G value:  255.000
    *pdwNext++ = 0x00ff0000;    //   B_3D           B value:  255.000
    *pdwNext++ = 0x0001942b;    //   DX_MAIN_3D     dx main:    1.579
    *pdwNext++ = 0x000b00b3;    //   Y_COUNT_3D     y count1: 11 y count2: 179
    *pdwNext++ = 0x000193e1;    //   DWIDTH1_3D     dwidth1:    2.155
    *pdwNext++ = 0xffe399a5;    //   DWIDTH2_3D     dwidth2:  -30.200
    *pdwNext++ = 0x00015787;    //   DR_MAIN_3D     dr main:    1.342
    *pdwNext++ = 0xfffea878;    //   DG_MAIN_3D     dg main:   -2.658
    *pdwNext++ = 0x00000000;    //   DB_MAIN_3D     db main:    0.000
    *pdwNext++ = 0x00000c9e;    //   DR_ORTHO_3D    dr orth:    0.049
    *pdwNext++ = 0x0000da21;    //   DG_ORTHO_3D    dg orth:    0.852
    *pdwNext++ = 0xffff1a26;    //   DB_ORTHO_3D    db orth:   -1.102
    *pdwNext++ = 0x03e80000;    //   Z_3D           Z val:   1000.000
    *pdwNext++ = 0x0024be40;    //   DZ_MAIN_3D     Z main:    36.743
    *pdwNext++ = 0xffe94220;    //   DZ_ORTHO_3D    Z orth:   -23.258
    *pdwNext++ = 0x68000000;    // IDLE
}
```

3.7.7 Polygon Showing Z-Buffering, Stippling, and Constant Lighting

```
/*********************************************************************
*
*  dl_poly_set2()
*
*  draw 1 polygon: uses constant lighting
*
*********************************************************************/

void dl_poly_set2(DWORD *pdwNext)
{
    // you have to load the pattern registers if you want to do stipple ...

    *pdwNext++ = 0x18001408;    // WRITE_3D_REG   reg: 4114  PATTERN_RAM_0_3D
    *pdwNext++ = 0x04150415;    // load pattern ram
    *pdwNext++ = 0x62736273;
    *pdwNext++ = 0x15041504;
    *pdwNext++ = 0x73627362;
    *pdwNext++ = 0x04150415;
    *pdwNext++ = 0x62736273;
    *pdwNext++ = 0x15041504;
    *pdwNext++ = 0x73627362;

    // render a stippled, lit, z-buffered poly that is flat-top (initial width
load)

    *pdwNext++ = 0x18001041;    // WRITE_3D_REG   reg: 4104  CONTROL0_3D
    *pdwNext++ = 0x04000002;    // - light source = COLOR_REG1_3D
                                // - preserve 16 bpp

    *pdwNext++ = 0x18001341;    // WRITE_3D_REG   reg: 4134  COLOR1_3D
    *pdwNext++ = 0x0022ff22;    // - load lighting value: keep red, dim green,
blue

    *pdwNext++ = 0x110c3014;    // DRAW_POLY  im: 10c2 (iw, stipple, light, z,
gouraud)
    *pdwNext++ = 0x00c88000;    //   X_3D           X value:  200.500
    *pdwNext++ = 0x00640000;    //   Y_3D           Y value:  100.000
    *pdwNext++ = 0x00640000;    //   R_3D           R value:  100.000
    *pdwNext++ = 0x00640000;    //   G_3D           G value:  100.000
    *pdwNext++ = 0x00f00000;    //   B_3D           B value:  240.000
    *pdwNext++ = 0x00005558;    //   DX_MAIN_3D     dx main:    0.333
```

```
        *pdwNext++ = 0x0000012c;    //   Y_COUNT_3D     y count1: 0 y count2: 300
        *pdwNext++ = 0x00c90000;    //   WIDTH1_3D      width 1:  201.000
        *pdwNext++ = 0x00000000;    //   WIDTH2_3D      width 2:    0.000
        *pdwNext++ = 0xffff5559;    //   DWIDTH1_3D     dwidth1:   -1.666
        *pdwNext++ = 0x00000000;    //   DWIDTH2_3D     dwidth2:    0.000

        *pdwNext++ = 0x00015787;    //   DR_MAIN_3D     dr main:    1.342
        *pdwNext++ = 0xfffea878;    //   DG_MAIN_3D     dg main:   -2.658
        *pdwNext++ = 0x00000000;    //   DB_MAIN_3D     db main:    0.000
        *pdwNext++ = 0x00000c9e;    //   DR_ORTHO_3D    dr orth:    0.049
        *pdwNext++ = 0x0000da21;    //   DG_ORTHO_3D    dg orth:    0.852
        *pdwNext++ = 0xffff1a26;    //   DB_ORTHO_3D    db orth:   -1.102

        *pdwNext++ = 0x23280000;    //   Z_3D           Z val:   9000.000
        *pdwNext++ = 0xfff60550;    //   DZ_MAIN_3D     Z main:   -10.021
        *pdwNext++ = 0xffe22760;    //   DZ_ORTHO_3D    Z orth:   -30.154
        *pdwNext++ = 0x180001c1;    // WRITE_3D_REG   reg: 401c  WIDTH1_3D
        *pdwNext++ = 0x00010000;    // - reset initial width

        *pdwNext++ = 0x68000000;    // IDLE
    }
```

3.7.8 Polygons Showing Texture Mapping

```
/*********************************************************************
*
*  dl_poly_set3()
*
*  draw 2 polygons with various features
*
**********************************************************************/

void dl_poly_set3(DWORD *pdwNext)
{
    DWORD   tex_y_addr;

    // set y offset for texture buffer; x offset is set to 0

    tex_y_addr = (LL_State.BufTextures.Extent.top / 16) << 20;

    // textured, z-buffered polygon (dont-load-color: no need with textures)
    //
    // NOTE: this example assumes a texture map has been previously loaded
    // into RDRAM into our texture buffer, and that the texture is in
    // 5:6:5 mode with dimensions 128x128 (this was accomplished by creating
    // a texture on-the-fly in LL_Init() for this example code)

    *pdwNext++ = 0x18001202;    // WRITE_3D_REG   reg: 4120  TX_CTL0_3D
    *pdwNext++ = 0x00000433;    // - texture is 5:6:5, 128 x 128
    *pdwNext++ = tex_y_addr;    // - tx_xybase reg = texture xy address

    *pdwNext++ = 0x1002a00f;    // DRAW_POLY      im: 002aTexLin Z-Buffer
DontLoadColor
    *pdwNext++ = 0x82268000;    //   X_3D           X value:  550.500
    *pdwNext++ = 0x00960000;    //   Y_3D           Y value:  150.000
    *pdwNext++ = 0xffff99a0;    //   DX_MAIN_3D     dx main:   -1.600
    *pdwNext++ = 0x00650095;    //   Y_COUNT_3D     y count1: 101 y count2: 149
    *pdwNext++ = 0x0001eee5;    //   DWIDTH1_3D     dwidth1:    2.866
    *pdwNext++ = 0xfffd19a8;    //   DWIDTH2_3D     dwidth2:   -3.200
    *pdwNext++ = 0x08340000;    //   Z_3D           Z val:   2100.000
    *pdwNext++ = 0x000d97b0;    //   DZ_MAIN_3D     Z main:    13.593
    *pdwNext++ = 0x00085fc0;    //   DZ_ORTHO_3D    Z orth:     8.374
    *pdwNext++ = 0x00000000;    //   V_3D           V value:    0.000
    *pdwNext++ = 0x007f0000;    //   U_3D           U value:  127.000
    *pdwNext++ = 0x0000820e;    //   DV_MAIN_3D     DV main:    0.508
    *pdwNext++ = 0x00000005;    //   DU_MAIN_3D     DU main:    0.000
    *pdwNext++ = 0xffffbcb4;    //   DV_ORTHO_3D    DV orth:   -1.737
    *pdwNext++ = 0xffff8fe9;    //   DU_ORTHO_3D    DU orth:   -1.562

    // perspective textured, z-buffered polygon (dont-load-color: no need with
textures)
    //
    // NOTE: this example assumes a texture map has been previously loaded
    // into RDRAM that is in 5:6:5 mode and dimensions 128x128

    *pdwNext++ = 0x1003a015;    // DRAW_POLY      im: 003aTexLinTexPrspZ-Buffer
DontLoadColor
    *pdwNext++ = 0x81f48000;    //   X_3D           X value:  500.500
    *pdwNext++ = 0x00640000;    //   Y_3D           Y value:  100.000
    *pdwNext++ = 0x00004923;    //   DX_MAIN_3D     dx main:    0.286
    *pdwNext++ = 0x0000015e;    //   Y_COUNT_3D     y count1: 0 y count2: 350
    *pdwNext++ = 0x0000923d;    //   DWIDTH1_3D     dwidth1:    1.142
    *pdwNext++ = 0x00000000;    //   DWIDTH2_3D     dwidth2:    0.000
    *pdwNext++ = 0x07d00000;    //   Z_3D           Z val:   2000.000
    *pdwNext++ = 0x00000000;    //   DZ_MAIN_3D     Z main:     0.000
    *pdwNext++ = 0x001dd8a0;    //   DZ_ORTHO_3D    Z orth:    29.846
    *pdwNext++ = 0x007f0000;    //   V_3D           V value:  127.000
    *pdwNext++ = 0x007f0000;    //   U_3D           U value:  127.000
    *pdwNext++ = 0xffffa324;    //   DV_MAIN_3D     DV main:   -1.637
    *pdwNext++ = 0x00000000;    //   DU_MAIN_3D     DU main:    0.000
    *pdwNext++ = 0xffff484c;    //   DV_ORTHO_3D    DV orth:   -1.282
    *pdwNext++ = 0xfffea699;    //   DU_ORTHO_3D    DU orth:   -2.651
    *pdwNext++ = 0x00000000;    //   D2V_MAIN_3D    D2V mn :    0.000
    *pdwNext++ = 0x00000000;    //   D2U_MAIN_3D    D2U mn :    0.000
    *pdwNext++ = 0x00000000;    //   D2V_ORTHO_3D   D2V ort:    0.000
    *pdwNext++ = 0x000001d6;    //   D2U_ORTHO_3D   D2U ort:    0.007
    *pdwNext++ = 0x00000086;    //   DV_ORTHO_ADD_3DDV oadd:    0.002
    *pdwNext++ = 0x00000000;    //   DU_ORTHO_ADD_3DDU oadd:    0.000
    *pdwNext++ = 0x68000000;    // IDLE
}
```

3.7.9 Polygons Showing Filtered Texture Mapping

```
/*********************************************************************
*
*  dl_poly_set4()
*
*  draw 2 polygons one with linear texturing, one with filtered
*  texture
*
*********************************************************************/

void dl_poly_set4(DWORD *pdwNext)
{
    DWORD   tex_y_addr;

    // set y offset for texture buffer; x offset is set to 0
    tex_y_addr = (LL_State.BufTextures.Extent.top / 16) << 20;

    *pdwNext++ = 0x18001202;    // WRITE_3D_REG   reg: 4120  TX_CTL0_3D
    *pdwNext++ = 0x00000433;    // - texture is 5:6:5, 128 x 128
    *pdwNext++ = tex_y_addr;    // - tx_xybase reg = texture xy address

    *pdwNext++ = 0x1002800c;    // DRAW_POLY      im: 0028TexLin DontLoadColor
    *pdwNext++ = 0x815e8000;    //   X_3D           X value:  350.500
    *pdwNext++ = 0x00320000;    //   Y_3D           Y value:   50.000
    *pdwNext++ = 0xffff666e;    //   DX_MAIN_3D     dx main:   -1.400
    *pdwNext++ = 0x00c90031;    //   Y_COUNT_3D     y count1: 201 y count2: 49
    *pdwNext++ = 0x0004665f;    //   DWIDTH1_3D     dwidth1:    4.799
    *pdwNext++ = 0xfffee673;    //   DWIDTH2_3D     dwidth2:   -3.800
    *pdwNext++ = 0x00000000;    //   V_3D           V value:    0.000
    *pdwNext++ = 0x007f0000;    //   U_3D           U value:  127.000
    *pdwNext++ = 0x0000820e;    //   DV_MAIN_3D     DV main:    0.508
    *pdwNext++ = 0x00000005;    //   DU_MAIN_3D     DU main:    0.000
    *pdwNext++ = 0xffffe266;    //   DV_ORTHO_3D    DV orth:   -1.884
    *pdwNext++ = 0xffff6c39;    //   DU_ORTHO_3D    DU orth:   -1.423

    *pdwNext++ = 0x18001201;    // WRITE_3D_REG   reg: 4120  TX_CTL0_3D
    *pdwNext++ = 0x00040433;    // - texture filter enable
                                // - preserve 5:6:5
                                // - 128x128

    *pdwNext++ = 0x1002800c;    // DRAW_POLY      im: 0028TexLin DontLoadColor
    *pdwNext++ = 0x82268000;    //   X_3D           X value:  550.500
```

```
        *pdwNext++ = 0x00960000;    //   Y_3D          Y value:  150.000
        *pdwNext++ = 0xffff666e;    //   DX_MAIN_3D    dx main:   -1.400
        *pdwNext++ = 0x00c90031;    //   Y_COUNT_3D    y count1: 201 y count2: 49
        *pdwNext++ = 0x0004665f;    //   DWIDTH1_3D    dwidth1:    4.799
        *pdwNext++ = 0xfffee673;    //   DWIDTH2_3D    dwidth2:   -3.800
        *pdwNext++ = 0x00000000;    //   V_3D          V value:    0.000
        *pdwNext++ = 0x007f0000;    //   U_3D          U value:  127.000
        *pdwNext++ = 0x0000820e;    //   DV_MAIN_3D    DV main:    0.508
        *pdwNext++ = 0x00000005;    //   DU_MAIN_3D    DU main:    0.000
        *pdwNext++ = 0xffffe266;    //   DV_ORTHO_3D   DV orth:   -1.884
        *pdwNext++ = 0xffff6c39;    //   DU_ORTHO_3D   DU orth:   -1.423
        *pdwNext++ = 0x68000000;    // IDLE
    }
```

3.7.10 Initial Width Instruction Modifier

3.7.10.1 Defining the Initial Span

There are two methods for initializing polygon drawing depending on the characteristics of the initial (top) span of the polygon. The CL-GD5464 has opcode modifiers for the following two cases:

Case-1 A polyon ***without*** an initial width (WIDTH1) parameter, nor an opposite width (WIDTH2) parameter. An initial width bias (W1_BIAS) is provided from a 5-bit field in a CONTROL register.

Case-2 A polyon with both WIDTH1 and WIDTH2 is supported (same as warp). The W1_BIAS field is not used in this case.

3.7.11 Double/Multi Buffering

The CRTC_START_ADDR2 register, when written by either the PCI interface (Coprocessor mode) or the WRITE_DEVICE_REGS (Display List mode), arms for transfer into the Primary Screen Start Address registers (located at bits 20:19, CR1D; 18:16, CR1B, 15:8, CRC; 7:0, CRD) at the next frame interval (at VBLANK). This is then loaded into the screen display refresh address counters at VSYNC time. On the next active frame the new display start address is used to refresh the screen. This address is used to refresh the screen until the next write to the Secondary Screen Start Address register.

This register is a read/write. For software double-buffering, three methods can be used. In one case the driver uses a software-polling mechanism looks like the following.

1) At the end of display list, a Load_Long_HIF writes the Secondary Screen Start Address register to the new display buffer refresh address.

2) The next instruction of the display list then starts the clear Z-buffer (for 3D operation) BitBLT operation. When completed, the CL-GD5464 goes to the IDLE state (goes to Coprocessor mode) by reading an IDLE instruction.

3) The software driver reads the primary display address or polls for VSYNC. When either case is seen, the software driver issues a BRANCH to the top of the new display list. This process repeats back to step 1.

A second method uses a hardware interrupt on VSYNC. This eliminates polling of the CPU and thus gives a better frame rate. Triple buffering works the same except that there is no wait to begin drawing is necessary. There is a cleared (or initialized) buffer, ready to begin 3D writes without disruption of the current display buffer.

A third and faster method uses the a WAIT instruction with the wait event being a flag in the Laguna 3D control register. This flag is read/write by the host software driver. The bottom of the display list instruction sequence is as follows:

1) At the end of display list, a WRITE_DEVICE_REGS writes the Secondary Screen Start Address register. The hardware sets an ARM signal on the write and clears the ARM at VBLANK.

2) The next instruction of the display list then starts the clear Z-buffer (3D operation).

3) The next display list instruction is a _WAIT on FLAG¶ when *TBD* in the Control register is a '0' (flag is '0'), the display list execution waits for a one (flag is '1'). This is set by the software driver when the next display list buffer has been loaded into memory and is ready for display list execution.

4) When the wait instruction has a '1' (flag is '1') in the *TBD* Control bit, the next display list instruction is called. The hardware clears (flag is '0') the *TDB* Control bit on the completion of the wait on flag instruction. The next instruction is a _WAIT for NOT_ARM, if double-buffering or a BRANCH to the top of the next display list sequence for triple-buffering. For the double-buffering operation, a BRANCH instruction would follow the WAIT for NOT_ARM instruction. The NOT_ARM indicates that the new Secondary Start Display Address register has been loaded into the Primary Start Display Address register and the hardware is at a frame interval.

In the above process, the advantage is the elimination of waiting by the software driver to instantiate the next branch to top of display list for drawing operation. This allows the CPU to continue display list building without waiting for buffer switch or issue of a branch instruction as in case 1.

There is one caveat of the secondary address register write, there must be a window that disables the write data to complete when the data is written during the same cycle where a VBLANK occurs. This forms a guard band during the write and arm cycle. Also, if the Primary Start Address register is read during this guard band, the read must be suspended till after the update (after VBLANK) of the Primary Display Start Address register.

3.7.12 2D Display Lists

TBD

3.7.13 Z-Collision Detection

The Z-buffer can be used to detect the proximity of two rendered primitive objects considering all three spatial dimensions. Figure 3-17 is a drawing of this method.

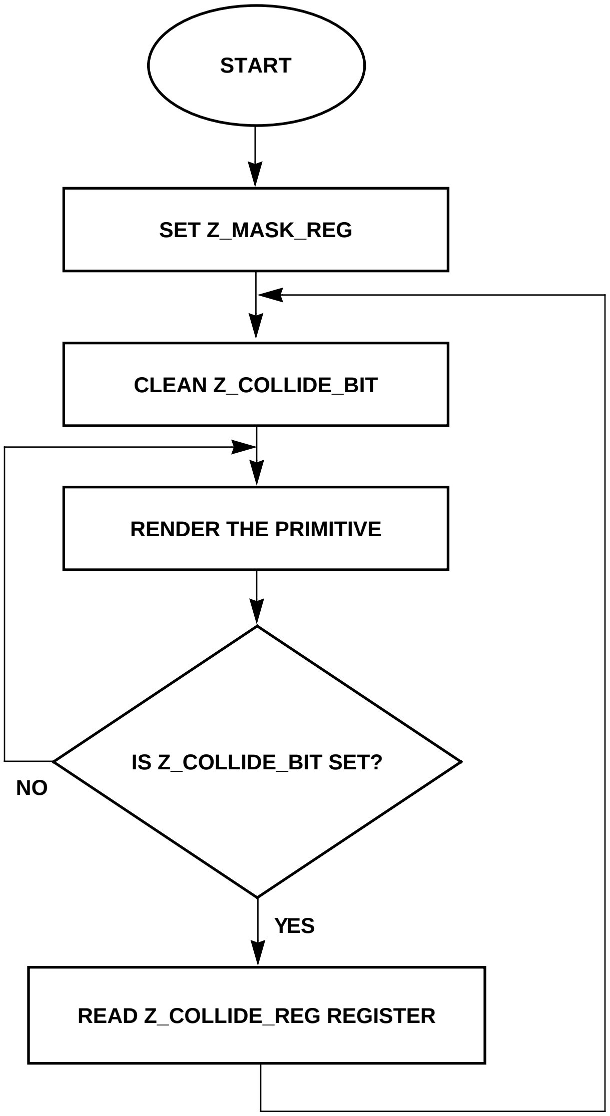

Figure 3-17. Z-Buffer Detect Method

In some application scenarios, it can be desirable to perform a more coarse comparison on the Z values that constitute a collision. This assumes not all of the 16 bits of Z are significant for the purposes of determining proximity. In addition, upper bits of Z can be allocated to identify objects (in a limited object space) if these bits can be masked from the Z-comparison function. The CL-GD5464 provides a means for defining the desired masking by separate masks for the upper and lower bytes of Z. The CONTROL1_3D register at 4110h defines the masks for each of the upper and lower halves of the Z value.

Table 3-77. Z-Value Masks

	15 8	**7 0**
Z-value affected	Bits 15:8	Bits 7:0
Mask field from CONTROL1_3D register	Z-Hit Object Mask bits 31:24	Z-Hit Precision Maskbits 7:0

During the interpolation, Z values from the buffer and interpolator are filtered through the two masks. Then they are compared and if the values are the same, a collision is indicated. The user can then check the collision Z value, and determine from the upper bits which object caused the collision.

Both halves of the Z value are masked before the compare so that the user can:

1) Mask lower 'n' (0 to 8) bits to '0', to allow decrease in Z resolution to assure that deep objects do not pass through each other, but are caught on a larger scale.

2) Mask upper 'n' (0 to 8) bits to '0', to allow for assigning IDs to objects. For example, there can be four different objects at the same Z distance, each with different top two bits in Z (00, 01, 10, 11). They are considered as in the same Z since those bits are masked out, but in the case of collision, buffered Z is copied to ZCOLLIDE register so user can examine it and see the object number that (top two bits) caused the collision.

3.7.14 2D BitBLT Examples

3.7.14.1 Host-to-Screen BitBLTs

For host-to-screen BitBLTs, the source XY location must yield a linear-physical address that is aligned on a 32-bit (dword) boundary. The destination pitch must be a multiple of 8 bytes; however, the destination BitBLT width can be on a pixel boundary.

3.7.14.2 Screen-to-Host BitBLTs

For screen-to-host BitBLTs, the source XY location must yield a linear-physical address that is aligned on a 32-bit (dword) boundary. The source pitch and source BitBLT width must both be a multiple of 8 bytes.

3.7.14.3 Host-to-Host BitBLTs

The host-to-host BitBLTs can be performed. **It is undetermined as yet if they will be s**upported.

Following is a load 'hif' sequence for a 2D BitBLT.

```
// color fill BLT
// OP_OPBGCOLOR             22222222h         #load fill color
// BLTDEF                   1070h             #op1 color source
// DRAWDEF                  00CCh             #rop - srccpy
// OP0_opRDRAM.pt.X         0h                #result at 0
// OP0_opRDRAM.pt.Y         0h                #result at 0
// BLTEXT_EX.pt.X           20h               #extent x is 32
// BLTEXT_EX.pt.Y           20h               #extent y is 32

// Load hif hif instruction
// | load hif | device | byte enables | stall | addresss | # params |
// OP_OPBGCOLOR                22222222h          #load fill color

memory_write_long_word(32'h08000000,{`WRITE_DEVICE_REGS_OPCODE,`ENG2D,4'b000
0,13'h05E4,6'h01});
memory_write_long_word(32'h08000004,32'h22222222);

// BLTDEF                      1070h              #op1 color source

memory_write_long_word(32'h08000008,{`WRITE_DEVICE_REGS_OPCODE,`ENG2D,4'b001
1,13'h0586,6'h01});
memory_write_long_word(32'h0800000C,32'h10700000);

// DRAWDEF                     00CCh              #rop - srccpy

memory_write_long_word(32'h08000010,{`WRITE_DEVICE_REGS_OPCODE,`ENG2D,4'b110
0,13'h0584,6'h01});
memory_write_long_word(32'h08000014,32'h000000CC);

// OP0_opRDRAM.pt.X and Y     0h                 #result at 0,0

memory_write_long_word(32'h08000018,{`WRITE_DEVICE_REGS_OPCODE,`ENG2D,4'b000
0,13'h0520,6'h01});
memory_write_long_word(32'h0800001C,32'h00000000);

// BLTEXT_EX.pt.X and Y       20h                #extent x is 32

memory_write_long_word(32'h08000020,{`WRITE_DEVICE_REGS_OPCODE,`ENG2D,4'b000
0,13'h0700,6'h01});
memory_write_long_word(32'h08000024,32'h00200020);

// Idle
memory_write_long_word(32'h08000028,{`IDLE_OPCODE,27'h000000});
```

3.7.15 Process Synchronization

READ_DEV_REGS allows synchronization between 3D operations and 2D operations by the ability to write a system memory semaphore within a display list.

3.7.16 CL-GD5464 3D Events

The following are the 3D events cur*rently being considered*. These generate some type of notification, either interrupt, internal or external pin, or register bit state change.

CRT_VBLANK — CRT controller vertical blank
CRT_BANK — CRT controller bank switch used for double buffer
CRT_HBLANK — CRT controller horizontal blank
CRT_BEAM_EQ — CRT controller vertical count equal

2D_ENG_IDLE — 2D engine idle strobe to synchronize 3D and 2D requests, that is, BitBLTs.

AUTO_BLT_A — 2D engine auto-BitBLT signal A
AUTO_BLT_B — 2D engine auto-BitBLT signal B
AUTO_BLT_C — 2D engine auto-BitBLT signal C

Z_COMPARE — trigger event on Z compare
C_COMPARE — trigger event on color compare

3.7.17 Self Interrupts

TBD