Overview

1. OVERVIEW

This volume presents an overview of the CL-GD546X and a programming model. The organization of the various relevant memories is discussed, as are the graphical operations. The header files that formally define the registers are listed in the register structures section. The last section covers system-level considerations when using more than one CL-GD546X or when using the CL-GD546X with other adapters.

Reading and Using the Programming Examples

Many of the programming examples used throughout this chapter are extracted from device test scenarios. They generally consist of register names followed by values to be written to those registers. The register names are case insensitive and follow the standard set in the register chapters (the *Laguna VisualMedia™ Accelerators Family — CL-GD546X Volume I (Hardware Reference Manual, Second Edition, September 1996)*) and in the ‘lgregs.h’ file. The values to be written to the registers can be in decimal or hexadecimal format. Hex numbers have a letter ‘h’ suffix on them. Numbers without a suffix are decimal format. The data width (8, 16, 32 bits) can be inferred from the register name by referring to the register description and can also be inferred by the size of the data to be written. Typically, four digit hex numbers and decimal numbers are written as 16-bit words, while eight digit hex values are always written as 32-bit dwords. The pound sign (#) indicates a comment and the text following it is information for the reader.

The examples listed in this chapter are also listed on the Cirrus Logic BBS as part of the ‘SS.c’ program.

1.1 Architectural Overview

This section provides a brief overview of the CL-GD546X graphics system from a programmer’s point of view. It starts with an overall system block diagram (Figure 1-1) that presents the entire graphics system. This is followed with a block diagram (Figure 1-2) showing a conceptual view of the CL-GD546X.

1.1.1 System Block Diagrams

Figure 1-1 shows a graphics subsystem based on the CL-GD546X. The blocks shown as solid lines are in the CL-GD546X device itself. The blocks shown as dotted lines are outside the CL-GD546X device.

The graphics subsystem provides a visible rectangular display mapped onto a rectangular memory space. This memory space is the frame buffer. In the CL-GD546X the frame buffer is implemented using RDRAMs.

On the input side of the frame buffer is the system CPU, a standard SVGA controller, a 2D engine, a 3D engine (CL-GD5464 only), and an enhanced 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 video blanking to the RAMDAC.

The RAMDAC maps memory contents to RGB color values. The frame buffer contains a description of each pixel on the screen. The format of the pixel descriptions in the frame buffer can be palettized, RGB, or YUV. In some cases, the frame buffer contains pixels in two formats.

Also available on the system bus are a set of standard VESA VBE 2.0 BIOS software routines for implementing the VESA SVGA standard, and for initializing and testing the system.

Finally, the CL-GD546X has a set of PCI Configuration registers.

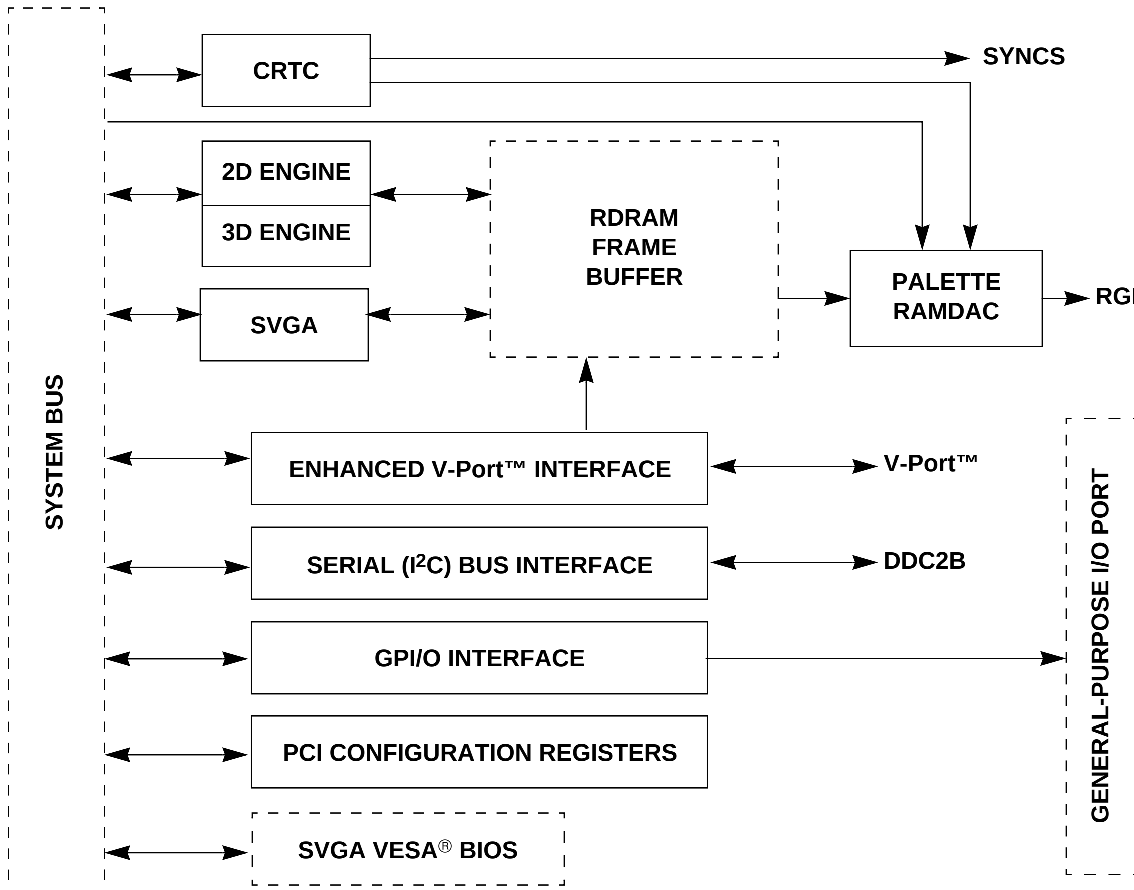


Figure 1-1. Graphics System Based on the CL-GD546X

1.1.2 Internal Architecture

Figure 1-2 is a conceptual diagram of the internal architecture of the CL-GD546X. It is implemented around two internal buses. The host bus is shown in the diagram as HIFBUS and the memory bus is shown in the diagram as RIFBUS. The HIFBUS is connected to the external host 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 sections of the CL-GD546X are connected to one or both of these buses. Each section is described briefly in the following paragraphs.

The 2D graphics engine provides acceleration for three operand raster operations as well as stretch operations. The graphics engine operates synchronously to the memory clock. It can produce one result qword for each memory clock.

The 3D graphics engine provides rendering acceleration for 3D bit-mapped polygon. It is described in the *Chapter 3, “3D Programmer’s Guide*”of this manual.

The VGA module provides VGA compatible host read/write access to the frame buffer as well as VGA functions. In addition, it contains a number of I/O registers and decodes.

The extended read/write module provides host read/write access to a linear frame buffer. This module also contains Memory-Mapped registers and some address decoding. An additional module, the address translate module, provides register decodes for the display path and display FIFO sections.

The display module contains the display pipeline, the YUV-to-RGB color space converter, the color palette, and the DACs. The display FIFO module contains the FIFO and FIFO controller, the display address generator, and the hardware cursor address generator.

The V-Port™ module contains the V-Port data packing and FIFO logic. The V-Port provides a mechanism for transferring video data from a peripheral decoder to the users’ display. Video data is input to the frame buffer and then modified under program control before being output to the monitor. Alternatively, video data can be routed directly to the DAC, bypassing the frame buffer altogether. These two modes are called memory attach, and DAC attach. The video data stream can be in any standard format such as RGB or YUV. Video timing is provided by the external decoder or by the CL-GD546X.

The general-purpose I/O port can be programmed to communicate with a variety of I/O devices. It has a configurable interface specification allowing it to support many bus timings.

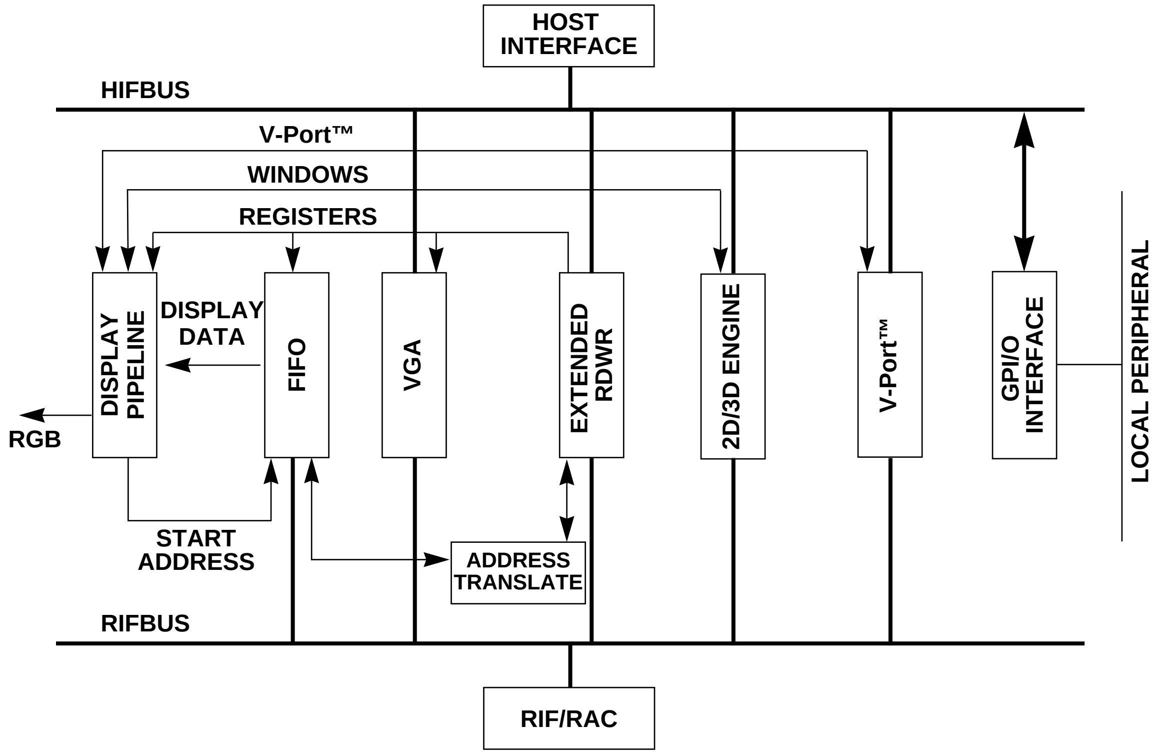


Figure 1-2. CL-GD546X Internal Architecture

1.2 Programming Model

This section covers three of the four CL-GD546X programming models: a super VGA controller, an advanced 2D accelerator, and a flat frame buffer. The 3D programming model is described in the *Chapter 3, "3D Programmer's Guide*"of this specification.

The super VGA controller is programmed like any compatible VGA by I/O registers and the display memory mapped into the A0000h–BFFFFh address range on the host. The 2D accelerator provides hardware assisted drawing operations to the frame buffer and is accessed by Memory-Mapped registers. The flat-frame buffer is controlled by Memory-Mapped registers and is accessed through 8-Mbyte apertures in the host memory address space. Bi-endian support is provided for the 2D accelerator and flat-frame buffer modes of operation.

1.2.1 VGA

The CL-GD546X VGA module provides compatibility with earlier graphics controllers based on VGA. Refer to the *Laguna VisualMedia™ Accelerators Family — CL-GD546X Volume I (Hardware Reference Manual, Second Edition, September 1996,* "VGA Core Registers" chapter for additional information.

The control unit contains the immediate and general registers, the drawing control and the command/data FIFO. The pixel path contains the three operand fetch units (OFU0, OFU1, OFU2), the ROPs unit, the transparency control, and the pixel FIFO. The frame buffer consists of 1, 2, 4 or 8 Mbytes of Rambus RDRAM memory.

Writes to the immediate registers take effect immediately and do not go through the write FIFO. These are used to read 2D engine status and write general control information. Writes to the general registers are queued through the 25-entry write FIFO and are used to set drawing parameters and initiate drawing operations. During a BitBLT operation, color pixel data is loaded into SRAM0. Color and/or monochrome pixel data is loaded into SRAM1 and SRAM2. Monochrome data is converted to color using the foreground and background color registers. Color pixel data is aligned with the destination. Then the three operands are combined in the ROPs unit to form the output pixel data that can be stored in the frame buffer, sent to the host, or stored in one SRAM. If pixel transparency is enabled, SRAM2 is used as the transparency mask. For monochrome masks, the output pixel is written if the corresponding bit in SRAM2 is '1'. For color masks, the output pixel is written if the corresponding pixel in SRAM2 compares with the background color. (The comparison can be programmed to be 'equal' or 'not-equal'.) SRAM0 is typically the destination operand, SRAM1 is typically the source operand, and SRAM2 is typically the pattern operand.

1.2.1.1 2D Frame Buffer

The 2D frame buffer is organized as a rectangular array of packed pixels, with pixel '0,0' at the upper left-hand corner and pixel 'xmax,ymax' at the lower right-hand corner. A rectangular portion of the frame buffer (the display rectangle) is visible on the display device. In the upper left-hand corner is pixel 'xs,ys' and in the lower right-hand corner is pixel 'xe,ye' ($0 \leq xs < xe \leq xmax$, $0 \leq ys < ye \leq ymax$). The display rectangle is shown in relation to the frame buffer in Figure 1-3. The display rectangle is typically aligned to the upper left corner of the display buffer (xs = 0, ys = 0), but can be positioned anywhere on the frame buffer surface. Pixel sizes of 8, 16, 24, and 32 bits are supported (see Section 1.2.2). Pixel addresses given to the 2D engine are always specified in two dimensional 'x,y' coordinates.

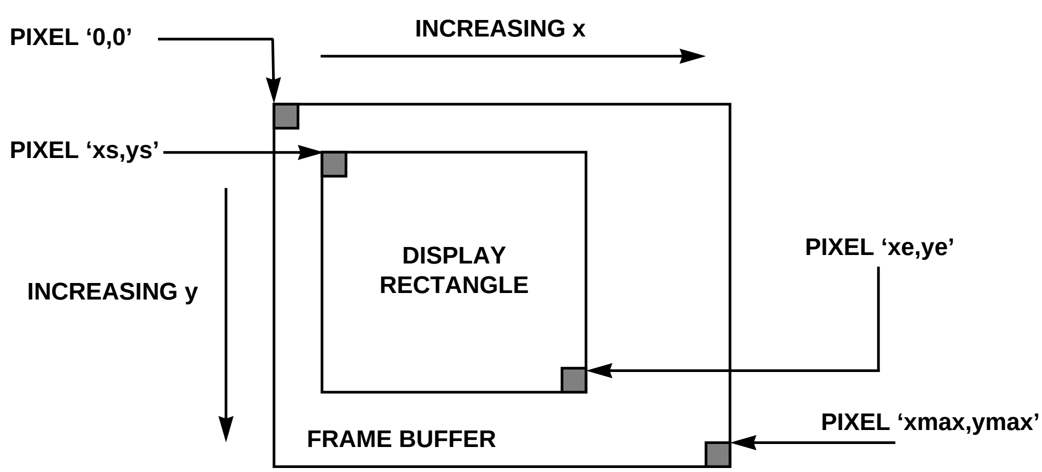


Figure 1-3. 2D Frame Buffer

1.2.1.2 Flat Frame Buffer

The CL-GD546X frame buffer can be accessed directly by software from the host computer. As described earlier, the frame buffer is organized as a rectangular array of packed pixels, with pixel '0,0' at the upper left-hand corner, and pixel 'xmax,ymax' at the lower right-hand corner. (xmax = screen_pitch/pixel_size; ymax = {frame_buffer_size/screen_pitch}/pixel_size; where pixel_size = 1, 2, 3, or 4.) The screen pitch is the number of bytes between vertically adjacent pixels. A rectangular portion of the frame buffer (the raster) is visible on the display device. The upper-left corner of the raster is pixel ‘xs,ys’ and the lower-left corner of the raster is pixel ‘xe,ye’. The upper-left corner of the raster is typically aligned with pixel ‘0,0’, where there is undisplayed off-screen memory to the right of and below the raster. The raster is shown in relation to the frame buffer in Figure 1-3.

Pixel sizes of 8, 16, 24, and 32 bits are supported (see Section 1.2.2). The frame buffer is accessed by host software as a linear array of bytes, words, or double words, with pixel ‘0,0’ located at byte offset 0 in the frame buffer. In general, the byte address of pixel ‘x,y’ is given as:

$$byte\_addr = ((y \times screen\_pitch) + (x \times pixel\_size))$$ **Equation 1-1**

The frame buffer is mapped into the host CPUs address space by Base Address Register 1. It is mapped into four contiguous 8-Mbyte apertures on a 32-Mbyte address boundary. The first aperture directly accesses the frame buffer without byte swapping. The second aperture swaps bytes within words. The third and fourth apertures swap bytes within double-words. Byte swapping is discussed in Section 1.2.3.

1.2.2 Pixels

A pixel is a picture element on the external display surface. Each pixel on the display surface maps uniquely to a pixel data structure in the frame buffer memory array. Pixels in the frame buffer are either 8-, 16-, 24-, or 32-bits wide, and contain data that specifies to the display pipeline how to set the color of its corresponding picture element on the display surface. Color modes define how the display pipeline interprets the contents of the pixel. Most conventional graphics display systems allow one mode for the entire frame buffer (and display surface). The CL-GD546X allows two modes simultaneously, a graphics mode and a video mode. Table 1-1 lists the color modes and indicates the modes that can be paired. (Video windows within the frame buffer can have a Video Color mode different from the Background Graphics Color mode.) The color modes are selected by setting the depth and format fields in the Graphics/Video Format register (MMI/O offset C0h).

Table 1-1. Color Mode Pairing Options

Option Number	Graphics and Video Modes	Concurrent Video Option	Depth Field	Format Field
1	8-bpp Palette	1, 2, 3, 4, 5	00b	000b
2	8-bpp Grayscale	1, 2, 3, 4, 5	00b	001b
3	8-bpp AccuPak[a]	n/a	00b	100b
4	16-bpp 5:6:5	4, 5	01b	010b
5	16-bpp YUV 4:2:2[a]	n/a	01b	101b
6	24-bpp 8:8:8	6, 7	10b	010b
7	24-bpp YUV 4:4:4[a]	n/a	10b	110b
8	32-bpp a:8:8:8	7, 8	11b	010b

[a] Video modes.

1.2.2.1 8-bpp Palettized

Each pixel is specified by one byte of display memory. The value of the byte is used to look up an entry in the color palette. If gamma correction is not enabled, three 6-bit values (one each for Red, Green, and Blue) are passed to the corresponding DACs for conversion to analog. If gamma correction is enabled, three 8-bit values are passed to the DACs for conversion to analog.

When the corresponding depth field is programmed to '00b' and the corresponding format field is programmed to '000b', 8-bpp palettized is selected.

7	6	5	4	3	2	1	0
COLOR VALUE							

1.2.2.2 8-bpp Grayscale

Each pixel is specified by one byte of display memory. If gamma correction is not enabled, the value passes to all three DACs in parallel for conversion to analog. The result is a gray pixel, whose luminance corresponds to the value of the byte.

If gamma correction is enabled, the value is used to look up the three corresponding entries in the color palette. The three 8-bit values pass to the corresponding DACs for conversion to analog. In this case, the hardware behaves like 8-bpp palettized with gamma correction enabled. However, the palette is programmed differently.

When the corresponding depth field is programmed to '00b' and the corresponding format field is programmed to '001b', 8-bpp grayscale is selected.

7	6	5	4	3	2	1	0
GRAYSCALE VALUE							

1.2.2.3 8-bpp AccuPak™

Each group of four adjacent pixels is specified by four adjacent bytes of display memory. The format of each packet is as shown in the following diagram. The luminance of each pixel is specified by five bits; the chrominance values are specified by six bits each. The four pixels share a common chrominance. If gamma correction is not enabled, the packet converts to four pixels of RGB values. This involves interpolation and color space conversion. The RGB values for each pixel pass to the respective DACs for conversion to analog.

If gamma correction is enabled, the packet is converted to four pixels of RGB values. For each pixel, the RGB values are independently used to look up values in the palette. The resulting values are passed to respective DACs for conversion to analog.

8-bpp AccuPak is selected when the corresponding depth field is programmed to ‘00b’ and the corresponding format field is programmed to ‘100b’.

31:27	26:22	21:17	16:12	11:6	5:0
Y3	Y2	Y1	Y0	U0	V0

1.2.2.4 16-bpp 5:6:5

Two adjacent bytes of display memory specify each pixel. The format of each pixel is shown in the following diagram. If gamma correction is not enabled, the three color values are left aligned and passed to the respective DACs for conversion to analog.

If gamma correction is enabled, the three color values are extended to eight bits each by appending zeroes. The resulting three 8-bit values are independently used to look up three values in the color palette. The resulting three 8-bit values are passed to the corresponding DACs for conversion to analog.

16-bpp 5:6:5 is selected by programming the corresponding depth field to ‘01b’ and the corresponding format field to ‘010b’.

15 11	10 5	4 0
RED	GREEN	BLUE

1.2.2.5 16-bpp 5:5:5

Two adjacent bytes of display memory specify each pixel. The format of each pixel is shown in the following diagram. If gamma correction is not enabled, the three color values are left aligned and passed to the respective DACs for conversion to analog.

If gamma correction is enabled, the three color values are extended to eight bits each by appending zeros. The resulting three 8-bit values are independently used to look up three values in the color palette. The resulting three 8-bit values are passed to the corresponding DACs for conversion to analog.

16-bpp 5:5:5 is selected by programming the corresponding depth field to ‘01b’ and the corresponding format field to ‘011b’.

15	14 10	9 5	4 0
	RED	GREEN	BLUE

1.2.2.6 16-bpp YUV 4:2:2

Four adjacent bytes of display memory specify each packet of two adjacent pixels. The format of each packet of two pixels is as shown in the following diagram. If gamma correction is not enabled, the packet converts to two pixels of RGB values. This involves interpolation and color space conversion. The RGB values for each pixel are passed to the DACs for conversion to analog.

If gamma correction is enabled, the packet is converted to two pixels of RGB values. For each pixel the RGB values are independently used to look up values in the palette. The resulting values are passed to respective DACs for conversion to analog.

16-bpp YUV 4:2:2 is selected by programming the corresponding depth field to ‘01b’ and the corresponding format field to ‘101b’.

31 24	23 16	15 8	7 0
Y1	V0	Y0	U0

1.2.2.7 24-bpp 8:8:8

Each pixel is specified by three adjacent bytes of display memory. The format of each pixel is as shown in the following diagram. If gamma correction is not enabled, the color values for each pixel are passed to the respective DACs for conversion to analog.

If gamma correction is enabled, the color values for each pixel are independently used to look up values in the color palette. The results are passed to the respective DACs for conversion to analog.

24-bpp 8:8:8 is selected when the corresponding depth field is programmed to ‘10b’ and the corresponding format field is programmed to ‘010b’.

23 16	15 8	7 0
RED	GREEN	BLUE

1.2.2.8 24-bpp YUV 4:4:4

Each pixel is specified by three adjacent bytes of display memory. The format of each pixel is as shown in the following diagram. If gamma correction is not enabled, the pixel is converted to RGB. This involves color-space conversion.

If gamma correction is enabled, the pixel is converted to RGB. The RGB values are independently used to look up values in the palette. The resulting values are passed to respective DACs for conversion to analog.

24-bpp YUV 4:4:4 is selected when the corresponding depth is programmed to ‘10b’ and the corresponding format is programmed to ‘110b’.

23 16	15 8	7 0
V	Y	U

1.2.2.9 32-bpp a:8:8:8

Each pixel is specified by four adjacent bytes of display memory. The format of each pixel is as shown in the following diagram. If gamma correction is not enabled, the color values for each pixel are passed to the respective DACs for conversion to analog.

If gamma correction is enabled, the color values for each pixel are independently used to look up values in the color palette. The results are passed to the respective DACs for conversion to analog.

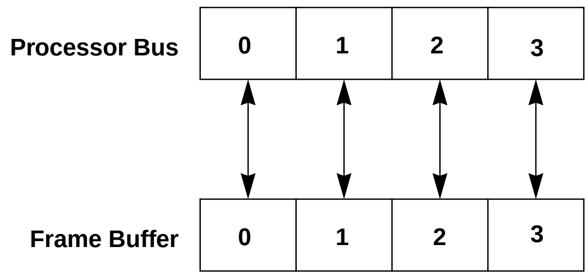
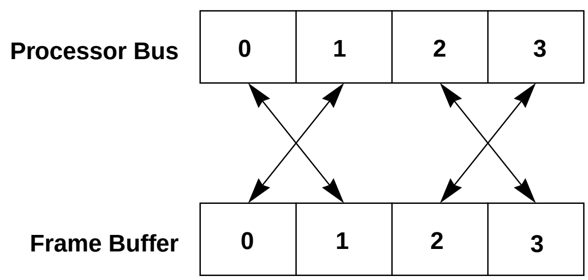
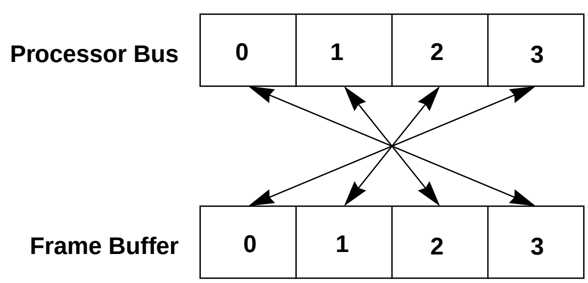
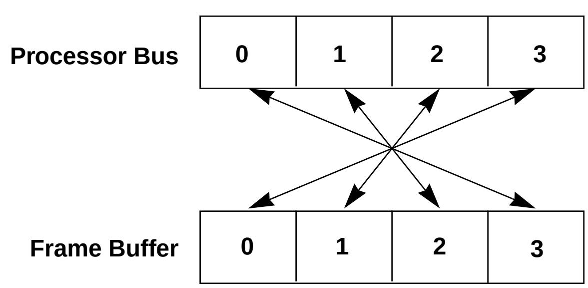
24-bpp a:8:8:8 is selected when the corresponding depth field is programmed to ‘11b’ and the corresponding format field is programmed to ‘010b’.

31 24	23 16	15 8	7 0
ALPHA	RED	GREEN	BLUE

1.2.3 Bi-endian Support

To support operation in a PowerPC PCI system, the CL-GD546X provides byte swapping logic in the host interface. The PowerPC and the PCI bridge perform the swapping required to support byte data. Data types that are larger than a single byte, such as 16-, 24-, and 32-bpp graphics data, require further alignment. Therefore, the frame buffer, host data port, and Memory-Mapped registers each have four different address maps or ‘apertures’. This allows the application software to control the data alignment depending on the pixel depth. In Table 1-2, the first aperture (1) performs no swapping, the second aperture (2) swaps bytes within words, and the third (3) and fourth (4) apertures swap bytes within double-words.

Table 1-2. Byte Swapping for Bi-Endian Support

Aperture	Swap	Diagram
Base address	No swap	
Base address plus 4 Kbytes	Word swap	
Base address plus 8 Kbytes	Dword swap	
Base address plus 12 Kbytes	Dword swap	

1.3 Bus Model

This section describes the programming of the bus model of the CL-GD546X. It has sixteen 32-bit registers that control device mapping and bus configuration. These registers are visible in either PCI configuration space or memory-mapped I/O space. Of the 16 Configuration registers, read or set up the registers listed in Table 1-3.

Table 1-3. PCI Register Offsets

Register	PCI Offset	Memory-Mapped Offset (from BASE_ADDRESS_0)
Command	04h	304h
Sub-Class	0Ah	30Ah
Base_Address_0 (Memory-Mapped)	10h	310h
Base_Address_1 (Frame Buffer)	14h	314h
ROM Base	30h	330h
VS_Control	FCh	3FCh

In PCI Bus mode, register configuration is handled by the system BIOS.

1.3.1 Accessing Registers

The PCI configuration registers are accessed through PCI BIOS calls. (Reference the *PCI BIOS Specification*, Revision 2.1, August 26, 1994.)

The physical address of the Memory-Mapped register is accessed in PCI mode by reading BASE_ADDRESS_0.

1.3.2 Managing the Memory-Mapped Register

To manage the Memory-Mapped register in PCI mode, BASE_ADDRESS_0 is set up to request 4 Kbytes of memory above 1 Mbyte. This is an issue if the programmer wants to program in real-mode, where the programmer must map the registers into the first Mbyte. To solve this problem, the VGA BIOS dynamically maps the registers in the A0000h–BFFFFh range when it needs to use these registers. An example sequence is as following:

```
;---------------------------------------------------------------
;
;             SetMemMap()
;
;             Function: Determine where in Memory to Map our Memory Mapped
Registers
;
;             Entry:      bp - Points to right spot on stack
;             Exit:       bp - 0c has Old Memory Mode Set
;                         bp - 8 has new segment address
;                         bp - 6 has bus and device number
```

```
;                          BAR0 - maps register < 1M in VGA hole <A0000-BFFFF>
;
;----------------------------------------------------------------
MemSeg                     dd          BAR_B800, BAR_B800, BAR_A000, BAR_A000
SetMemMap proc                         near
             push          eax
             push          bx
             push          ecx
             push          dx
             push          di
             push          si
             push          ds

IFDEF PCI
             call          Look4ActPCI                     ; Get PCI Address
             mov           BUSDEVNUM, bx                   ; Save Bus Number
             mov           di, MEMREG                      ; Memory Map
Registers
             mov           ax,PCI_RDW                      ; Read Double
Word
             int           PCI_INT                         ; PCI Interrupt
             mov           eax, ecx                        ; Get Result
ELSE
             mov           dx, VL_BAR                      ; VL BAR
             .386
             in            eax, dx                         ; Get Old Address
ENDIF
             mov           OLDMAPADDR, eax                 ; Save it
             ASSUME        ds:VGA_Data_Area
UBAR:
             push          dx
             push          si
             mov           dx, GFXIDX                      ; Get GR06
             mov           al, 06h
             call          getreg                          ; read it
             xchg          ah, al                          ; data in al
             xor           ah, ah                          ; Clear ah
             and           al, 0Ch                         ; Mask Bits
             mov           si, OFFSET MemSeg
             add           si, ax
             mov           eax, cs:[si]
             pop           si
```

```
                pop         dx
BAR_FIX:
IFDEF PCI
                mov         ecx, eax                            ; Get Value to
Write
                mov         ax, PCI_WDW                         ; Write Double
Word
                int         PCI_INT                             ; Do it
                mov         eax, ecx                            ; Restore eax
ELSE
                out         dx, eax
ENDIF
                shr         eax, 4                              ; Get Segment
                mov         MEMMAPSEG, ax                       ; Save Offset

                pop         ds
                pop         si
                pop         di
                pop         dx
                pop         ecx
                pop         bx
                pop         eax
                ret
SetMemMap endp
;-----------------------------------------------------------------
```

To restore the address, use the following example code:

```
;-----------------------------------------------------------------
;
;               ClrMemMap()
;
;               Function:                   Clears Memory Map
;               Entry:                      bp - Points to right spot on
stack
;                                           bp - 8 Memory Mapped Sement
;                                           bp - c Old Memory Mapped
Sement
;               Exit:                       BAR0 - Points to Old Memory
Mapped Address
;
;-----------------------------------------------------------------
ClrMemMap proc                  near
                push        eax                                 ; Save eax
```

```
                push        ds                                      ; Save ds

    IFDEF PCI
                push        si
                push        di
                push        bx
                push        ecx


                call        Look4ActPCI                             ; Get PCI Address
                mov         BUSDEVNUM, bx                           ; Save Bus Number
                mov         di, MEMREG                              ; Memory Map
    Registers
                mov         ax,PCI_WDW                              ; Read Double
    Word
                mov         ecx, OLDMAPADDR                         ; Get Result
                int         PCI_INT                                 ; PCI Interrupt
                pop         ecx
                pop         bx
                pop         di
                pop         si
    ELSE
                push        dx
                mov         dx, VL_BAR
                mov         eax, OLDMAPADDR                         ; Get Old Value
                out         dx, eax
                pop         dx
    ENDIF

                pop         ds                                      ; Restore ds
                pop         eax                                     ; Restore eax
                ret
    ClrMemMap endp
```


1.3.3 Initializing Configuration Registers

The registers listed in Table 1-4 are initialized at POST time.

Table 1-4. Initializing Registers at POST Time

Register	PCI Mode
Command	The PCI System BIOS initializes.
Base_Address_0	The PCI System BIOS initializes to above 1 Mbyte.
Base_Address_1	The PCI System BIOS initializes to above 1 Mbyte.
ROM Base	The PCI System BIOS initializes to C0000h.
VS_Control	The VGA BIOS initializes to 01003401h.

1.3.4 VGA Sleep Mode

To disable memory and I/O access, the CL-GD546X Command register is initialized to zero by the hardware reset. The CL-GD546X does not respond to any memory or I/O accesses after reset in PCI mode. This is used to enable or disable the CL-GD546X VGA in PCI mode.

1.4 Memory Organization

This section covers the organization of the memories and register spaces in the CL-GD546X.

The four memory spaces are shown on the left of Figure 1-4. The Frame Buffer can be addressed using four apertures of 8 Mbytes each. The base address is programmed into the PCI Base 1 register. The four apertures provide byte swapping. The memory-mapped I/O provides access to most registers with two sets of four apertures of 4 Kbytes each (32 Kbytes total). The base address is programmed into the PCI Base 0 register. The expansion ROM is addressed with a single 32-Kbyte space. The base address is programmed into the PCI Expansion ROM Base Address register. The standard VGA window into the frame buffer is fixed at A0000h–BFFFFh.

The standard VGA registers are accessible at 3B0h–3BBh, 3C0h–3C2, and 3C4h–3DFh.

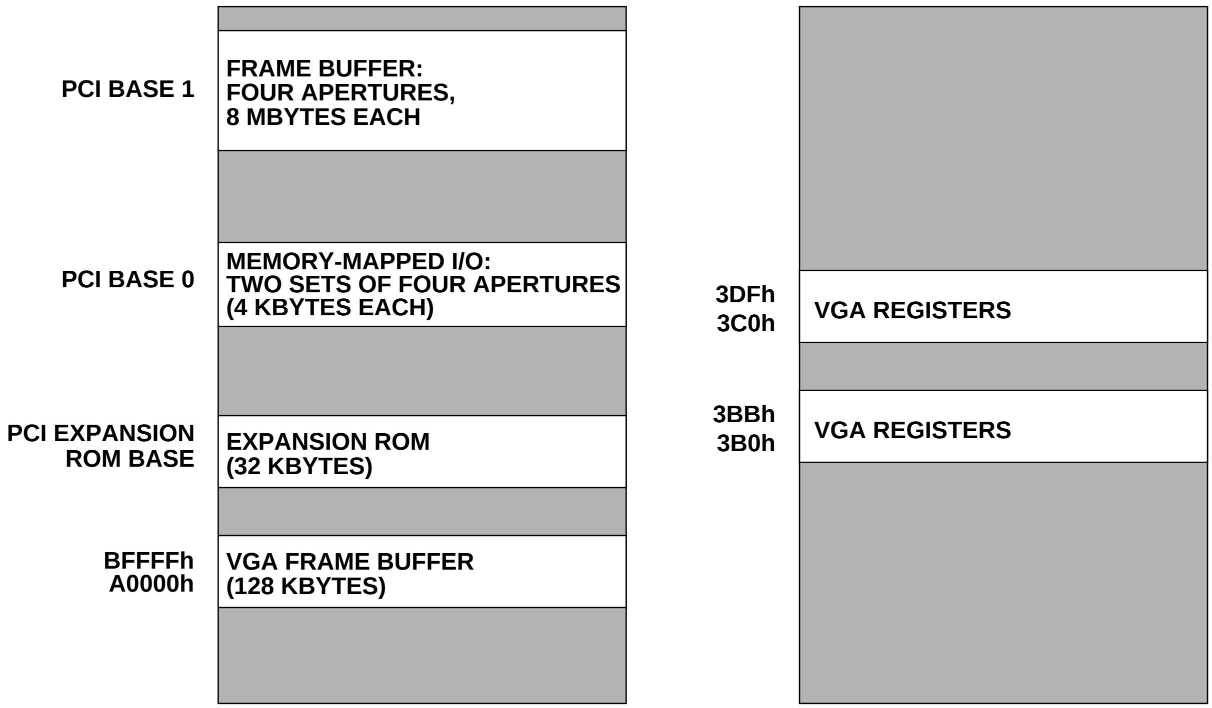


Figure 1-4. Memory and I/O Spaces (Not to Scale)

1.4.1 Frame Buffer Linear Addressing

The frame buffer can be accessed by the host as a linear string with an extent of 32 Mbytes. Program the base address of the frame buffer into PCI14, Frame Buffer Base Address register (described in the *Laguna VisualMedia™ Accelerators Family — CL-GD546X Volume I (Hardware Reference Manual, Second Edition, September 1996)*). This is the Frame Buffer Base Address register.

The 32-Mbyte address space is made of four apertures. Each aperture allows direct access to all 8 Mbytes of frame buffer memory possible on the CL-GD546X. The difference in the four apertures is the way bytes are swapped. The last two apertures are the same. Refer to Table 1-2 on page 1-12.

1.4.2 Frame Buffer Addressing: VGA Compatibility

The CL-GD546X is capable of addressing up to 8 Mbytes of display memory. In the DOS environment there are 128 Kbytes of memory space at A0000h–BFFFFh reserved for display memory. Since the VGA has to share this memory with MDA, Hercules, or CGA, it is left with the single 64-Kbyte segment from A0000h–AFFFFh.

The CL-GD546X supports single- and dual-paging display memory addressing schemes that allow mapping of two 32-Kbyte segments, or one 64-Kbyte segment of display memory into CPU address space. Only the first Mbyte is accessible with VGA compatibility addressing. Byte swapping is not available with VGA compatibility addressing. This addressing mode is retained only for compatibility with the Alpine family. Readers interested in this topic should refer to the *Alpine™ VGA Family Technical Reference Manual*.

1.4.3 Linear and Tiled Modes

The frame buffer memory organization is available in either linear or tiled format. The frame buffer interface logic handles the linear/tiled translation and the translation is transparent to the programmer. Reading and writing data through the linear frame buffer aperture does not require the programmer to make any special allowances for the tiled or linear configuration of memory. Since graphics source objects (patterns, fonts, masks) can be placed on tiled boundaries and accessed in a minimal number of page-crossing fetches, there are significant performance enhancements that are realized by using tiled mode. Statistically, most BitBLTs (such as text) fit within small rectangles that are smaller than a tile, and do not cause excessive page breaks during operand fetching and result writes.

The Rambus memory is physically organized as a set of 2048-byte pages. The number of pages is dependent on how much physical memory is in the display adapter (1 Mbyte, 2 Mbytes, and so on up to 8 Mbytes). The memory pages can be mapped to the frame buffer as lines (Linear mode), or as tiles (Tiled mode). By maximizing the number of fetches from a page once it has been enabled, minimum memory fetch latency can be achieved. The pages called tiles, can be organized in one of the three ways:

1) 2048 by 1 is referred to as Linear mode.
2) 128 by 16 is referred to as Narrow Tiled mode.
3) 256 by 8 is referred to as Wide Tiled mode.

For example, if the programmer configures for 640 $\times$ 480 at 16-bpp no interleave (IL = 1), the X,Y to memory address translations, for the first tile in the frame buffer, are as designated in Table 1-5.

Table 1-5. Tile Modes

Address	Linear (X,Y)	Narrow (X,Y)	Wide (X,Y)
0	0,0	0,0	0,0
126	63,0	63,0	63,0
128	64,0	0,1	64,0
254	126,0	63,1	126,0
256	128,0		0,1
510	255,0		126,1
1278	639,0		
1280	0,1		
2046	383,1	63,15	126,7

There are restrictions on the use of Tiled mode. Do not use the tiled configuration in VGA modes. It can only be used in accelerated modes that rely on the 2D engine. Programmers relying on the Cirrus Logic BIOS do not have to concern themselves with these implications as the BIOS takes care of managing the tiling configuration during a mode switch. After making a BIOS mode switch call Set_Video_Mode, the programmer should call Enable_Tiled_Mode to configure for Tiled mode. If the number of tiles per scanline is not evenly divisible into the total number of tiles in the memory, then tiling creates an artifact in the off-screen memory map. This causes the last row of tiles to be incomplete. This last row of tiles in memory can be accessed by XY coordinates as normal. It can still be accessed by the linear frame buffer interface. However, use caution as the 'missing' tiles creates a rectangular gap in the address map in the lower-right corner of the frame buffer (see Section 1.4.3.3).

1.4.3.1 Programming Considerations

The programming of Linear versus Tiled mode is covered in the Chapter 5, "*System Operation*". It is also possible to request the VGA BIOS to do it. This is done by the following code sequence:

```
Mode Switch
<i.e. INT 10, ah=0, al = Valid Cirrus Logic Mode Number>
Tile Mode
<i.e. INT 10, ah=12h, bl=b3h >
```

1.4.3.2 Linear versus Tiled Restrictions

When deciding whether to use linear versus tiled formats, be aware of certain restrictions:

1) Optimal performance of the 2D engine is achieved by using the Tiled mode.
2) Packed Pixel VGA modes are not compatible with the Tiled mode.
3) In some modes organized with the Tiled mode, not all of off-screen memory is simple to use by the linear frame buffer interface, due to translation optimizations (see Section 1.4.3.3).

1.4.3.3 Off-Screen Memory Problems

Some memory addresses near the maximum frame buffer address translate to invalid physical memory addresses in Tiled mode. The formula for calculating the lowest invalid linear address is as following:

$$lowest\_invalid\_linear\_addr = ((n\_rows \times il\_row\_height) + (rem\_tiles \times tile\_width))$$

Where

n_rows	is the number of complete interleaved tile rows (not counting any partial row of tiles at the end of the frame buffer.
il_row_height	is the interleaved row height = (2048 / tile_width) x interleave [1, 2, or 4].
rem_tiles	is the number of (interleaved groups of) tiles in the last partial row of tiles (can be zero).
tile_width	is the tile width in bytes [128 or 256].

For example, use a 1-Mbyte frame buffer configured for 640 × 480 at 16 bpp, n_tiles = 5, interleave = 0 no interleave, and wide_tile = 1 (256 bytes). If the programmer is not using tiled memory, the maximum physical linear address would be FFFFFh. If the programmer uses tiles, there are 512 tiles to allocate in rows of five per scanline. This means that tiles 0–509 form a simple contiguous region up to address FEFFFh. Tiles 510 and 511 form a partial row starting at FF000h–FF1FFh, continuing at FF500h–FF6FFh all the way to 101300h–1014FFh. Addresses FF200h–FF4FFh, FF700h–FF9FFh, and so on do not correspond to physical memory. Data writes to these addresses and above are lost. Data reads from these addresses are undefined.

The following function documents the algorithm for calculating the off-screen areas. This includes the size and location of the last row rectangle, which can be an artifact of tiling. The function refers to four rectangles that comprise the frame buffer memory. These four rectangles are the visible rectangle displayed in the upper-left portion of memory, the right rectangle, the bottom rectangle, and the extra rectangle. The right and left rectangles are the classically available regions of off-screen memory. The extra rectangle is an artifact of tiling that appears to dangle from the lower left of memory in some configurations.

```
/*****************************************************************
*****************************************************************
*
*             *****************************************
*             * Copyright (c) 1995, Cirrus Logic, Inc. *
*             *            All Rights Reserved         *
*             *****************************************
*
* PROJECT:                             CL-GD546X
*
* FILE:                                5462mem.c
*
* DESCRIPTION:                         Calculates off screen areas based on
memory
*                                      configuration.
*
****************************************************************/

/*------------------- DEFINES ----------------------------------*/
#define TRUE -1
#define FALSE 0
#define ILLEGALTILECONFIG -1

/*------------------- TYPES ------------------------------------*/
typedef unsigned long ULONG;

/*********************************************************
* FUNCTION NAME:                       Lookup_Tiles_Per_Line()
* DESCRIPTION:                         lookup tiles per line based on tile width
*                                      and X extent
*********************************************************/
int Lookup_Tiles_Per_Line(ulTileWidth, ulXExtentBytes)
             ULONG                     ulTileWidth;
             ULONG                     ulXExtentBytes;
{
             ULONG                     TilesPerLine = ILLEGALTILECONFIG;
```

```
if (128 == ulTileWidth) {
            if (4096 < ulXExtentBytes)
                        return (ILLEGALTILECONFIG);
            if (4096 >= ulXExtentBytes          TilesPerLine = 32;
            if (3328 >= ulXExtentBytes          TilesPerLine = 26;
            if (2560 >= ulXExtentBytes          TilesPerLine = 20;
            if (2048 >= ulXExtentBytes          TilesPerLine = 16;
            if (1664 >= ulXExtentBytes          TilesPerLine = 13;
            if (1280 >= ulXExtentBytes          TilesPerLine = 10;
            if (1024 >= ulXExtentBytes          TilesPerLine = 8;
            if (640  >= ulXExtentBytes          TilesPerLine = 5;
            } else if (256 == ulTileWidth) {
            if (2 * 4096 < ulXExtentBytes)
                        return (ILLEGALTILECONFIG);
            if (2 * 4096 >= ulXExtentBytes      TilesPerLine = 32;
            if (2 * 3328 >= ulXExtentBytes      TilesPerLine = 26;
            if (2 * 2560 >= ulXExtentBytes      TilesPerLine = 20;
            if (2 * 2048 >= ulXExtentBytes      TilesPerLine = 16;
            if (2 * 1664 >= ulXExtentBytes      TilesPerLine = 13;
            if (2 * 1280 >= ulXExtentBytes      TilesPerLine = 10;
            if (2 * 1024 >= ulXExtentBytes      TilesPerLine = 8;
            if (2 * 640  >= ulXExtentBytes      TilesPerLine = 5;

            } else         /* ulTileWidth = 2048, aka untiled */

return (ILLEGALTILECONFIG);
return TilesPerLine;
}

/************************************************************
* FUNCTION NAME:                  Legal_Interleave()
* DESCRIPTION:                    validate interleave against memory size.
************************************************************/
int
Legal_Interleave(MegaBytes, Interleave)
            int                    MegaBytes;
            int                    Interleave;
{
            // Legal Memory Size and Interleave combinations are:
            // IL = 1, MEG = 1, 2, 3, 4, 5, 6, 7, 8
            // IL = 2, MEG = 2, 4, 6, 8
```



```
                // IL = 4, MEG = 4, 8

                return ((MegaBytes % Interleave) ? FALSE : TRUE);
}

/************************************************************
* FUNCTION NAME:                    main()
* DESCRIPTION:                      Calculate off screen areas based on memory
*                                   configuration.
************************************************************/
void main(void)
{
                ULONG               MegaBytesInstalled = -1;
                ULONG               MemorySizeInBytes = -1;
                ULONG               MemoryInterleave = -1;
                ULONG               TileWidth = -1;
                ULONG               BytesPerTile = 2048;
                ULONG               BitsPerPixel = -1;
                ULONG               X_Extent = -1;
                ULONG               Y_Extent = -1;
                ULONG               X_Extent_Bytes = -1;
                ULONG               TileHeightInLines = -1;
                ULONG               TilesPerLine = -1;
                ULONG               BytesPerLine = -1;
                ULONG               ExtraMemory = -1;
                ULONG               AvailableMemory = -1;
                ULONG               NumberOfRows = -1;
                ULONG               VisibleRectangle = -1;
                ULONG               VisibleRectangle_SizeInBytes = 0;
                ULONG               VisibleRectangleMemReqd = 0;
                ULONG               RightHandRectangle = -1;
                ULONG               RightHandRectangle_x0 = 0;
                ULONG               RightHandRectangle_y0 = 0;
                ULONG               RightHandRectangle_X_Extent = 0;
                ULONG               RightHandRectangle_Y_Extent = 0;
                ULONG               RightHandRectangle_SizeInBytes = 0;
                ULONG               BottomRectangle = -1;
                ULONG               BottomRectangle_x0 = 0;
                ULONG               BottomRectangle_y0 = 0;
                ULONG               BottomRectangle_X_Extent = 0;
                ULONG               BottomRectangle_Y_Extent = 0;
                ULONG               BottomRectangle_SizeInBytes = 0;
```

```
            ULONG                   ExtraRectangle = -1;
            ULONG                   ExtraRectangle_x0 = 0;
            ULONG                   ExtraRectangle_y0 = 0;
            ULONG                   ExtraRectangle_X_Extent = 0;
            ULONG                   ExtraRectangle_Y_Extent = 0;
            ULONG                   ExtraRectangle_SizeInBytes = 0;
            ULONG                   ExtraRectangle_NumberOfTiles = 0;

            int                     im, memsizes[] = {8, 4, 3, 2, 1, 0};

            int                     xi = 0;
            ULONG                   XE[5] = {640, 800, 1024, 1280, 1600};
            ULONG                   YE[5] = {480, 600, 768, 1024, 1200};

for (xi = 0;  xi < 5;  xi++) {
            /* 640, 800, 1024, 1280, 1600 */
            X_Extent = XE[xi];
            Y_Extent = YE[xi];

for (BitsPerPixel = 8; BitsPerPixel <= 32;
            BitsPerPixel += 8) {
            /* 8, 16, 24, 32 */

for (im = 0; memsizes[im] != 0; im++) {
            MegaBytesInstalled = memsizes[im];

for (TileWidth= 128; TileWidth<= 256; TileWidth += 128) {
            /* 256, 128 */

for (MemoryInterleave = 1; MemoryInterleave <= 4;
            MemoryInterleave *= 2) {
            /* 1, 2, 4 */

            VisibleRectangle = FALSE;
            VisibleRectangle_SizeInBytes = 0;
            VisibleRectangleMemReqd = 0;
            RightHandRectangle = FALSE;
            RightHandRectangle_x0 = 0;
            RightHandRectangle_y0 = 0;
            RightHandRectangle_X_Extent = 0;
            RightHandRectangle_Y_Extent = 0;
            RightHandRectangle_SizeInBytes = 0;
```



```
                    BottomRectangle = FALSE;
                    BottomRectangle_x0 = 0;
                    BottomRectangle_y0 = 0;
                    BottomRectangle_X_Extent = 0;
                    BottomRectangle_Y_Extent = 0;
                    BottomRectangle_SizeInBytes = 0;
                    ExtraRectangle = FALSE;
                    ExtraRectangle_x0 = 0;
                    ExtraRectangle_y0 = 0;
                    ExtraRectangle_X_Extent = 0;
                    ExtraRectangle_Y_Extent = 0;
                    ExtraRectangle_SizeInBytes = 0;
                    ExtraRectangle_NumberOfTiles = 0;

                    MemorySizeInBytes =                 MegaBytesInstalled *
                                                        1024l * 1024l;

                    X_Extent_Bytes =                    X_Extent * (BitsPerPixel /
      8);
                    VisibleRectangle_SizeInBytes = X_Extent_Bytes *
                                                        Y_Extent;

                    TileHeightInLines =                 BytesPerTile / TileWidth;
                    TilesPerLine =                      Lookup_Tiles_Per_Line(
                                                        TileWidth, X_Extent_Bytes);

                    BytesPerLine =                      TilesPerLine * TileWidth;
                    VisibleRectangleMemReqd = BytesPerLine * Y_Extent;

                    if ((TilesPerLine != ILLEGALTILECONFIG) &&
                                // too big to fit in maximum tiles per line
                                // (MemorySizeInBytes > VisibleRectangleMemReqd) &&
                                // not enough memory (Legal_Interleave
                                // (MegaBytesInstalled, MemoryInterleave))
                                // illegal interleave/memsize combination
                                ) {

                                // Calculate if any memory exists in “extra”
                                // rectangle at lower left corner of memory space.
                                ExtraMemory =
      MemorySizeInBytes %
                                                                    (TilesPerLine *
                                                                    BytesPerTile *
```

```
MemoryInterleave);
                         AvailableMemory =
MemorySizeInBytes -
                                                                    ExtraMemory;
                         NumberOfRows =         AvailableMemory /
                                                BytesPerLine;
                         VisibleRectangle =     (VisibleRectangle-
                                                MemReqd < MemorySizeIn-
                                                Bytes) ? TRUE : FALSE;

                         RightHandRectangle =   (X_Extent_Bytes <
                                                BytesPerLine) ?
                                                TRUE : FALSE;
                         if (RightHandRectangle) {
                                    RightHandRectangle_x0 = X_Extent_Bytes;
                                    RightHandRectangle_y0 = 0;
                                    RightHandRectangle_X_Extent =
                                                BytesPerLine -
X_Extent_Bytes;
                                    RightHandRectangle_Y_Extent = Y_Extent;
                                    RightHandRectangle_SizeInBytes =
                                                RightHandRectangle_X_Extent
*
                                                RightHandRectangle_Y_Extent;
             }
             BottomRectangle = (VisibleRectangleMemReqd <
                         MemorySizeInBytes) ? TRUE : FALSE;

             if (BottomRectangle) {
                         BottomRectangle_x0 = 0;
                         BottomRectangle_y0 = Y_Extent;
                         BottomRectangle_X_Extent = BytesPerLine;
                         BottomRectangle_Y_Extent =
                                    (AvailableMemory-
VisibleRectangleMemReqd) /
                                    BytesPerLine;
                         BottomRectangle_SizeInBytes =
                                    BottomRectangle_X_Extent *
                                    BottomRectangle_Y_Extent;
             }
             ExtraRectangle = (ExtraMemory > 0) ?
                         TRUE : FALSE;
             if (ExtraRectangle) {
```



```
                        ExtraRectangle_x0 = 0;
                        ExtraRectangle_y0 =
                                Y_Extent + BottomRectangle_Y_Extent;
                        ExtraRectangle_Y_Extent = MemoryInterleave *
                                TileHeightInLines;
                        ExtraRectangle_X_Extent = ExtraMemory /
                                ExtraRectangle_Y_Extent;
                        ExtraRectangle_SizeInBytes =
                                ExtraRectangle_Y_Extent *
                                ExtraRectangle_X_Extent;
                        ExtraRectangle_NumberOfTiles = ExtraMemory /
                        BytesPerTile;
                }
    }           //endif (valid config)
    } } } } }
    }
```

1.4.4 Registers

The CL-GD546X Memory-Mapped registers are listed in Table 1-6.

Table 1-6. CL-GD546X Memory-Mapped Registers

	Byte Lane			
Offset	**3**	**2**	**1**	**0**
0				Horizontal Total
4				Horizontal Display End
8				Horizontal Blanking Start
C				Horizontal Blanking End
10				Horizontal Sync Start
14				Horizontal Sync End
18				Vertical Total
1C				Overflow
20				Screen A Preset Row Scan
24				Character Cell Height
28				Text Cursor Start
2C				Text Cursor End
30				Screen Start Address High
34				Screen Start Address Low
38				Text Cursor Location High
3C				Text Cursor Location Low
40				Vertical Sync Start
44				Vertical Sync End
48				Vertical Display End
4C				Offset
50				Underline Row Scanline
54				Vertical Blanking Start
58				Vertical Blanking End
5C				Mode Control
60				Line Compare
64				Interlace End
68				Miscellaneous Control
6C				Extended Display Controls
70				

Table 1-6. CL-GD546X Memory-Mapped Registers *(cont.)*

	Byte Lane			
Offset	**3**	**2**	**1**	**0**
74				Screen Start Addr Extension
78				Vertical Total Extension
7C				
80				Miscellaneous Output
84				VCLK3 Numerator
88				VCLK3 Denominator
8C				MCLK Select
90				Signature Generator Control
94				Signature Result Low Byte
98				Signature Result High Byte
9C				
A0				Palette Mask
A4				Palette Read Address/State
A8				Palette Write Address
AC				Palette Data
B0				Palette State
B4				External Overlay
B8				Color Key
BC				Color Key Mask
C0			Format	
C4				
C8	START_BLT_3	STOP_BLT_3		
CC	Y_START_2		X_START_2	
D0	Y_END_2		X_END_2	
D4			START_BLT_2	STOP_BLT_2
D8				
DC	START_BLT_1	STOP_BLT_1		
E0	CURSOR_Y		CURSOR_X	
E4	CURSOR_CONTROL		CURSOR_PRESET	
E8	Display Threshold and Tiling		Cursor Location	
EC				

Table 1-6. CL-GD546X Memory-Mapped Registers *(cont.)*

Byte Lane				
Offset	**3**	**2**	**1**	**0**
F0	TEST_HT		TEST	
F4			TEST_VT	
F8	Reserved for Test			
FC	Reserved for Test			
100	X Start (Even)		X Start (Odd)	
104	Y Start (Even)		Y Start (Odd)	
108		V-Port Height	V-Port Width	
10C			V-Port Mode 0	
110:1FC				
200	RAC Control		RIF Control	
204			Rambus Transaction	
208:23C				
240:27C	Rambus Data			
280			Serial Port	
284:2FC				
300	Device ID		Vendor ID	
304	Status		Command	
308			Class Code	Revision ID
30C		Header Type		
310	Base Address 0			
314	Base Address 1			
318:328				
32C	Subsystem ID		Subsystem Vendor ID	
330	Expansion ROM Base			
334:338				
33C			Interrupt Pin	Interrupt Line
340:3F4				
3F8	VGA_Shadow			
3FC	VS_Control			
400	CONTROL		STATUS	
404	TILE_CTRL	TIMEOUT	OFFSET_2D	QFREEE

Table 1-6. CL-GD546X Memory-Mapped Registers *(cont.)*

	Byte Lane			
Offset	**3**	**2**	**1**	**0**
408	RESIZEA_opRDRAM			
40C	RESIZEB_opRDRAM			
410	RESIZEC_opRDRAM			
414:47C				
480	COMMAND			
484:4FC				
500	MAJY		MINY	
504			ACCUMY	
508	MAJX		MINX	
50C	LNCNTL		ACCUMX	
510	CHROMA_CNTL		STRETCH_CNTL	
514:51C				
520	OP0_opRDRAM			
524	OP0_opMRDRAM			
528	PATOFF		OP0_opSRAM	
52C:53C				
540	OP1_opRDRAM			
544	OP1_opMRDRAM			
548	OP1_opMSRAM		OP1_opSRAM	
54C:55C				
560	OP2_opRDRAM			
564	OP2_opMRDRAM			
568	OP2_opMSRAM		OP2_opSRAM	
56C:57C				
580	SHRINKINC		SRCX	
584	BLTDEF		DRAWDEF	
588				MONOQW
58C:5DC				
5E0	OP_opFGCOLOR / ALPHA_{A,B}			
5E4	OP_opBGCOLOR			

Table 1-6. CL-GD546X Memory-Mapped Registers *(cont.)*

	Byte Lane			
Offset	**3**	**2**	**1**	**0**
5E8	BITMASK			
5EC			TAGMASK	
5F0	CHROMA_LOWER			
5F4	CHROMA_UPPER			
5F8:5FC				
600	BLTEXT_XEX			
604	BLTEXTFF_XEX			
608	BLTEXTR_XEX			
60C			BLTEXT_LN_EX	
610:61C				
620	MBLTEXT_XEX			
624				
628	MBLTEXTR_XEX			
62C:6FC				
700	BLTEXT_EX			
704	BLTEXTFF_EX			
708	BLTEXTR_EX			
70C:71C				
720	MBLTEXT_EX			
724				
728	MBLTEXTR_EX			
72C:7FC				

1.4.4.1 Memory-Mapped I/O

Most registers in the CL-GD546X are accessed using memory-mapped I/O. There is a 16-Kbyte extent, comprising four 4-Kbyte apertures. 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 chapter.

The four apertures of the memory-mapped I/O address space control byte swapping. This works just the same as frame buffer access.

1.4.4.2 I/O Mapped Registers

The VGA Core registers are accessible using normal I/O. These registers are 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. Nearly all the I/O addresses are standard VGA. A few registers are accessible both in the memory space and the I/O space. Most of these registers are in the CRT Controller and each have addresses in the appropriate columns of the summary tables in each chapter.

1.4.5 SRAM

The CL-GD546X uses SRAM caches and queues extensively to enhance performance. It increases parallel processing and minimizes frame-buffer memory accesses. The primary purpose of the operand SRAMs is to optimize hardware performance. The programmer can explicitly use the SRAM for improving performance, but should exercise caution.

The SRAMs of interest, to the programmer, are the three 128-byte caches (two 128-byte and one 1024-byte on the CL-GD546X) associated with each operand fetch unit. The BitBLT engine automatically caches fetched data from the frame buffer or host in these SRAMs and performs the raster operations. Under software control, the programmer can specify these SRAMs as source, pattern, or destination operands. The programmer can also specify these SRAMs as the result of the raster operation.

The programmer can specify the SRAM as source or result of a BitBLT by setting up the BLTDEF register. The SRAM cannot be a source or destination for a stretch or shrink BitBLT. The BLTDEF register OP1 and OP2 fields can be programmed for monochrome or color data that is fetched from SRAM. The fetched data can be pattern data if it is monochrome, 8- or 16-bit color. Color pattern data that is 24- and 32-bit does not fit into the 128 bytes of SRAM assigned to each operand fetch unit. The OP2 SRAM data can be used for transparency masking by setting the transparency control bits in the DRAWDEF register.

The OPn_opSRAM registers are pixel pointers when written, and byte pointers when read back. The OPn_opMSRAM registers are bit pointers and read back as written. The OPn_opSRAM and OPn_opMSRAM registers are physically the same registers within the CL-GD546X. Values that are written to one access name affect the values read back by the other access name.

Any of the three SRAMs can be designated as the result of a BitBLT operation by setting the BD_Res field in BLTDEF. SRAM 1 and SRAM 2 can also be designated as a common result destination with the OP0_opRDRAM.pt.X pointer designating the result offset within the SRAMs. This results in a halving of the number of RDRAM fetches required for filling the SRAMs. This can be useful when common transparency and monochrome-to-color masks are used in the next operation.

When doing explicit SRAM source or destination operations use caution to disable auto-BitBLTs. Since the auto-BitBLT can intervene between programmed BitBLT operations and change the contents of SRAM, unexpected results can occur.