Avenger (a.k.a. Voodoo3)

SUPER HIGH PERFORMANCE GRAPHICS ENGINE FOR 3D GAME ACCELERATION

Revision 1.0
November 30, 1999

3Dfx Interactive, Inc.
4435 Fortran Drive
San Jose, CA 95134
Phone: (408) 935-4400
Fax: (408) 935-4424

Copyright Notice:

[English translations from legalese in brackets]

No Warranty

THESE SPECIFICATIONS ARE PROVIDED BY 3DFX "AS IS" WITHOUT ANY REPRESENTATION OR WARRANTY, EXPRESS OR IMPLIED, INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY RIGHTS, OR ARISING FROM THE COURSE OF DEALING BETWEEN THE PARTIES OR USAGE OF TRADE. IN NO EVENT SHALL 3DFX BE LIABLE FOR ANY DAMAGES WHATSOEVER INCLUDING, WITHOUT LIMITATION, DIRECT OR INDIRECT DAMAGES, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SPECIFICATIONS, EVEN IF 3DFX HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

[You're getting it for free. We believe the information provided to be accurate. Beyond that, you're on your own.]

1. Introduction

Avenger Graphics Engine is a second generation 3D graphics engine based on the original SST1 architecture. Avenger incorporates all of the original SST1 features such as true-perspective texture mapping with advanced mipmapping and lighting, texture anti-aliasing, sub-pixel correction, gouraud shading, depth-buffering, alpha blending and dithering. Avenger also has 2 full-featured texturing units, which allow for advanced features like trilinear filtering, dual-texturing or bump mapping to be performed at the rate of a pixel per clock. In addition to the SST1 features, Avenger includes a VGA core, 2D graphics acceleration, and support for Intel's AGP bus.

Features

- SST1 baseline features with 2 texturing units.
- SST1 software compatible
- AGP2X / PCI bus compliant
- Native VGA core
- 2D acceleration
  Binary/Ternary operand raster ops
  Screen to Screen, Screen to Texture space, and Texture space to Screen Blits.
  Color space conversion YUV to RGB.
  1:N monochrome expansion
  Rendering support of 2048x2048
- Integrated DAC and PLLs.
- Bilinear video scaling
- Video in via feature connector
- Supports SGRAM/SDRAM memories
- TV out interface runs at 100MHz DDR

Video-In:

- operates simultaneously with TVout interface.
- decimation
- support for interlaced video data
- support VMI, SAA7110 video connectors
- triple buffers for video-in data

Video-Out:

- Bilinear scaling zoom-in (from 1 to 10x magnification in increments of 0.25x)
- decimation for zoom-out (0.25x, 0.5x, 0.75x)
- chroma-keying for video underlying and overlaying
- support for stereoscopic display
- hardware cursor
- double buffer frame buffers for video refresh
- DDC support for monitor communication
- DPMS mode support
- overlay windows (for 3D and motion video)

1.1 Resolutions

VGA

MODE #	Mode Type	# of Colors	Native Resolution	Alpha Format
0,1	Alpha	16/256K	320x200	40x25
0,1	Alpha	16/256K	320x350	40x25
0,1	Alpha	16/256K	360x400	40x25
2,3	Alpha	16/256K	640x200	80x25
2,3	Alpha	16/256K	720x400	80x25
2,3	Alpha	16/256K	320x200	80x25
4,5	Graphics	4/256K	640x200	40x25
6	Graphics	2/256K	120x350	80x25
7	Alpha	mono	320x200	80x25
D	Graphics	16/256K	640x350	40x25
E	Graphics	16/256K	640x350	40x25
F	Graphics	mono	640x350	80x25
10	Graphics	16/256K	640x350	80x25
11	Graphics	2/256K	640x480	80x30
12	Graphics	16/256K	640x480	80x30
13	Graphics	256/256K	320x200	40x25

VESA

MODE #	Mode Type	# of Colors	Native Resolution	Alpha Format
100	Graphics	256/256K	640x400	80x25
101	Graphics	256/256K	640x480	80x30

2. Performance

2.1 2D Performance

Estimated triangle performance.

8-bits per pixel, 1024x768 resolution (linear)

10-pixel 2D lines	4M lines/sec
100-pixel 2D lines	400K lines/sec
500-pixel 2D lines	80K lines/sec
10 x 10 filled rectangle	2-4M rectangles/sec
100 x 100 filled rectangle	170K rectangles/sec
500 x 500 filled rectangle	11.5K rectangles/sec
host blit to screen 10 x 10	40M bytes/sec
host blit to screen 100 x 100	50M bytes/sec
screen to screen blit 500 x 500	350M bytes/sec

2.2 3D Performance

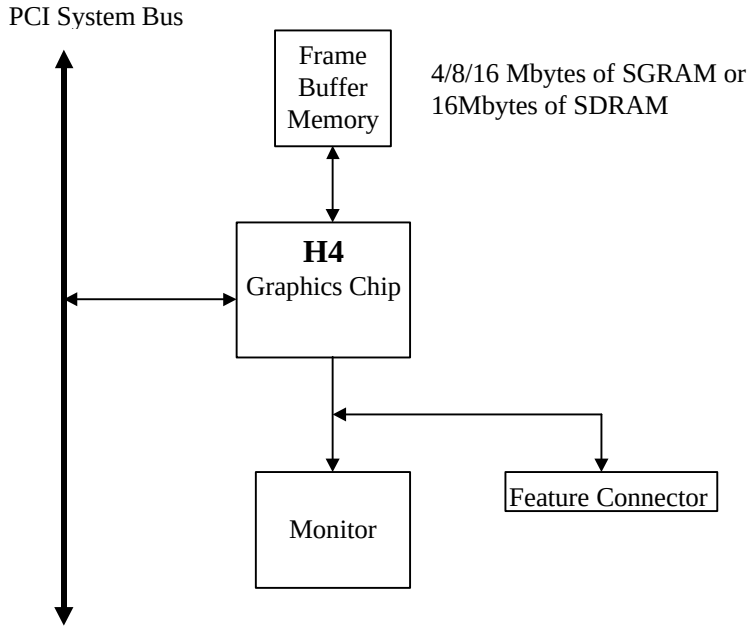
16-bits per pixel, 640x480

1 pixel gouraud, Z, unlit	1.8M tris/sec
5 pixel gouraud, Z, unlit	1.8M tris/sec
50 pixel gouraud, Z, unlit	1.4M tris/sec
1000 pixel gouraud, Z, unlit	70k tris/sec
50 pixel Z, blinear textured	1.0M tris/sec
50 pixel Z, trilinear mip-mapped	375K tris/sec

3. Functional Overview

3.1 System Level Diagrams

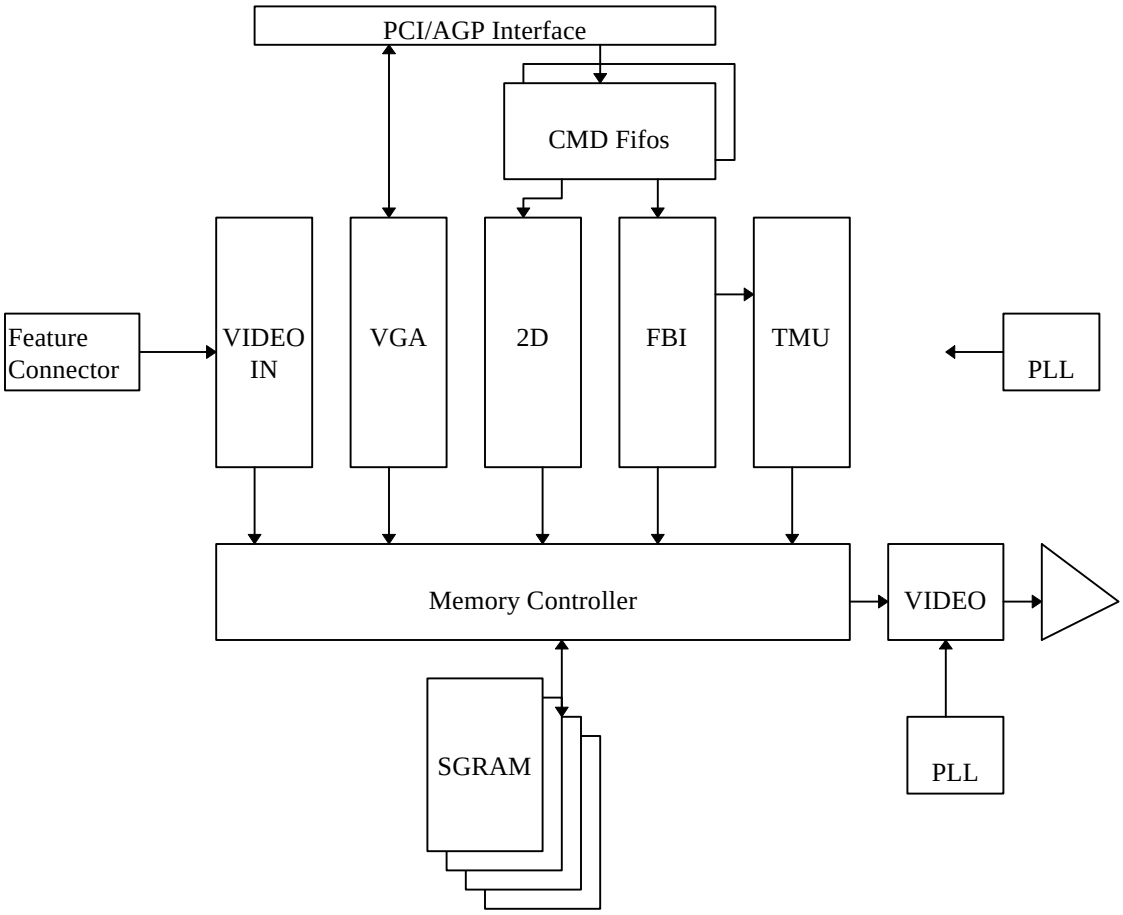
In its entry configuration, an Avenger graphics solution consists of a single ASIC + RAM. Avenger is a PCI Slave device, that receives commands from the CPU via direct writes or through memory backed fifo writes. Avenger includes an entire VGA core, 2D graphic pipeline, 3D graphics engine, texture raster engine, and video display processor. Avenger supports all VGA modes plus a number of VESA modes.



3.2 Architectural Overview

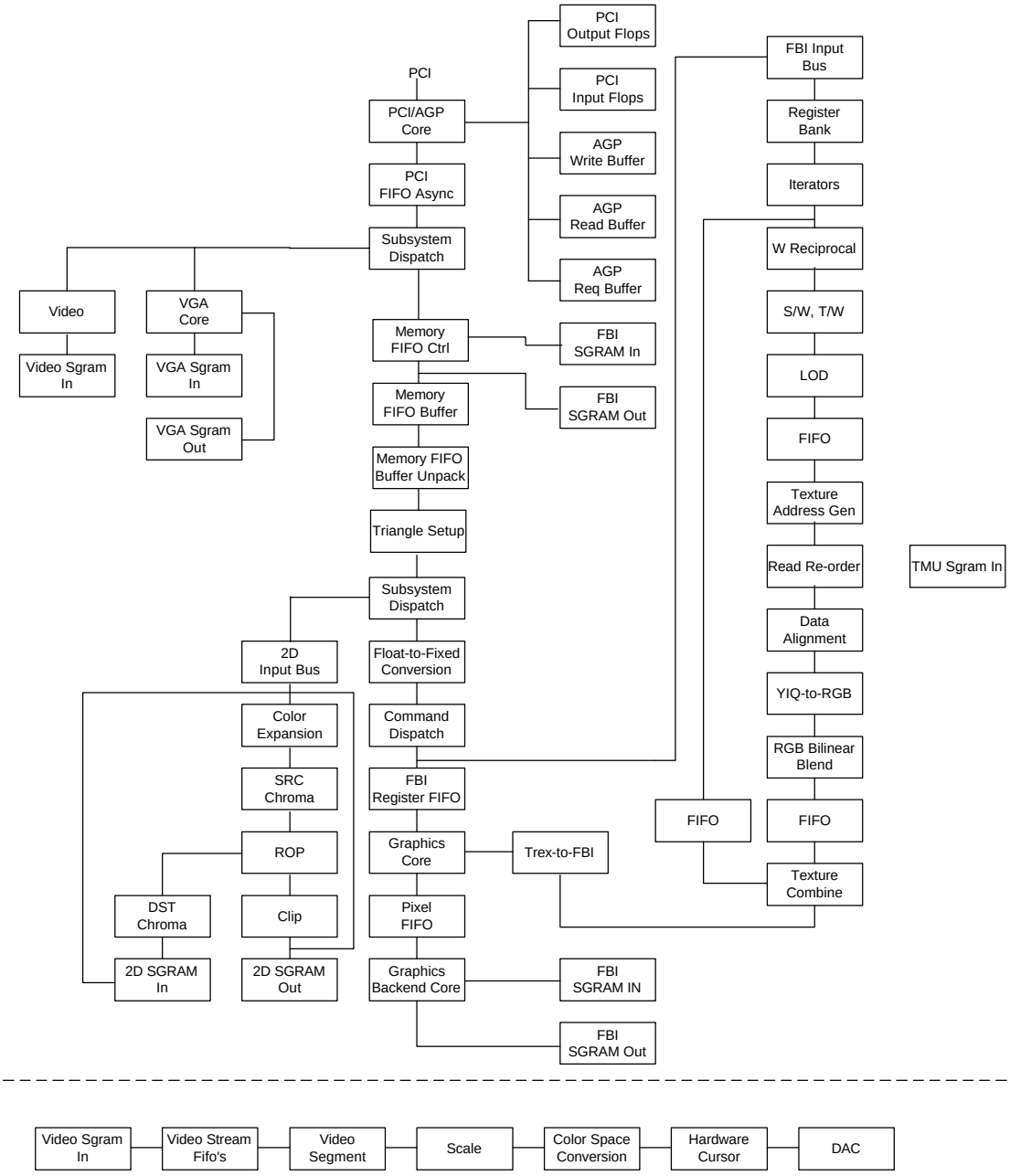
3.2.1 Overall Overview

The diagram below illustrates the overall architecture of the Avenger graphics subsystem.

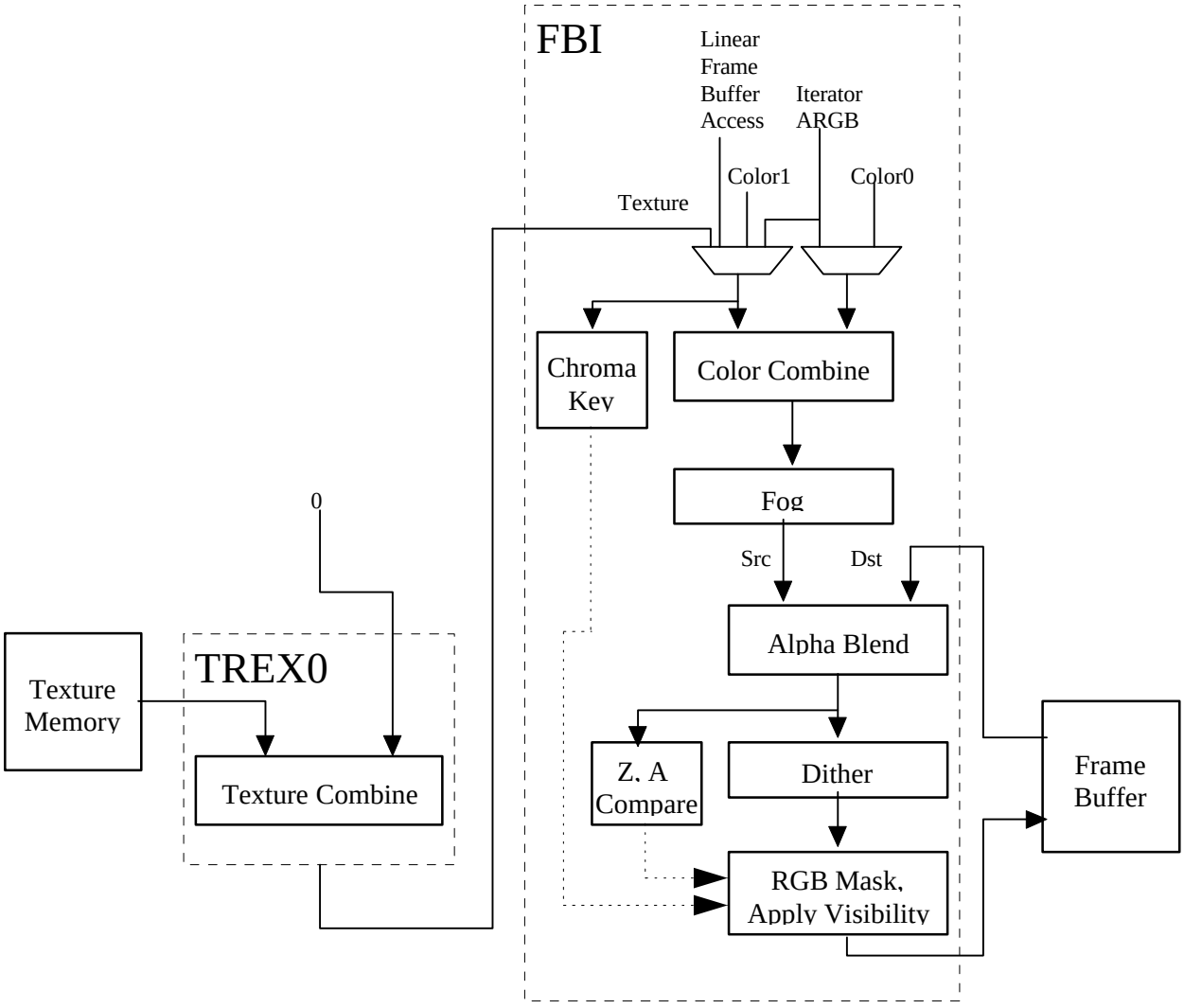


3.2.2 Detailed Datapath Diagram

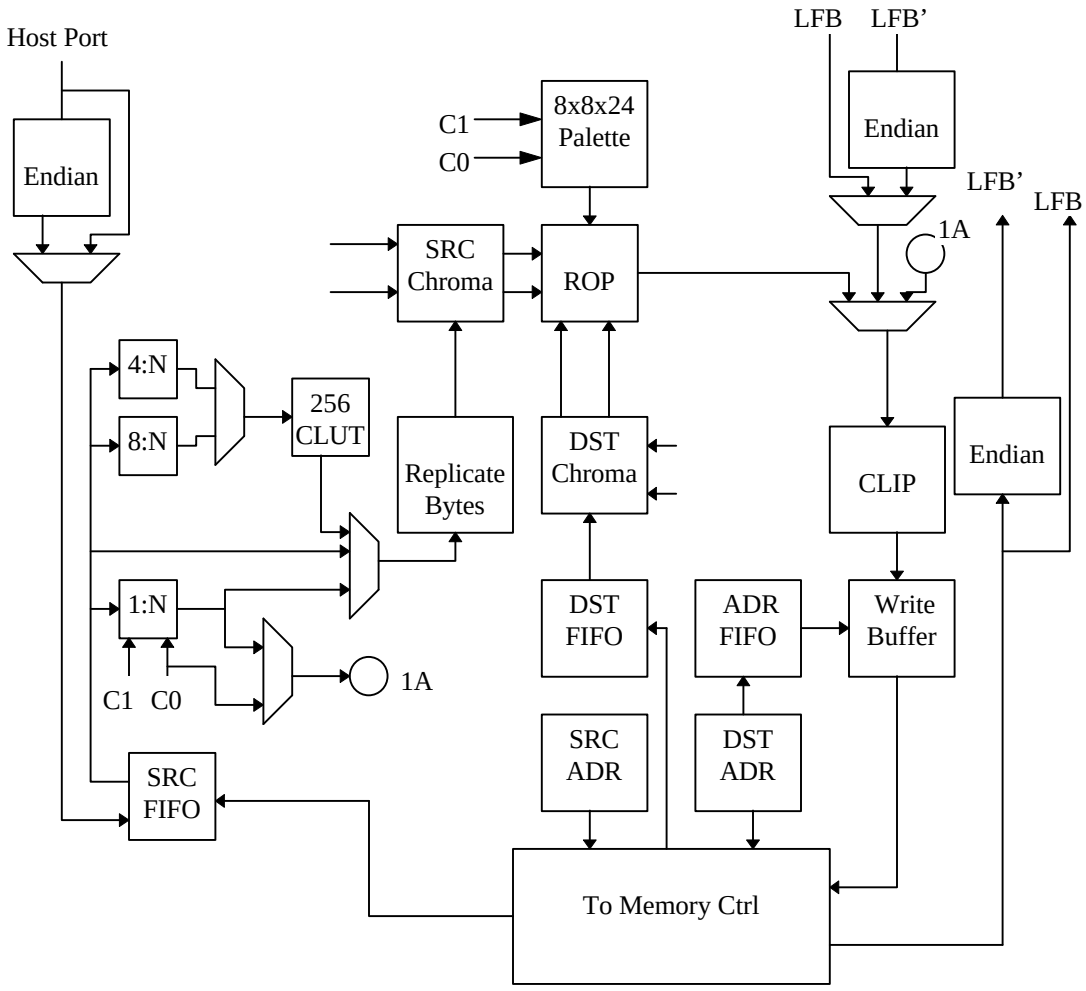
H3 Data Path



3.2.3 FBI/TMU



3.2.4 2D



3.3 Functional Overview

Bus Support: Avenger implements both the PCI bus specification 2.1 and AGP specification 1.0 protocols. Avenger is a slave only device on PCI, and a master device on AGP. Avenger supports zero-wait-state transactions and burst transfers.

PCI Bus Write Posting: Avenger uses an synchronous FIFO 32ntries deep which allows sufficient write posting capabilities for high performance. The FIFO is asynchronous to the graphics engine, thus allowing the memory interface to operate at maximum frequency regardless of the frequency of the PCI bus. Zero-wait-state writes are supported for maximum bus bandwidth.

VGA: Avenger includes a 100% IBM PS/2 model 70 compatible VGA core, which is highly optimized for 128 bit memory transfers. The VGA core supports PC '97 requirements for multiple adapter, and vga disable.

Memory FIFO: Avenger can optionally use off-screen frame buffer memory or AGP memory to increase the effective depth of the PCI bus FIFO. The depth of this memory FIFO is programmable, and when used as an addition to the regular 32 entry host FIFO, allows up to 1Mbyte host writes to be queued

without stalling the PCI interface. Avenger supports 2 independent command streams that are asynchronous to each other. Either command stream can be located in AGP memory or frame buffer memory.

Memory Architecture: The frame buffer controller of Avenger has a 128-bit wide Datapath to RGB, alpha/depth-buffer, 2D desktop, video, and texture memory with support for up to 100 MHz SGRAMs or SDRAMS. For 2D fills using the standard 2D bitBLT engine, 8 16-bit pixels are written per clock, resulting in a 800 Mpixel/sec peak fill rate. For screen clears using the color expansion capabilities specific to SGRAM, 64 bytes are written per clock, resulting in a 6.4 Gbytes/sec peak fill rate. For Gouraud-shaded or textured-mapped polygons with depth buffering enabled, one pixel is written per clock – this results in a 75 Mpixels/sec peak fill rate. The minimum amount of memory supported by Avenger is 4 Mbytes, with a maximum of 16 Mbytes supported.

Storing texture bitmaps, the texture memory controller of Avenger must share the 128-bit wide Datapath to Avenger memory. The texture unit uses sophisticated caching to reduce the required bandwidth of memory to perform bilinear texture filtering with no performance penalty. The amount of texture memory is only limited by the maximum amount of Avenger frame buffer memory.

Host Bus Addressing Schemes: Avenger occupies a combined 64 Mbytes of memory mapped address space, using two PCI memory base address pointers. Avenger also occupies 256 bytes of I/O mapped address space for video and initialization registers. The register space of Avenger occupies 6 Mbytes of address space, the linear frame buffer occupies 32 Mbytes of address space, the ordered texture download port occupies 2 Mbytes of address space, and the 3D pipeline linear frame buffer takes 8 Mbytes of address space.

2D Architecture: Avenger implements a full featured 128-bit 2D windows accelerator capable of displaying 8, 16, 24, and 32 bits-per-pixel screen formats. Avenger supports 1, 8, 16, 24, and 32 bits-per-pixel RGB source pixel maps for BitBlts. 4:2:2 and 4:1:1 YUV colorspace are supported as source bitmaps for host to screen BitBlts. Avenger supports screen-to-screen and host-to-screen stretch BitBlts at 100 Mpixels/Sec. Avenger supports source and destination colorkeying, multiple clip windows, and full support of ternary ROP's. Patterned Bresenham line drawing with full rop support, along with polygon fills are supported in Avenger's 2D core. Fast solid fills, pattern fills, and transparent monochrome bitmap BitBlts in 8 bits-per-pixel, 16 bits-per-pixel, and 32 bits-per-pixel modes.

Linear Frame Buffer and Texture Access: Avenger supports linear frame buffer, texture download access, and 3D pipeline frame buffer access for software ease and regular porting. Multiple color formats are supported for linear frame buffer write. Any pixel may be written to the 3D pixel pipeline for fogging, lighting, alpha blending, dithering, etc. Texture maps can be downloaded into common Avenger memory either through standard linear frame buffer space, 3D pixel pipeline frame buffer access, or down through the ordered texture memory access address space.

Triangle-based Rendering: Avenger supports an triangle drawing primitive and supports full floating point hardware triangle setup. Triangle primitives may be passed from the CPU to Avenger as independent triangles, as part of a triangle strip, or as part of a triangle fan. Only the parameter vertex information is required by the host CPU, as Avenger automatically calculates the parameter slope and gradient information required for proper triangle iteration.

Additional drawing primitives such as spans and lines are rendered as special case triangles. Complex primitives such as quadrilaterals must be decomposed into triangles before they can be rendered by Avenger.

Gouraud-shaded Rendering: Avenger supports Gouraud shading by providing RGBA iterators with rounding and clamping. The host provides starting RGBA and ΔRGBA information, and Avenger automatically iterates RGBA values across the defined span or trapezoid.

Texture-mapped Rendering: Avenger supports full-speed texture mapping for triangles. The host provides starting texture S/W, T/W, 1/W information, and Avengerautomatically calculates their slopes Δ(S/W),

Δ(T/W), and Δ(1/W) required for triangle iteration. Avenger automatically performs proper iteration and perspective correction necessary for true-perspective texture mapping. During each iteration of triangle walking, a division is performed by 1/W to correct for perspective distortion. Texture image dimensions must be powers of 2 and less than or equal to 256. Rectilinear and square texture bitmpas are supported.

Texture-mapped Rendering with Lighting: Texture-mapped rendering can be combined with Gouraud shading to introduce lighting effects during the texture mapping process. The host provides the starting Gouraud shading RGBA as well as the starting texture S/W, T/W, 1/W, and Avenger automatically calculates their slopes ΔRGBA, Δ(S/W), Δ(T/W) required for triangle iteration. Avenger automatically performs the proper iteration and calculations required to implement the lighting models and texture lookups. A texel is either modulated (multiplied by), added, or blended to the Gouraud shaded color. The selection of color modulation or addition is programmable.

Texture Mapping Anti-aliasing: Avenger allows for anti-aliasing of texture-mapped rendering with support for texture filtering and mipmapping. Avenger supports point-sampled, bilinear, and trilinear texture filters. While point-sampled and bilinear are single pass operations, Avenger supports trilinear texture filtering as a two-pass operation.

In addition to supporting texture filtering, Avenger also supports texture mipmapping. Avenger automatically determines the mipmap level based on the mipmap equation, and selects the proper texture image to be accessed. Additionally, the calculated mipmap LOD may be biased and/or clamped to allow software control over the sharpness or “fuzziness” of the rendered image. When performing point-sampled or bilinear filtered texture mapping, dithering of the mipmap levels can also optionally be used to remove mipmap “banding” during rendering. Using dithered mipmapping with bilinear filtering results in images almost indistingusihable from full trilinear filtered images.

Texture Map Formats: Avenger supports a variety of 8-bit and 16-bit texture formats as listed below:

8-bit Texture Formats	16-bit Texture Formats
RGB (3-3-2)	RGB (5-6-5)
Alpha(8)	ARGB(8-3-3-2)
Intensity(8)	ARGB(1-5-5-5)
Alpha-Intensity(4-4)	ARGB(4-4-4-4)
YAB(4-2-2)	Alpha-Intensity(8-8)
PalettedRGB(8 expanded to RGB 8-8-8)	Alpha-Paletted RGB(8-8 expanded to RGB 8-8-8)
PalettedRGBA(8 expanded to ARGB 6-6-6-6)	AYAB (8-4-2-2)

Avenger includes an internal 256-entry texture palette, which can be downloaded directly from the host CPU or via a command to load the palette directly from texture memory. Either during downloads or rendering, software programs a palette offset register to control which portion of the texture palette is to be used.

Texture-space Decompression: Texture data compression is accomplished using a “narrow channel” YAB compression scheme. 8-bit YAB format is supported. The compression is based on an algorithm which compresses 24-bit RGB to a 8-bit YAB format with little loss in precision. The compression scheme is called “YAB” because it effectively creates a unique color space for each individual texture map - examples of potential color spaces utilized include YIQ, YUV, etc. This YAB compression algorithm is especially suited to texture mapping, as textures typically contain very similar color components. The algorithm is performed by the host CPU, and YAB compressed textures are passed to Avenger. The advantages of using compressed textires are increased effective texture storage space and lower bandwidth requirements to perform texture filtering.

Polygonal Anti-Aliasing: To eliminate the "jaggies" on the edges of triangles, Avenger supports polygonal anti-aliasing. To use the anti-aliasing support in Avenger, triangles must be sorted before rendering, either back-to-front or front-to-back. When front-to-back triangle ordering is used, the standard OpenGL alpha-saturate algorithm is used to anti-alias the polygon edges. When back-to-front triangle ordering is used, standard alpha-blending is used to partially blend the edges of the triangles into the previously rendered scene. Regardless of which triangle ordering technique is used, the hardware automatically determines the pixels on the edges of the rendered triangles which are special-cased and rendered with less than full-intensity to smooth the triangle edges.

Depth-Buffered Rendering: Avenger supports hardware-accelerated depth-buffered rendering with minimal performance penalty when enabled. The standard 8 depth comparison operations are supported. To eliminate many of the Z-aliasing problems typically found on 16-bit Zbuffer graphics solutions, Avenger allows the (1/W) parameter to be used as the depth component for hardware-accelerated depth-buffered rendering. When the (1/W) parameter is used for depth-buffering, a 16-bit floating point format is supported. A 16-bit floating point(1/W)-buffer provides much greater precision and dynamic range than a standard 16-bit Z-buffer, and reduces many of the Z-aliasing problems found on 16-bit Z-buffer systems.

To handle co-planar polygons, Avenger also supports depth biasing. To guarantee that polygons which are co-planar are rendered correctly, individual triangles may be biased with a constant depth value - this effectively accomplishes the same function as stenciling used in more expensive graphics solutions but without the additional memory costs.

Pixel Blending Operation: Avenger supports alpha blending functions which allow incoming source pixels to be blended with current destination pixels. An alpha channel (ie. Destination alpha) stored in offscreen memory is only supported when depth-buffering is disabled. The alpha blending function is as follows:

$Dnew \Leftarrow (S \bullet \alpha) + (Dold \bullet \beta)$

where

Dnew	The new destination pixel being written into the frame buffer
S	The new source pixel being generated
Dold	The old (current) destination pixel about to be modified
α	The source pixel alpha function.
β	The destination pixel alpha function.

Fog: Avenger supports a 64-entry lookup table to support atmospheric effects such as fog and haze. When enabled, a 6-bit floating point representation of (1/W) is used to index into the 64-entry lookup table. The output of the lookup table is an "alpha" value which represents the level of blending to be performed between the static fog/haze color and the incoming pixel color. Low order bits of the floating point (1/W) are used to blend between multiple entries of the lookup table to reduce fog "banding." The fog lookup table is loaded by the host CPU, so various fog equations, colors, and effects are supported.

Color Modes: Avenger supports 16-bit RGB (5-6-5) buffer displays only. Internally, Avenger graphics utilizes a 32-bit ARGB 3D pixel pipeline for maximum precision, but the 24-bit internal RGB color is dithered to 16-bit RGB before being stored in the color buffers. The host may also transfer 24-bit RGB pixels to Avenger using 3D linear frame buffer accesses, and color dithering is utilized to convert the input pixels to native 16-bit format with no performance penalty.

Chroma-Key and Chroma-Range Operation: Avenger supports a chroma-key operation used for transparent object effects. When enabled, an outgoing pixel is compared with the chroma-key register. If a match is detected, the outgoing pixel is invalidated in the pixel pipeline, and the frame buffer is not updated. In addition, a superset of chroma-keying, known as chroma-ranging, may be used. Instead of

matching outgoing pixels against a single chroma-key color, chroma-ranging uses a range of colors for the comparison. If the outgoing pixel is within the range specified by the chroma-range registers and chroma-ranging is enabled, then the frame buffer is updated with the pixel.

Color Dithering Operations: All operations internal to Avenger operate in native 32-bit ARGB pixel mode. However, color dithering from the 24-bit RGB pixels to 16-bit RGB (5-6-5) pixels is provided on the back end of the pixel pipeline. Using the color dithering option, the host can pass 24-bit RGB pixels to Avenger, which converts the incoming 24-bit RGB pixels to 16-bit RGB (5-6-5) pixels which are then stored in the 16-bit RGB buffer. The 16-bit color dithering allows for the generation of photorealistic images without the additional cost of a true color frame buffer storage area.

Programmable Video Timing: Avenger uses a programmable video timing controller which allows for very flexible video timing. Any monitor type may be used with Avenger , with 76+ Hz vertical refresh rates supported at 800x600 resolution, and 100+ Hz vertical refresh rates supported at 640x480 resolution. Lower resolutions down to 320x200 are also supported.

Video Output Gamma Correction: Avenger uses a programmable color lookup table to allow for programmable gamma correction. The 16-bit dithered color data from the frame buffer is used an an index into the gamma-correction color table -- the 24-bit output of the gamma-correction color table is then fed to the monitor

Video Overlay: Avenger supports one full featured video overlay that is unlimited in size, and supports pixel formats of YUV 411, YUV 422, RGB (1-5-5-5), and RGB (5-6-5). The video overlay can be double, tripple or quad buffered, and can be bilinear scaled to full screen resolutions.

Video In: VMI video in port with complete host port is fully supported in Avenger. Video in is double buffered and can be optionally deinterlaced by replicating lines in a single frame or by merging 2 frames together.

PLL/DAC: Avenger contains 3 independent PLL's for clock generation. The PLL's are totally programmable giving the capability to change video, graphics, and memory clocks to any specified frequency. Avenger supports a high speed 300 MHz ramdac, capable of doing 1600x1280 @ 76Hz refresh.

3.4 Modifications from SST1

- Colbufsetup
- Auxbufsetup
- Chroma Range
- intrCtrl, userIntrCMD
- fbiTriangles register
- Full triangle setup registers
- Fogmode
- Fogtable
- fbzColorPath
- fbzMode
- increase of rendering window to -4k to 4k
- Additional clip rectangle
- Byte access lfb
- New command fifo interface
- Texture mirroring
- Addition of VGA core

- Addition of Video surfaces
- Additional 6666 palettized texture format
- Full featured 2D accelerator engine.
- Separate filter controls for Alpha, and RGB.
- Combined TMU unit
- Increased blending fraction from 1.4 to 1.8.
- Separate register / LFB byte swizzling for big endian machines.
- PC ’97 compliant

3.5 Additions to Avenger from Banshee

- Higher core clock frequency = 143MHz.
- Graphics core and memory interface now all run on a master graphics clock.
- 350MHz RAMDAC.
- 0.25 micron, 5lm technology
- 452pin BGA
- AGP 2x support
- Deeper on-chip command fifo RAM to increase AGP command fifo performance.
- Programmable watermarks for lfb/cmdfifo write fifo (pciInit0); can increase efficiency of command transport.
- 2 TMUs for to enable single-cycle special effects such as trilinear filtering, dual-texturing and bump-mapping.
- *tsplit* functionality added back in to the TMUs.
- Video fetch performance modification (controlled with CYA in vidProcCfg); boost video performance by making video fifo thresholds more effective.
- Increased performance for minified textures (texture fetch engine was modified).
- Adjustable delay for TV-out clock.
- Support for simultaneous VMI and TV-out.
- Additional internal status observability registers:cmdStatus0, cmdStatus1.
- Removal of separate mclk domain (mclk domain is now gclk domain).
- Two device ID’s supported: 5=high-speed Avenger, 4=lower speed sort; different PLL programming is required depending on device ID: see section on PLL programming.

3.6 Programming Notes on Avenger vs. Banshee

- Video register changes per TV-out interface: addtion of (**VidInStatusCurrentLine**, **vidTvOutBlankHCount**, **vidTvOutBlankVCount, vidInFormat, vidSerialParallelPortRegister, vidInYDecimDeltas**)
- Additional flushing code required around texture downloads (*Maintaining Cache Coherency*, **section 18.3**)
- Additional texture download aperture: see Avenger Address Space and Command Packet 5 sections.
- Software should try to tune video fifo watermarks to boost performance, given the enhanced video fetch logic.
- Programming of PLL depends on device ID: id==5 -> m, n, k are all fully programmable; id==4 -> m is fixed to 0x18 (24d); see **section 9**.
- Problem with VGA-space P6-style write combining is fixed.
- Board Note: Because of the presence of an AGP pll, it is strongly recommended that the chip not be run in AGP pll bypass mode.

- SDRAM **fastfillCMD** command must **still** be done by using just the color-plane fill.
- Swapbuffer pending count logic is fixed, and will increment/decrement as described in the documentation.

4. Avenger Address Space

MemoryBase0

Memory Address	
0x0000000 – 0x007FFFF	I/O register remap (See I/O section below)
0x0080000 – 0x00FFFFF	CMD/AGP transfer/Misc. registers
0x0100000 – 0x01FFFFF	2D registers
0x0200000 – 0x05FFFFF	3D registers
0x0600000 – 0x07FFFFF	TMU0 Texture Download
0x0800000 – 0x09FFFFF	TMU1 Texture Download
0x0a00000 – 0x0BFFFFF	Reserved.
0x0C00000 – 0x0FFFFFF	YUV planar space
0x1000000 – 0x1FFFFFF	3D LFB space

Memory Base1

Memory Address	
0x0000000 - 0x1FFFFFF	LFB space

I/O Base0

I/O Address	
0x00 - 0x03	status Register
	Initialization registers
0x04 - 0x07	pciInit0 register
0x08 - 0x0b	sipMonitor register
0x0c - 0x0f	lfbMemoryConfig register
0x10 - 0x13	miscInit0 register
0x14 - 0x17	miscInit1 register
0x18 - 0x1b	dramInit0 register
0x1c - 0x1f	dramInit1 register
0x20 - 0x23	agpInit register
0x24 - 0x27	tmuGbeInit register
0x28 - 0x2b	vgaInit0 register
0x2c - 0x2f	vgaInit1 register
0x30 - 0x33	dramCommand register (see 2D offset 0x70)
0x34 - 0x37	dramData register (see 2D offset 0x064)
0x38 - 0x3b	strapInfo
	A Video Register:
0x3c – 0x3f	VidTvOutBlankVCount
	PLL and Dac registers
0x40 - 0x43	pllCtrl0
0x44 - 0x47	pllCtrl1
0x48 - 0x4b	pllCtrl2
0x4c - 0x4f	dacMode register.
0x50 - 0x53	dacAddr register.
0x54 - 0x57	dacData register.
	Video Registers part I
0x58 - 0x5b	rgbMaxDelta register

0x5c - 0x5f	vidProcCfg register.
0x60 - 0x63	hwCurPatAddr register.
0x64 - 0x67	hwCurLoc register.
0x68 - 0x6b	hwCurC0 register
0x6c - 0x6f	hwCurC1 register.
0x70 - 0x73	vidInFormat register
0x74 - 0x77	vidTvOutBlankHCount register
0x78 - 0x7b	vidSerialParallelPort register
0x7c - 0x7f	vidInXDecimDeltas register.
0x80 - 0x83	vidInDecimInitErrs register.
0x84 - 0x87	vidInYDecimDeltas register.
0x88 - 0x8b	vidPixelBufThold register
0x8c - 0x8f	vidChromaMin register.
0x90 - 0x93	vidChromaMax register.
0x94 - 0x97	vidInStatusCurrentLine register.
0x98 - 0x9b	vidScreenSize register.
0x9c - 0x9f	vidOverlayStartCoords register.
0xa0 - 0xa3	vidOverlayEndScreenCoord register.
0xa4 - 0xa7	vidOverlayDudx register
0xa8 - 0xab	vidOverlayDudxOffsetSrcWidth register.
0xac - 0xaf	vidOverlayDvdy register.
	VGA Registers
0xb0 - 0xdf	vga registers (only in I/O space, not memory mapped)
	Video Registers part II
0xe0 - 0xe3	vidOverlayDvdyOffset register.
0xe4 - 0xe7	vidDesktopStartAddr register.
0xe8 - 0xeb	vidDesktopOverlayStride register.
0xec - 0xef	vidInAddr0 register
0xf0 - 0xf3	vidInAddr1 register.
0xf4 - 0xf7	vidInAddr2 register.
0xf8 - 0xfb	vidInStride register.
0xfc - 0xff	vidCurrOverlayStartAddr register.

VGA Address Space

Memory Address
0x00A0000 - 0x00BFFFF
I/O Addresses (8 bit / 16 bit) addressable
0x0102
0x03B4 - 0x03B5
0x03BA
0x3C0 - 0x03CA
0x03CE - 0x03CF
0x03DA
0x46E8

5. VGA Register Set

5.1 Overview of the Avenger VGA Controller

The Avenger VGA core supports all standard VGA modes with full backward compatibility. This allows the 3D controller to be able to share the frame buffer with the 2D controller, thereby saving total solution cost.

In addition to the legacy VGA, Avenger also supports Vesa BIOS extensions. This is accomplished by extending the standard register set and implementing a flexible memory aperture such that VBE applications can page select memory through the standard VGA address space.

5.2 Using VGA Registers When Avenger is not the Primary VGA

For systems not requiring VGA or a VGA device already exists, Avenger allows the use of the VGA registers in an extended fashion. In this mode, VGA registers are not decoded in legacy VGA space, but in relocatable IO and memory space.

Avenger should be powered on with the device type set to ‘Multimedia
registers. Avenger will not respond to any legacy I/O or memory space. In order to use the VGA registers, Avenger should be set up to be a motherboard device (**VGAINIT0** bit 8), and the IO base + 0xc3 bit 0 should be set to 1.

In this configuration, all of the VGA registers (except 0x46e8 and 0x0102) are available by truncating the leading ‘0x03’ from the legacy address, and adding that address to the I/O base address.

Note that in this configuration, however, memory is not accessible through the VGA aperture.

5.3 Locking VGA Timing for Virtualized Modes

When running VGA applications in a window, it is possible to restrict changes to the VGA timing registers set. This is accomplished by setting the lock bits in **vgaInit1**. The locks prevent applications from changing the values in the associated registers.

5.4 Setting VGA Timing for Video 2 Pixels per Clock Mode

For extended resolutions that run at frequencies greater than 135Mhz, it is required that the Video Unit be placed in a 2 pixel per clock mode. This implies that the video clock is divided by 2 (see **dacMode**). Since the clock is running at half the frequency, all horizontal timing registers should also be divided in half.

Note: All horizontal video timing must be divisible by 16 pixels.

VGA Registers

This section outlines the compatible VGA register set followed by a brief description of their operation.

PORT	INDEX	Register Name
0x3B4/0x3D4	-	CRTC Index Register
0x3B5/0x3D5	0x0	Horizontal Total
0x3B5/0x3D5	0x1	Horizontal Display Enable End
0x3B5/0x3D5	0x2	Start Horizontal Blanking
0x3B5/0x3D5	0x3	End Horizontal Blanking
0x3B5/0x3D5	0x4	Start Horizontal Retrace
0x3B5/0x3D5	0x5	End Horizontal Retrace
0x3B5/0x3D5	0x6	Vertical Total
0x3B5/0x3D5	0x7	Overflow
0x3B5/0x3D5	0x8	Preset Row Scan
0x3B5/0x3D5	0x9	Maximum Scan Line
0x3B5/0x3D5	0xA	Cursor Start
0x3B5/0x3D5	0xB	Cursor End
0x3B5/0x3D5	0xC	Start Address High
0x3B5/0x3D5	0xD	Start Address Low
0x3B5/0x3D5	0xE	Cursor Location High
0x3B5/0x3D5	0xF	Cursor Location Low
0x3B5/0x3D5	0x10	Vertical Retrace Start
0x3B5/0x3D5	0x11	Vertical Retrace End
0x3B5/0x3D5	0x12	Vertical Display Enable End
0x3B5/0x3D5	0x13	Offset
0x3B5/0x3D5	0x14	Underline Location
0x3B5/0x3D5	0x15	Start Vertical Blank
0x3B5/0x3D5	0x16	End Vertical Blank
0x3B5/0x3D5	0x17	CRTC Mode Control
0x3B5/0x3D5	0x18	Line Compare
0x3B5/0x3D5	0x1A	Horizontal Extension Register
0x3B5/0x3D5	0x1B	Vertical Extension Register
0x3B5/0x3D5	0x1C	Extension Byte 0/ PCI Configuration
0x3B5/0x3D5	0x1D	Extension Byte 1
0x3B5/0x3D5	0x1E	Extension Byte 2
0x3B5/0x3D5	0x1F	Extension Byte 3
0x3B5/0x3D5	0x22	Latch Read Back
0x3B5/0x3D5	0x24	Attribute Controller Index/Data State

CRTC Register Set

Read Port	Write Port	Register Name
0x3CC	0x3C2	Miscellaneous Output

0x3C2	--	Input Status Register 0
0x3BA/0x3DA	--	Input Status Register 1
0x3CA	0x3BA/0x3DA	Feature Control
0x3C3	0x3C3	Motherboard Enable
--	0x46E8	Adapter Enable
0x102	0x102	Subsystem Enable

General Register Set

PORT	INDEX	Register Name
0x3C4	-	Sequencer Index Register
0x3C5	0x0	Reset
0x3C5	0x1	Clocking Mode
0x3C5	0x2	Map Mask
0x3C5	0x3	Character Map Select
0x3C5	0x4	Memory Mode

Sequencer Register Set

PORT	INDEX	Register Name
0x3CE	-	Graphics Controller Index Register
0x3CF	0x0	Set/Reset
0x3CF	0x1	Enable Set/Reset
0x3CF	0x2	Color Compare
0x3CF	0x3	Data Rotate
0x3CF	0x4	Read Map Select
0x3CF	0x5	Graphics Mode
0x3CF	0x6	Miscellaneous
0x3CF	0x7	Color Don't Care
0x3CF	0x8	Bit Mask

Graphics Controller Register Set

PORT	INDEX	Register Name
0x3C0	0x0-0xF	Palette Registers
0x3C0	0x10	Attribute Mode Control Register
0x3C0	0x11	Over Scan Control Register
0x3C0	0x12	Color Plane Enable Register
0x3C0	0x13	Horizontal PEL Panning Register
0x3C0	0x14	Color Select Register

Attribute Controller Register Set

PORT	Register Name
0x3C6	Pixel Mask
0x3C7	Read Index
0x3C7	Read Status
0x3C8	Write Index

0x3C9	Data

RAMDAC Register Set

5.5 General Registers:

5.5.1 Input Status 0 (0x3C2)

Bit	R/W	Description
7	R	Interrupt Status. When its value is “1”, denotes that an interrupt is pending.
6:5	R	Feature Connector. These 2 bits are readable bits from the feature connector.
4	R	Sense. This bit reflects the state of the DAC monitor sense logic.
3:0	R	Reserved. Read back as 0.

Data written to port 0x3C2 is stored in the **Miscellaneous Output Register** (0x3CC).

5.5.2 Input Status 1 (0x3BA/0x3DA)

Bit	R/W	Description
7:6	R	Reserved. These bits read back 0.
5:4	R	Display Status. These 2 bits reflect 2 of the 8 pixel data outputs from the Attribute controller, as determined by the Attribute controller index 0x12 bits 4 and 5.
3	R	Vertical sync Status. A “1” indicates vertical retrace is in progress.
2:1	R	Reserved. These bits read back 0x2.
0	R	Display Disable. When this bit is 1, either horizontal or vertical display end has occurred, otherwise video data is being displayed.

5.5.3 Feature Control Write (0x3BA/0x3DA)

Bit	R/W	Description
7:4	-	Reserved
3	W	Vertical Sync Select
2	-	Reserved
1:0	W	Feature Control

5.5.4 Feature Control Read (0x3CA)

Bit	R/W	Description
7:6	R	Reserved
5:4	R	Video Status. Reads back two bits of the VGA video stream. See 0x3c0, index 0x12, bits 5:4.
3	R	Vertical Sync Select
2	R	Reserved
1:0	R	Feature Control

5.5.5 Miscellaneous Output (0x3CC)

BIT	R/W	Description
7	R	Vertical Sync Polarity (0 = positive, 1= negative).
6	R	Horizontal Sync Polarity (0 = positive, 1= negative).
5	R	Page Select. When in Odd/Even mode Select High 64k bank if set.
4	R	Reserved
3:2	R	Clock Select
1	R	Ram Enable (1= Enable)
0	R	CRTC I/O Address. (1 = Color. Base Address=0x3D?; 0 = Mono. Base Address=0x3B?).

Data is written to this register via port 0x3C2. Bits 6-7 also indicate the number of lines on the display,

[7:6]	Displayed Lines
0	Reserved
1	400
2	350
3	480

[3:2]	Frequency
0	25.175 Mhz
1	28.322 Mhz
2	50 Mhz
3	Programmable PLL.

while bit 3-2 select the video clock frequency.

5.5.6 Motherboard Enable (0x3C3)

Bit	R/W	Description
7:1	R	Reserved
0	R/W	Video Subsystem enable

5.5.7 Adapter Enable (0x46E8)

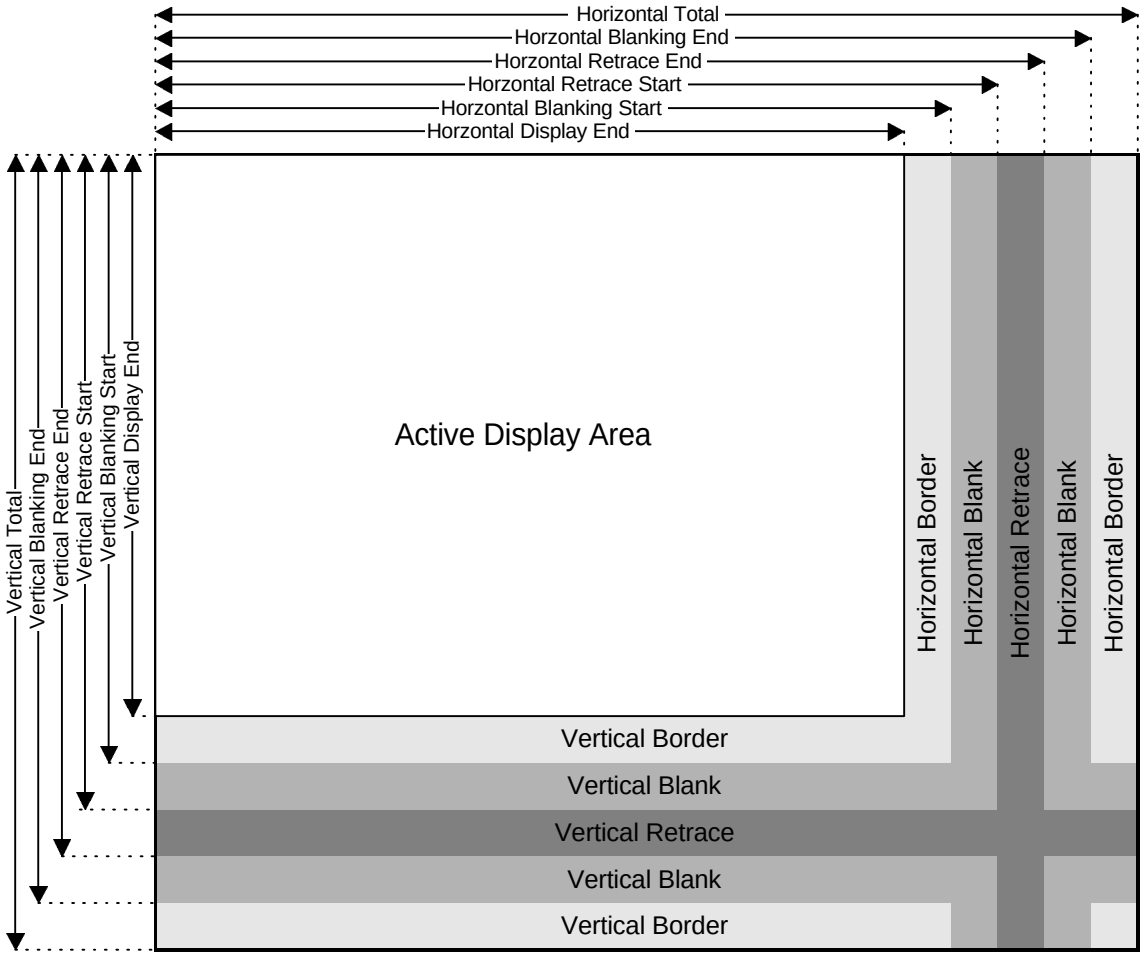
Bit	R/W	Description
7:5	R	Reserved
4	W	Setup Mode
3	W	Video Subsystem Enable
2:0	R	ROM Bank Address - Unused.

5.5.8 Subsystem Enable (0x102)

Bit	R/W	Description
7:1	R	Reserved
0	W	Global Subsystem enable

5.6 CRTC Registers:

The CRTC registers are responsible for the video timing on Avenger. By default, Avenger is a 100% compatible VGA. However, Avenger can also be set up to drive much larger resolutions than that allowed by the VGA standard.



The following chart indicates the bit locations for the timing registers.

	10	9	8	7	6	5	4	3	2	1	0
Horizontal											
Total			1A[0]	0 [7]	0[6]	0[5]	0[4]	0[3]	0[2]	0[1]	0[0]
Active End			1A[2]	1[7]	1[6]	1[5]	1[4]	1[3]	1[2]	1[1]	1[0]
Start Blank			1A[4]	2[7]	2[6]	2[5]	2[4]	2[3]	2[2]	2[1]	2[0]
End Blank					1A[5]	5[7]	3[4]	3[3]	3[2]	3[1]	3[0]
Start Sync			1A[6]	4[7]	4[6]	4[5]	4[4]	4[3]	4[2]	4[1]	4[0]
End Sync						1A[7]	5[4]	5[3]	5[2]	5[1]	5[0]
Vertical											
Total	1B[0]	7[5]	7[0]	6[7]	6[6]	6[5]	6[4]	6[3]	6[2]	6[1]	6[0]
Active End	1B[3]	7[6]	7[1]	12[7]	12[6]	12[5]	12[4]	12[3]	12[2]	12[1]	12[0]
Blank Start	1B[4]	9[5]	7[3]	15[7]	15[6]	15[5]	15[4]	15[3]	15[2]	15[1]	15[0]
Blank End				16[7]	16[6]	16[5]	16[4]	16[3]	16[2]	16[1]	16[0]
Sync Start		7[7]	7[2]	10[7]	10[6]	10[5]	10[4]	10[3]	10[2]	10[1]	10[0]
Sync End								11[3]	11[2]	11[1]	11[0]

5.6.1 CRTC Index Register (0x3B4/0x3D4)

This register provides index information for any subsequent accesses to 0x3B5/0x3D5.

Bit	R/W	Description
7:6	R	Reserved
5:0	R/W	CRTC Index Register.

5.6.2 Index 0x0-Horizontal Total (0x3B5/0x3D5)

This register defines the total width of the display in character clocks, including retrace time, minus 5. Bit 8 of this register is found in the **Horizontal Extension Register** (index 0x1A) bit 0.

Bit	R/W	Description
7:0	R/W	Total Horizontal Character Count less 5.

The 5 character clocks are reserved to provide adequate prefetch time for the beginning data on the first line.

5.6.3 Index 0x1-Horizontal Display Enable End (0x3B5/0x3D5)

This register defines the total number of visible horizontal characters on the display, minus one. Bit 8 of this register is found in the **Horizontal Extension Register** (index 0x1A) bit 2.

Bit	R/W	Description
7:0	R/W	Display Active Characters -1.

5.6.4 Index 0x2-Start Horizontal Blanking (0x3B5/0x3D5)

Horizontal blanking begins when the horizontal character counter reaches this character clock value. Bit 8 of this register is found in the **Horizontal Extension Register** (index 0x1A) bit 4.

Bit	R/W	Description
7:0	R/W	Start Horizontal Blanking

5.6.5 Index 0x3-End Horizontal Blanking (0x3B5/0x3D5)

Bit	R/W	Description
7	R/W	Compatibility Read. When this bit is set to '1' Vertical Sync Start and Vertical Sync End are both readable and writeable. When set to '0' these registers are still writeable, but not readable.
6:5	R/W	Display Enable Signal Skew Control. These bits define the display enable signal skew time in relation to horizontal synchronization pulses.
4:0	R/W	End Horizontal Blanking. End Horizontal Blank signal width is determined as the value of start blanking register plus *W* in character clocks. The least significant five bits are programmed in this register, while the most significant bit is the **End Horizontal Retrace Register** (Index 0x05) bit 7.

5.6.6 Index 0x4-Start Horizontal Sync (0x3B5/0x3D5)

This register contains the character count at which horizontal sync output pulse becomes active. Bit 8 of this register is found in the **Horizontal Extension Register** (index 0x1A) bit 6.

Bit	R/W	Description
7:0	R/W	Start Horizontal Sync Character Count.

5.6.7 Index 0x5-End Horizontal Sync (0x3B5/0x3D5)

Bit	R/W	Description
7	R/W	Horizontal Blank Overflow Bit 5. MSB (bit 5) of End Horizontal Blanking Register
6:5	R/W	Horizontal Sync Skew. These bits define the number of character clocks the horizontal Sync signal is skewed.
4:0	R/W	End Horizontal Sync Pulse Width "W". Start retrace register value is added to the character count for width *W*. The least significant five bits are programmed in this register. When the Start Horizontal Retrace Register value matches these five bits, the horizontal retrace signal is turned off.

5.6.8 Index 0x6-Vertical Total (0x3B5/0x3D5)

The least significant eight bits of a ten bit count of raster scan lines for a display frame less 2. Time for vertical retrace, and vertical sync are also included. The ninth and tenth bits of this count are loaded into the Vertical Overflow Register (Index 0x7) bit 0 and bit 5 respectively. Bit 8 of this register is found in the **Horizontal Extension Register** (index 0x1B) bit 0.

Bit	R/W	Description
7:0	R/W	Raster Scan Line Total Less 2.

5.6.9 Index 0x7-Overflow (0x3B5/0x3D5)

This register contains 'Overflow' bits from other CRTC registers.

Bit	R/W	Description	Base Index
7	R/W	Vertical Sync Start Bit 9.	0x10
6	R/W	Vertical Display Enable End Bit 9.	0x12
5	R/W	Vertical Total Bit 9.	0x6
4	R/W	Line Compare Bit 8.	0x18
3	R/W	Start Vertical Blank Bit 8.	0x15
2	R/W	Vertical Retrace Start Bit 8.	0x10
1	R/W	Vertical Display Enable End Bit 8.	0x12
0	R/W	Vertical Total Bit 8.	0x6

5.6.10 Index 0x8-Preset Row Scan (0x3B5/0x3D5)

BIT	R/W	Description
7	R	Reserved.
6:5	R/W	Byte Panning Control. These bits allow up to 3 bytes to be panned in modes programmed as multiple shift modes.
4:0	R/W	Preset Row Scan Count. These bits preset the vertical row scan counter once after each vertical retrace. This counter is incremented after each horizontal retrace period, until the maximum row scan count is reached. When maximum row scan count is reached, the counter is cleared. This register can be used for smooth vertical scrolling of text.

	5	Left Shift
0	0	0 Pixels
0	1	8 Pixels
1	0	16 Pixels
1	1	24 Pixels

5.6.11 Index 0x9-Maximum Scan Line (0x3B5/0x3D5)

Bit	R/W	Description
7	R/W	Line Doubling. 0= Normal Operation. 1 = Activate line doubling.
6	R/W	Line Compare. Bit 9 of the **Line Compare Register** (index = 0x18).
5	R/W	Start Vertical Blank. Bit 9 of the **Start Vertical Blank Register** (index = 0x15).
4:0	R/W	Maximum Scan Line. Maximum number of scanned lines for each row of characters. The value programmed is the maximum number of scanned rows per character minus 1.

5.6.12 Index 0xA-Cursor Start (0x3B5/0x3D5)

Bit	R/W	Description
7:6	R	Reserved. Defaults to 0.
5	R/W	Cursor Control. 0=Cursor on, 1= Cursor off.
4:0	R/W	Cursor Start Scan Line These bits specify the row scan counter value within the character box where the cursor begins. These bits contain the value of the character row less 1. If this value is programmed with a value greater than the **Cursor End Register** (index = 0xB), no cursor is generated.

5.6.13 Index 0xB-Cursor End (0x3B5/0x3D5)

Bit	R/W	Description
7	R	Reserved. Defaults to 0.
6:5	R/W	Cursor Skew Bits. Delays the displayed cursor to the right by the skew value in character clocks e.g., 1 character clock skew moves the cursor right by 1 position on the screen.
4:0	R/W	Cursor End Scan Line. These bits specify the last row scan counter value within the character box during which the cursor is active. If this value is less than the cursor start value, no cursor is displayed.

5.6.14 Index 0xC-Start Address High (0x3B5/0x3D5)

Eight high order bits of the 16 bit video memory address, used for screen refresh. The low order eight bit register is at index 0xD.

Bit	R/W	Description
7:0	R/W	Display Screen Start Address Upper Byte Bits.

5.6.15 Index 0xD-Start Address Low (0x3B5/0x3D5)

The lower order eight bits of the 16 bit video memory address.

Bit	R/W	Description
7:0	R/W	Start Address Low Byte.

5.6.16 Index 0xE-Cursor Location High (0x3B5/0x3D5)

The eight higher order bits of 16 bit cursor location in VGA modes. For the lower order eight bits, see the **Cursor Location Low Register** at index 0xF.

Bit	R/W	Description
7:0	R/W	Cursor Address Upper Byte Bits.

5.6.17 Index 0xF-Cursor Location Low (0x3B5/0x3D5)

Bit	R/W	Description
7:0	R/W	Cursor Address Lower Byte Bits. The lower order eight bits of the 16 bit video memory address.

5.6.18 Index 0x10-Vertical Retrace Start (0x3B5/0x3D5)

The lower eight bits of the ten bit **Vertical Retrace Start Register**. Bits 8 and 9 are located in the **Overflow Register** (index = 0x7). Bit 10 is in the **Vertical Extension Register** (index 0x1B) bit 6.

Bit	R/W	Description
7:0	R/W	Vertical Sync Start Pulse Lower Eight Bits.

5.6.19 Index 0x11-Vertical Retrace End (0x3B5/0x3D5)

Bit	R/W	Description
7	R/W	CRTC Registers Write Protect. When this bit is 0, writes to CRT index registers 0x0 to 0x7 are enabled. When this bit is 1, writes to CRT Controller index registers in the range of index 0x0 to 0x7 are protected except line compare bit 4 in the Overflow Register 0x7.
6	R/W	DRAM Refresh/Horizontal Scan Line. Historically, this register selected DRAM refresh cycles per horizontal scan line. This function is not implemented.
5	R/W	Enable Vertical Retrace Interrupt. (0=Enable, 1= Disable)
4	R/W	Clear Vertical Retrace Interrupt. (0=Clear Vertical retrace interrupt, 1= Allow an interrupt to be generated after the last displayed scan of the frame has occurred (i.e., the start of the bottom border).
3:0	R/W	Vertical Retrace End. This register specifies the scan count at which vertical sync becomes inactive. For retrace signal pulse width W, add scan counter for W to the value of the Vertical Retrace Start Register. The 4 bit result is written in the Vertical Retrace End Register.

5.6.20 Index 0x12-Vertical Display Enable End (0x3B5/0x3D5)

This register specifies the eight lower bits of ten bit register that defines where the active display frame ends. The programmed count is in scan lines minus 1. Bit 8 and 9 are in the **Overflow Register** (index 0x7) at bit positions 1 and 6 respectively. Bit 10 is in the **Vertical Extension Register** (index 0x1b) bit 2.

Bit	R/W	Description
7:0	R/W	Vertical Display Enable End Lower Eight Bits.

5.6.21 Index 0x13-Offset (0x3B5/0x3D5)

This register specifies the width of display memory in terms of an offset from the current row start address to the next character row. The offset value is a word address adjusted for word or double word display memory access.

Bit	R/W	Description
7:0	R/W	Logical Line Screen Width.

5.6.22 Index 0x14-Underline Location (0x3B5/0x3D5)

Bit	R/W	Description
7	R	Reserved.
6	R/W	Double Word Mode. (0 = Display memory addressed for byte or word access. 1= Display memory addressed for double word access).
5	R/W	Count By 4 For Double Word Access. (0= Memory address counter clocked for byte or word access, 1 = Memory address counter is clocked at the character clock divided by 4.)
4:0	R/W	Underline Location. These bits specify the row scan counter value within a character matrix where under line is to be displayed. Load a value 1 less than the desired scan line number.

5.6.23 Index 0x15-Start Vertical Blank (0x3B5/0x3D5)

The lower eight bits of the ten bit Start Vertical Blank Register. Bit 8 is in the **Overflow Register** (index = 0x7) and bit 9 is in the **Maximum Scan Line Register** (index = 0x9). The ten bit value is reduced by 1 from the desired scan line count where the vertical blanking signal starts.

Bit	R/W	Description
7:0	R/W	Start Vertical Blank Lower Eight Bits.

5.6.24 Index 0x16-End Vertical Blank (0x3B5/0x3D5)

Bit	R/W	Description
7:0	R/W	Vertical Blank Inactive Count.

End Vertical Blank is an 8 bit value calculated as follows:

End Vertical Blank = (Start Vertical Blank - 1) + (Vertical Blank signal width in scan lines).

5.6.25 Index 0x17-CRTC Mode Control (0x3B5/0x3D5)

Bit	R/W	Description
7	R/W	Sync Enable. (0= retrace outputs disabled, 1= retrace outputs enabled)
6	R/W	Word or Byte Mode. (0= Word address mode, 1= Byte address mode)
5	R/W	Address Wrap. In word address mode, setting this bit to 0 enables bit 13 to appear at MAO, otherwise in byte address mode bit 0 appears on MAO. Setting this bit to 1 selects MA15 for odd/even mode.
4	R	Reserved.
3	R/W	Count by 2 (0 = Character clock increments memory address counter, 1= Character clock divided by 2 increments the address counter).
2	R/W	Horizontal Retrace Clock Rate Select For Vertical Timing Counter. 0= Normal, 1= Selects horizontal retrace clock rate divided by 2.)
1	R/W	Select Row Scan Counter.0=Selects row scan counter bit 1 as output at MA14 address pin.1 Selects bit 14 of the CRTC address counter as output at MA14 pin.
0	R/W	6845 CRT Controller compatibility mode support for CGA operation. 0 = Row scan address bit 0 is substituted for memory address bit 13 at MA13 output pin during active display time. 1=Enable memory address pin 13 to be output at MA13 address pin.

5.6.26 Index 0x18-Line Compare (0x3B5/0x3D5)

Bit	R/W	Description
7:0	R/W	Line Compare Lower Eight Bits. Lower eight bits of the ten bit Scan Line Compare Register. Bit 8 is in the **Overflow Register** (index = 0x7) and bit 9 is in the **Maximum Scan Line Register** (index = 0x9). When the vertical counter reaches this value, the internal start of the line counter is cleared.

5.6.27 Index 0x1A-Horizontal Extension Register (0x3B5/0x3D5)

This register is an extension of the VGA core in order to increase the total horizontal resolution available to Avenger. This register is only active when **VGAINIT0** bit 6 is '1'.

Bit	R/W	Description	Base Index
7	R/W	Horizontal Retrace End bit 5.	-
6	R/W	Horizontal Retrace Start bit 8	0x4
5	R/W	Horizontal Blank End bit 6.	-
4	R/W	Horizontal Blank Start bit 8.	0x3
3	R/W	Reserved.	-
2	R/W	Horizontal Display Enable End bit 8.	0x1
1	R/W	Reserved.	-
0	R/W	Horizontal Total bit 8.	0x0

5.6.28 Index 0x1B-Vertical Extension Register (0x3B5/0x3D5)

This register is an extension of the VGA core in order to increase the total Vertical resolution available to Avenger. This register is only active when **VGAINIT0** bit 6 is '1'.

Bit	R/W	Description	Base Index
7	R/W	Reserved	-
6	R/W	Vertical Retrace Start bit 10	0x10
5	R/W	Reserved.	-
4	R/W	Vertical Blank Start bit 10.	0x15
3	R/W	Reserved.	-
2	R/W	Vertical Display Enable End bit 10	0x12
1	R/W	Reserved.	-
0	R/W	Vertical Total bit 10.	0x6

5.6.29 Index 0x1C-PCI Config/Extension Byte 0 (0x3B5/0x3D5)

On power up, the AVENGER is configured to allow read back of the PCI configuration information a byte at a time through this register. In order to use this feature, first follow the standard wake up sequence. To selectively read back configuration information, write the index into this register. Data read back from this register is the configuration byte at that index.

Bit	R/W	Description
7:0	R/W	PCI Configuration/Scratch Pad Register.

The use of the extended register space is decoded as follows:

VGAINIT0		
7	6	Description
0	0	Allow Configuration data to be read back from PCI (Indexed)
0	1	Extended registers Are scratch Pad
1	X	Extended registers Disabled

5.6.30 Index 0x1D-Extension Byte 1 (0x3B5/0x3D5)

This register is only active when **VGAINIT0** bit 6 is '1'

Bit	R/W	Description
7:0	R/W	Scratch Pad Register.

5.6.31 Index 0x1E-Extension Byte 2 (0x3B5/0x3D5)

This register is only active when **VGAINIT0** bit 6 is '1'

Bit	R/W	Description
7:0	R/W	Scratch Pad Register.

5.6.32 Index 0x1F-Extension Byte 3 (0x3B5/0x3D5)

This register is only active when **VGAINIT0** bit 6 is '1'

Bit	R/W	Description
7:0	R/W	Scratch Pad Register.

5.6.33 Index 0x20-Vertical Counter pre-load Low (0x3B5/0x3D5)

This register, in combination with index 0x20, allows the vertical counter to be pre-loaded for testing purposes. The vertical counter is pre-loaded on reset, which can be caused either through a hard reset or a soft reset. This register is only active when **VGAINIT0** bit 6 is '1'.

Bit	R/W	Description
7:0	R/W	Scratch Pad Register.

5.6.34 Index 0x21- Vertical Counter pre-load High(0x3B5/0x3D5)

This register is only active when **VGAINIT0** bit 6 is '1'

Bit	R/W	Description
2:0	R/W	Scratch Pad Register.

5.6.35 Index 0x22-Latch Read Back (0x3B5/0x3D5)

Bit	R/W	Description
7:0	R/W	Latch Data Register. This register reflects the contents of one of the four Graphics Data Controller latches. The plane selected for read back is determined by Graphics Controller **Read Map Select Register** (index 0x4) bits 0 and 1.

5.6.36 Index 0x24-Attribute Controller Index/Data State (0x3B5/0x3D5)

Bit	R/W	Description
7	R	Attribute Controller Index/Data State. When this is 1, the Attribute controller register is set to 'Data' state. When set to 0, the Attribute controller register is set to 'Index' state. Reading 0x3DA will always put the Attribute Controller back to Index State.
6:0	R	Reserved.

5.6.37 Index 0x26-Display Bypass/Attribute Controller Index (0x3B5/0x3D5)

Bit	R/W	Description
5	R	Display Bypass. Reflects the value of the Attribute Controller index register, bit 5.
4:0	R	Attribute index.

5.7 Graphics Controller Registers:

5.7.1 Graphics Controller Index Register (0x3CE)

Data written to this 8 register reflects the index of the Graphics Controller register space accessed through 0x3CF.

Bit	R/W	Description
7:4	R	Reserved.
3:0	R/W	Index for accesses at 0x3CF.

5.7.2 Index 0-Set/Reset (0x3CF)

Bit	R/W	Description
7:4	R	Reserved.
3:0	R/W	Set/Reset Map.

When the CPU executes display memory write with Write Mode 0 selected and the **Enable Set/Reset Register** (index = 0x1) activated, the eight bits of the value in this register, which have been operated on by the Mask Register, are then written to the corresponding display memory map. It is an eight fill operation. The map designations are defined below:

0	Reset.
1	Set.

5.7.3 Index 1-Enable Set/Reset (0x3CF)

Bit	R/W	Description
7:4	R	Reserved.
3:0	R/W	Enable Set/Reset Register (Index 0x0). When Write Mode 0 is selected, these bits enable memory map access defined by the **Set/Reset Register** (index = 0x0), and the respective memory map is written with the **Set/Reset Register** value.

5.7.4 Index 2-Color Compare (0x3CF)

The color compare contains the value to which all 8 bits of the corresponding memory map are compared. This comparison also occurs across all four maps, and a 1 is returned for the map positions where the bits of all four maps equal the **Color Compare Register**. If a system read is done with 3 = 0 for the **Graphics Mode Register** (index = 0x5), data is returned without comparison. Color compare map coding is shown below.

Bit	R/W	Description
7:4	R	Reserved.
3:0	R/W	Color Compare.

5.7.5 Index 3-Data Rotate (0x3CF)

Bit	R/W	Description
7:5	R	Reserved.
4:3	R/W	Function Select. Function select for any of the write mode operations defined in **the Graphics Mode Register** (index = 0x5) is defined in the following table.
2:0	R/W	Rotate Count. These bits specify number of positions of rotation to the right and is ineffective in write mode 2, defined by the **Graphics Mode Register** (index =0x5).

4	3	Function
0	0	Move
0	1	And
1	0	Or
1	1	Xor

5.7.6 Index 4-Read Map Select (0x3CF)

Bit	R/W	Description
7:2	R	Reserved.
1:0	R/W	Map Select. These bits select memory map in system read operations. It has no effect on color compare read mode. In odd/even modes, the value can be 0x0 or 0x1 to select chained maps 0 & 1 or value 0x2 or 0x3 to select the chained maps 2 & 3.

5.7.7 Index 5-Graphics Mode (0x3CF)

Bit	R/W	Description
7	R	Reserved.
6:5	R/W	Shift Mode. 00 = data is shifted out normally. 01 = data is shifted out Even/Odd 1x = 256 Color Mode shift
4	R/W	CGA compatible Odd/Even Mode. When set to '1' , Sequential addressing is as defined by bit 2 of the **Memory Mode Register** (index = 0x4) in the Sequencer Register. Even system addresses access maps 0 or 2 and odd system addresses access maps 1 or 3.
3	R/W	Read Mode. When set to 0, System reads data from memory maps selected by **Read Map Select Register** (index 0x4). This setting has no effect if bit 3 of the **Sequencer Memory Mode Register** = 1. When set to 1, System reads the comparison of the memory maps and the **Color Compare Register**.
2	R	Reserved.
1:0	R/W	Write Mode. The table on the following page defines the four write modes.

Bits 1:0	Write Mode	Description
00	0	CPU or data from the Set/Reset Register is written to graphics memory.
01	1	Latch data is written to graphics memory
10	2	Plane n is filled with data bit n
11	3	The addressed byte in each plane is filled with the value of the corresponding bits in the **Set/Reset Register** (index 0x0). The **Enable Set/Reset Register** (index 0x1) has no effect. Rotated CPU data is logically ANDed with the **Mask Register** (index 0x8).

5.7.8 Index 6-Miscellaneous (0x3CF)

Bit	R/W	Description
7:4	R	Reserved.
3:2	R/W	Memory Map 1,0 Display memory map control into the CPU address space is shown in the following table.
1	R/W	Odd/Even Mode. When set to 1, CPU address A0 is replaced by higher order address bit. A0 is then used to select odd or even maps. A0 = 0 selects map 0 or 2, while A0 = 1 selects map 1 or 3.
0	R/W	Graphics/Alphanumeric Mode. 0 = Alphanumeric mode, 1= Graphics mode.

3	2	Physical Address	Size	Typical Usage
0	0	0xA0000	128K	None
0	1	0xA0000	64K	EGA/VGA/Extended Graphics Modes
1	0	0xB0000	32K	Monochrome Text Modes
1	1	0xB8000	32K	Color Text / CGA Graphics Modes

5.7.9 Index 7-Color Don't Care (0x3CF)

Bit	R/W	Description
7:4	R	Reserved.
3:0	R/W	Memory Map Color Compare Operation. 1=Enable, 0 = Disable.

5.7.10 Index 8-Mask (0x3CF)

Mask operation applies simultaneously to all the four maps. In Write Modes 0 and 2, this register provides selective changes to any stored in the system latches during processor writes. Data must be first latched by reading the addressed byte. After setting the **Mask Register**, new data is written to the same byte in a subsequent operation. Mask operation is applicable to any data written by the processor.

Bit	R/W	Description
7:0	R/W	Mask. 0 = Mask, 1 = Disable mask.

5.8 Attribute Registers

5.8.1 Attribute Index Register (0x3C0)

The Attribute Index Register has an internal flip-flop, rather than an input bit, which controls the selection of the Address and Data Registers. Reading the Input Status Register 1 (port = 0x3BA/0x3DA) clears the flip flop and selects the Address Register, which is read through address 0x3C1 and written at address 0x3C0. Once the Address Register has been loaded with an index, the next write operation to 0x3C0 will load the Data Register. The flip-flop toggles between the Address and the Data Registers after every write to address hex 0x3C0, but does not toggle for reads to address 0x3C1.

Bit	R/W	Description
7:6	R	Reserved.
5	R/W	Palette Address Source. (0=Disable palette outputs, 1=Enable palette outputs.)
4:0	R/W	Attribute Controller Index Register Address Bits

5.8.2 Index 0x0 through 0xF-Palette Registers (0x3C0/3C1)

The Palette Registers are effectively a lookup table 6 bits wide by 16 levels deep. The purpose of this lookup table is to allow dynamic color mapping from the original video data stream. The palette provides a translation from 4 bits to 6 bits of data. The palette output data is either combined with the **Color Select Register** (index 0x14), or two the result of two shifts are appended together, resulting in an 8 bit video stream.

Bit	R/W	Description
7:6	R	Reserved.
5:0	R/W	Palette Pixel Colors.

5.8.3 Index 10-Attribute Mode Control Register (0x3C0)

Bit	R/W	Description
7	R/W	VID5, VID4 Select (0=Use palette outputs, 1=use Color select Register index 0x14.)
6	R/W	Pixel Width (0= one pixel every VCLK, 1 = one pixel every 2 VCLK)
5	R/W	Pixel Panning Compatibility. (0=Enable Pixel Pan on line compare, 1 = disable on line compare)
4	R	Reserved.
3	R/W	Background Intensity/Blink Selection. (0= MSB of attribute is background color, 1= MSB of attribute is blink)
2	R/W	Line Graphics Character Code. Setting this bit to 0 forces ninth dot to be the same color as background in line graphics character codes. Setting this bit to 1 forces the ninth dot character to be identical to the eighth character dot. Set this to zero for character fonts that do not utilize line graphics character codes.
1	R/W	Mono/Color Emulation. (0=Color, 1 = Mono)
0	R/W	Graphics/Alphanumeric Mode Enable. (0=alphanumeric, 1= graphics)

5.8.4 Index 11-Over Scan Control Register (0x3C0)

This register determines the over scan or border color. For monochrome displays, this register is set to 0.

Bit	R/W	Description
7:0	R/W	Over Scan/Border Color

5.8.5 Index 12-Color Plane Enable Register (0x3C0)

Bit	R/W	Description
7:6	R	Reserved.
5:4	R/W	Video Status Control. These bits select 2 out of 8 color outputs which can be read by the Input Status Register 1 (port = 0x3BA/0x3DA) bits 4 and 5.
3:0	R/W	Color Plane Enable. Setting a bit to 0 disables the respective color plane(s).

5.8.6 Index 13-Horizontal Pixel Panning Register (0x3C0)

These bits select pixel shift to the left horizontally. For 9 dots/character modes, up to 8 pixels can be shifted horizontally to the left. Likewise, for 8 dots/character up to 7 pixels can be shifted horizontally to the left. For 256 color, up to 3 position pixel shifts can occur.

Bit	R/W	Description
7:4	R	Reserved.
3:0	R/W	Horizontal Pixel Panning. See table.

ɮits [3:0]	bit text	56 color	)ther
0x0	1	0	0
0x1	2	½	1
0x2	3	1	2
0x3	4	1 ½	3
0x4	5	2	4
0x5	6	2 ½	5
0x6	7	3	6
0x7	8	3 ½	7
0x8-0xf	0	-1	-1

5.8.7 Index 14-Color Select Register (0x3C0)

Bit	R/W	Description
7:4	R	Reserved.
3:2	R/W	Color Value MSB. Two most two significant bits of the eight digit color value for the video DAC. They are normally used in all modes except 256 color graphics.
1:0	R/W	Substituted Color Value Bits. These bits can be substituted for VID5 an VID4 output by the Attribute Controller palette registers, to create eight color value. They are selected by the Attribute Controller **Mode Control Register** (index = 0x10).

5.9 Sequencer Registers

5.9.1 Sequencer Index Register (0x3c4)

Bit	R/W	Description
7:0	R/W	Index for accesses at 0x3c5.

5.9.2 Index 0-Reset (0x3c5)

Bit	R/W	Description
7:2	R	Reserved.
1	R/W	Synchronous Reset. 0=Video Timing is cleared and halted. This is used to synchronize changing the either bits 3 or 2 of the **Miscellaneous Output Register**. 1= Operational mode.
0	R/W	Asynchronous Reset. 0=Sequencer is cleared and halted asynchronously. This bit is used to force the Sequencer into a reset state, regardless of the operation it is performing. 1=Operational mode.

5.9.3 Index 1-Clocking Mode (0x3c5)

Bit	R/W	Description
7:6	R	Reserved
5	R/W	Screen Off. When this bit is set to 1 the screen turned off, all requests for video FIFO refresh are disabled, allowing additional bandwidth for other memory operations. SYNC signals remain active.
4	R/W	Video Serial Shift Register Loading. When this bit is 0, serial shift registers are loaded every character or every other character clock depending on bit 2 of this register; otherwise when this bit is 1, Serial shift registers loaded every 4^{th} character clock (32 fetches).
3	R/W	Dot Clock Selection (0= Normal dot clock selected by VCLK input frequency, 1 = Dot Clock divided by 2 (used for 320/360 pixel width display modes).
2	R/W	Shift Load. This is only effective if bit 4 of this register = 0. (0=Video serializers will be loaded every character clock, 1 = Video serializers are loaded every other character clock).
1	R	Reserved.
0	R/W	8/9 Dot Clock. (0= 9 dot wide character clock, 1 = 8 dot wide character clock)

5.9.4 Index 2-Map Mask (0x3c5)

Bit	R/W	Description
7:4	R	Reserved.
3:0	R/W	Map Enables. If a bit is 0, writing to the corresponding map(0-3) is disabled.

5.9.5 Index 3-Character Map Select (0x3c5)

If Sequencer Register index 4 bit 1 is 1, then the attribute byte 3 in text modes is redefined to control switching between character sets in alphanumeric modes. An attribute of 0 selects character map B, while a 1 selects character map A.

Bit	R/W	Description
7:6	R	Reserved.
5	R/W	Character Map A High Select. The Most Significant (MSB) of character map A along with bits 3 and 2, select the location of character map A as shown below.
4	R/W	Character Map B High Select. The MSB of character map B along with bits 1 and 0, select the location of character map B as shown below.
3:2	R/W	Character Map Select A. Refer to Character Map A Select table.
1:0	R/W	Character Map Select B. Refer to Character Map B Select table.

Bit			Map	Table Location
5	3	2	Selected	(Maps 2 or 3)
0	0	0	0	1st 8K
0	0	1	1	3rd 8K
0	1	0	2	5th 8K
0	1	1	3	7th 8K
1	0	0	4	2nd 8K
1	0	1	5	4th 8K
1	1	0	6	6th 8K
1	1	1	7	8th 8K

Character Map A Select

Bit			Map	Table Location
4	1	0	Selected	(Maps 2 or 3)
0	0	0	0	1st 8K
0	0	1	1	3rd 8K
0	1	0	2	5th 8K
0	1	1	3	7th 8K
1	0	0	4	2nd 8K
1	0	1	5	4th 8K
1	1	0	6	6th 8K
1	1	1	7	8th 8K

Character Map B Select

5.9.6 Index 4-Memory Mode (0x3c5)

Bit	R/W	Description
7:4	R	Reserved.
3	R/W	Chain 4 Maps. (0= Processor sequentially accesses data using map mask register, 1 = The two lower order video memory address pins (MAO,MA1) to select the map to be addressed)
2	R/W	Odd/Even. Bit 3 of this register must be 0 for this bit to be effective. (0=Odd/Even Mode, 1 = Normal)
1	R/W	Extend Memory. (0= restrict size to 16/32K, 1= allow 256K).
0	R	Reserved

5.10 RAMDAC Registers

5.10.1 RAMDAC Pixel Mask (0x3c6)

Bit	R/W	Description
7:0	R/W	RAMDAC pixel mask

The contents of this register are logically ANDed with the output of the VGA data stream before it is presented to the RAMDAC. The value of this register has no effect on modes other than VGA.

5.10.2 RAMDAC Read Index /Read Status (0x3c7)

Bit	R/W	Description
7:0	W	RAMDAC Read Index
1:0	R	RAMDAC State. 0 = a read operation is in effect, 3 = a write operation is in effect.

When data is written to this register, it causes the CLUT to go into a 'Read State'. It should be followed be three consecutive reads of 0x3c9 in order to retrieve the red, green and blue values of the CLUT. This index will auto increment following the completion of the last data read. Note that only the first 256 locations of the CLUT may be accessed via this port.

When data is read from this register, bits 1:0 indicate the read/write state of the CLUT.

5.11 RAMDAC Write Index (0x3c8)

Bit	R/W	Description
7:0	R/W	RAMDAC Write Index

When data is written to this register, it causes the CLUT to go into a 'Write State'. It should be followed be three consecutive writes of 0x3c9 in order to store the red, green and blue values of the CLUT. This index will auto increment following the completion of the last data write. Note that only the first 256 locations of the CLUT may be accessed via this port.

5.11.1 RAMDAC Data (0x3c9)

Bit	R/W	Description
7:0	R/W	RAMDAC palette data

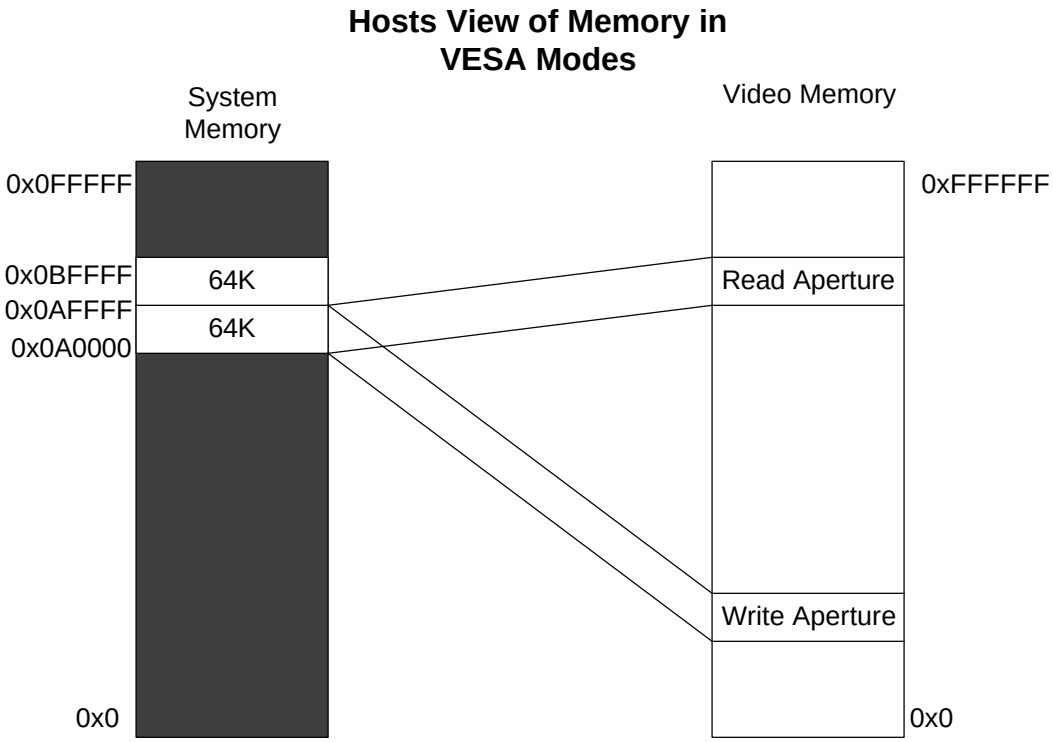
This register contains the data written to the CLUT. Data in this register is either 6 bit (VGA compatible) or 8 bit, as determined by VGAINIT0 bit 2. When data is in 6 bit format, the 2 MSBs are replicated into the 2 LSBs to maintain full scale and linearity on the DAC.

6. Accessing memory in VESA modes

VGA is restricted to only see 128K of memory through 0x0A0000. This supports baseline VGA graphics modes well; however, extended resolutions and video color depths in VESA modes require use of more memory than that allowed by the VGA standard.

Access to the entire frame buffer is available in VESA modes through a method of re-mapping the 0x0A0000 host memory space into part of the video memory. Memory accessed through 0x0A0000 in VESA modes is unaffected by the settings of the **Graphics Control** or **Sequencer Registers**.

There are two aperture controls, one for reading memory and one for writing memory. This allows memory to be moved from addresses greater than 64K apart without frequently modifying the aperture pointers. Each aperture can point to video memory anywhere along a 32K boundary.



7. 2D

7.1 2D Register Map

Memory Base 0: Offset 0x0100000

Register Name	Address	Reg	Bits	R/W	Description
status	0x000(0)	0x0	31:0	R	Avenger status register
intCtrl	0x004(4)	0x1	31:0	R/W	Interrupt control and status
clip0Min	0x008(8)	0x2	28:0	R/W	Min X & Y clip values when clip select is 0
clip0Max	0x00c(12)	0x3	28:0	R/W	Max X & Y clip values when clip select is 0
dstBaseAddr	0x010(16)	0x4	23:0	R/W	Destination base address
dstFormat	0x014(20)	0x5	17:0	R/W	Destination stride and bits per pixel
srcColorkeyMin	0x018(24)	0x6	23:0	R/W	Source Colorkey range (min)
srcColorkeyMax	0x01c(28)	0x7	23:0	R/W	Source Colorkey range (max)
dstColorkeyMin	0x020(32)	0x8	23:0	R/W	Destination Colorkey range (min)
dstColorkeyMax	0x024(36)	0x9	23:0	R/W	Destination Colorkey range (max)
bresError0	0x028(40)	0xA	31:0	R/W	Initial error for lines, right edges & stretch blt x
bresError1	0x02c(44)	0xB	31:0	R/W	Initial error for left poly edges & stretch blt y
rop	0x030(48)	0xC	31:0	R/W	4 Ternary Raster operations
srcBaseAddr	0x034(52)	0xD	23:0	R/W	Source base address
commandExtra	0x038(56)	0xE	31:0	R/W	Extra control bits
lineStipple	0x03c(60)	0xF	31:0	R/W	Monochrome pattern for lines
lineStyle	0x040(64)	0x10	28:0	R/W	Style register for lines
pattern0Alias	0x044(68)	0x11	31:0	R/W	Alias to colorPattern(0)
pattern1Alias	0x048(72)	0x12	31:0	R/W	Alias to colorPattern(1)
clip1Min	0x04c(76)	0x13	28:0	R/W	Min X & Y clip values when clip select is 1
clip1Max	0x050(80)	0x14	28:0	R/W	Max X & Y clip values when clip select is 1
srcFormat	0x054(84)	0x15	18:0	R/W	Source stride and bits per pixel
srcSize	0x058(88)	0x16	28:0	R/W	Height and width of source for stretch blts
srcXY	0x05c(92)	0x17	28:0	R/W	Starting pixel of blt source data Starting position for lines Top-most point for a polygon fill
colorBack	0x060(96)	0x18	31:0	R/W	Background color
colorFore	0x064(100)	0x19	31:0	R/W	Foreground color
dstSize	0x068(104)	0x1A	28:0	R/W	Destination width and height for blts and rectangle fills
dstXY	0x06c(108)	0x1B	28:0	R/W	Starting X and Y of destination for blts End point for lines
command	0x070(112)	0x1C	31:0	R/W	2D command mode & control bits
RESERVED	0x074(116)	0x1D	31:0		Do not write
RESERVED	0x078(120)	0x1E	31:0		Do not write
RESERVED	0x07c(124)	0x1F	31:0		Do not write
launchArea	0x080(128) to 0x0ff(255)	0x20 to 0x3F	31:0	R	Initiates 2D commands
colorPattern	0x100(256) to 0x1fc(508)	0x40 to 0x7F	31:0	R/W	Pattern Registers (64 entries)

7.2 Register Descriptions

The 2D register set is described in the sections below.

All 2D registers can be read, and all registers except for the status register are fully write-able. Reading a 2D register will always return the value that will be used if a new operation is begun without writing a new value to that register. This value will either be the last value written to the register, or, if an operation has been performed since the value was written, the value after all operations have completed.

All registers for the 2D section are unsigned unless specified otherwise.

7.2.1 status Register

The **status** register provides a way for the CPU to interrogate the graphics processor about its current state and FIFO availability. The **status** register is read only, but writing to **status** clears any Avenger generated PCI interrupts. For the definition of this register please see section XXX on PCI configuration and Initialization registers.

7.2.2 command Register

The command register sets the command mode for the 2D engine, as well as selecting a number of options.

Bits (3:0) set the command mode for the 2D drawing engine as shown in the table below. If bit(8) is set, the command will be initiated as soon as the **command** register is written. If bit(8) is cleared, drawing will be initiated by a write to the launch area. For descriptions and examples of each command, see the 2D launch area section.

Command[3:0]	Command
0	Nop - wait for idle
1	Screen to screen blt
2	Screen to screen stretch blt
3	Host to screen blt
4	Host to screen stretch blt
5	Rectangle fill
6	Line
7	Polyline
8	Polygon fill
13	Write Sgram Mode register
14	Write Sgram Mask register
15	Write Sgram Color register

Setting Bit(9) makes line drawing reversible. If this bit is set, drawing a line from point A to point B will result in the same pixels being drawn as drawing a line from point B to point A.

Bits(11:10) control the value placed in dstXY after each blt or rectangle fill command is executed. If bit(10) is 0, dst_x is unchanged. If bit(10) is 1, dst_x gets dst_x + dst_width. If bit(11) is 0, dst_y is unchanged. If bit(11) is 1, dst_y gets dst_y + dst_height.

Bit(12) controls whether lines are stippled or solid. If bit(12) is 0, lines will be a solid color. If bit(12) is 1, lines will either be made up of either a two color pattern using **colorFore** and **colorBack** or will be a transparent stipple using **colorFore**, as determined by the transparency bit - bit(16).

Bit(13) controls the format of the pattern data. If bit(13) is set to 0, the pattern must be stored in the destination format. If it is set to 1, the pattern will be stored as a monochrome bitmap; Pattern registers 0 and 1 will be used as an 8x8x1bpp pattern, which will be expanded into the destination format using the **colorBack** and **colorFore** registers. Note that if Bit(13) is set, and Bit(16) is set to indicate that monochrome data is transparent, the pattern will be used to determine pixel transparency without regard to the contents of the ROP register.

Bits(15:14) control the direction of blting during screen-to-screen copies. Note that the corner of the source and destination rectangles passed in the **srcXY** and **dstXY** registers will change depending on the blting direction. Bit(15) also controls the direction of blting for host-to-screen copies. This can be used to flip a pixel map so that the top span in host memory is drawn as the bottom span on the screen. Note that the direction bits only apply to “pure” screen to screen blits, but not to stretch blits. Also, destination and source color keying, along with color conversions, cannot be used with right to left blits.

Bit(16) controls whether monochrome source bitmaps, and monochrome patterns will be transparent or opaque. When bit(16) is 0, source bitmaps are opaque; a 0 in the bitmap will result in **colorBack** being written to the destination. When bit(16) is 1, source bitmaps and monochrome patterns are transparent. In this case, a 0 in the bitmap will result in the corresponding destination pixel being left unchanged.

The X and Y pattern offsets give the coordinates within the pattern of the pixel which corresponds to the destination pixel pointed to by the destination base address register. In other words, if a pattern fill is performed which covers the origin, pixel (0,0) in the destination pixel map will be written with the color in pattern pixel (x_pat_offset, y_pat_offset).

Bit(23) controls whether the clip0 or clip1 registers will be used for clipping. When bit(31) is 0, clipping values from **clip0Min** and **clip0Max** will be used, when bit(31) is 1, clipping values from **clip1Min** and **clip1Max** will be used.

Bits(31:24) contain ROP0, the ternary ROP that is used when colorkeying is disabled. For more information on ROPs, see the description of the **rop** register.

Command

Bit	Description
3:0	Command
7:4	RESERVED
8	Initiate command (1=initiate command immediately, 0 = wait for launch write)
9	Reversible lines (1=reversible, 0=non-reversible)
10	Increment destination x-start after blt or rectangle command (1=increment, 0=don’t)
11	Increment destination y-start after blt or rectangle command (1=increment, 0=don’t)
12	Stipple line mode (1 = stippled lines, 0 = solid lines)
13	Pattern Format (1 = monochrome, 0 = color)
14	X direction (0 = left to right, 1 = right to left)
15	Y direction (0 = top to bottom, 1 = bottom to top)
16	Transparent monochrome (1 = transparent, 0 = opaque)
19:17	X pattern offset
22:20	Y pattern offset
23	Clip select (0=clip0 registers, 1 = clip1 registers)
31:24	ROP0

7.2.3 commandExtra Register

This register contains miscellaneous control bits in addition to those in the command register.

Bits(1:0) enable colorkeying, if the bit is 0, colorkeying is disabled. Enabling source colorkeying with monochrome source, or in line, polyline, polygon, or rectangle modes has no effect. For further explanation of these bits, see the description of the colorkey registers.

If bit(2) is set, the current command, and any following it will not proceed until the next vertical blanking period begins. Wait for Vsync should not be used when performing non-DMA host blts.

If bit(3) is set, only row 0 of the pattern will be used, rather than the usual 8 pattern rows.

Command

Bit	Description
0	Enable source colorkey (1=source colorkeying enabled, 0=source colorkeying disabled)
1	Enable destination colorkey (1=enable dst colorkeying, 0=disable dst colorkeying)
2	Wait for Vsync (1=wait for vsync, 0=execute immediately)
3	Force pattern row 0 (1 = use only row 0, 0 = use all 8 pattern rows)

7.2.4 colorBack and colorFore Registers

The **colorBack** and **colorFore** registers specify the foreground and background colors used in solid-fill and monochrome bitmap operations, and operations using a monochrome pattern. The color registers must be stored in the destination color format.

The following tables shows the format of the color registers for each destination format.

P = palette index
R = red color channel
G = green color channel
B = blue color channel

Dst Format	Bits stored
8 bpp	0000_0000_0000_0000_0000_0000_PPPP_PPPP
15 bpp	0000_0000_0000_0000_ARRR_RRGG_GGGB_BBBB
16 bpp	0000_0000_0000_0000_RRRR_RGGG_GGGB_BBBB
24 bpp	0000_0000_RRRR_RRRR_GGGG_GGGG_BBBB_BBBB
32 bpp	AAAA_AAAA_RRRR_RRRR_GGGG_GGGG_BBBB_BBBB

colorFore

Bit	Description
31:0	foreground color

colorBack

Bit	Description
31:0	background color

7.2.5 Pattern Registers

The **pattern** registers contain an 8 pixel by 8 pixel pattern. The pixels must be either in the color format of the destination surface, or in 1bpp (monochrome) format. The pixels are to be written to the **pattern** registers in packed format. So, only registers 0 and 1 will be used for monochrome patterns, registers 0 through 15 will be used when the destination is 8 bpp, registers 0 through 31 will be used when the destination is 16 bpp.

Pixels should be written into the **pattern** registers starting with the upper left-hand corner of the pattern, proceeding horizontally left to right, and then vertically top to bottom. The least-significant bits of **pattern**[0] should always contain pixel(0,0) of a color pattern.

The table below give the bit position of monochrome pixels within the **pattern** registers. The bits are numbered such that bit(0) represents the lsb of a register, and bit(31) represents the msb.

7.2.5.1 Order of pixel storage in the pattern registers for a monochrome pattern

pattern(0)

Row 0	7	6	5	4	3	2	1	0
Row 1	15	14	13	12	11	10	9	8
Row 2	23	22	21	20	19	18	17	16
Row 3	31	30	29	28	27	26	25	24

pattern(1)

Row 4	7	6	5	4	3	2	1	0
Row 5	15	14	13	12	11	10	9	8
Row 6	23	22	21	20	19	18	17	16
Row 7	31	30	29	28	27	26	25	24

pattern(0-64)

Bit	Description
31:0	pattern color data

7.2.6 srcBaseAddr and dstBaseAddr Registers

Bits(23:0) of these registers contain the addresses of the pixels at x=0, y=0 on the source and destination surfaces in frame-buffer memory. Bit(31) of each register specifies whether the address points to tiled or linear memory.

The **srcBaseAddr** register is used only for screen-to-screen blts. For host-blts, the alignment of the initial pixel sent from the host is determined by the x entry in the **srcXY** register.

For YUYV422 and UYVY422 surfaces, the base address must be dword aligned. Thus bits(1:0) of **srcBaseAddr** must be 0.

SrcBaseAddr

Bit	Description

23:0	Source base address
30:24	RESERVED
31	Source memory is tiled

dstBaseAddr

Bit	Description
23:0	Destination base address
30:24	RESERVED
31	Destination memory is tiled

7.2.7 srcSize and dstSize Registers

These registers are used only for blts and rectangle fills. They contain the height and width in pixels of the source and destination rectangles. The **srcSize** register will only be used in Stretch-blt modes. For non-stretched blts, the blt source size will be the same as the blt destination size, determined by the **dstSize** register.

srcSize

Bit	Description
12:0	Blt Source Width
15:13	RESERVED
28:16	Blt Source Height
31:29	RESERVED

dstSize

Bit	Description
12:0	Blt Destination Width
15:13	RESERVED
28:16	Blt Destination Height
31:29	RESERVED

7.2.8 srcXY and dstXY Registers

During screen-to-screen blts, the **srcXY** registers sets the position from which blt data will be read. Note that the starting position for a blit depends on the direction of the blt as shown in the table below. For lines and polylines, **srcXY** is the starting point of the first line segment. For polygons, the **srcXY** should be the topmost vertex of the polygon - that is, the vertex with the lowest y value. If there are multiple vertices sharing the lowest y value, the **srcXY** should be set to the leftmost vertex with that y value. Reading the **srcXY** register while in polygon mode will always return the last polygon vertex that the host sent for the left side of the polygon.

The values in the **srcXY** are signed, however for blts **srcXY** must contain only positive values.

During host-to-screen blts, only the x entry of the **srcXY** register is used. This entry determines the alignment of the initial pixel in the blt within the first dword sent from the host. For monochrome bitmaps, bits[4:0] are used to determine the bit position within the dword of the initial pixel. For color bitmaps, bits[1:0] are give the position within the dword of the first byte of pixel data. Host blts are always performed left-to-right (the x-direction bit in the **command** register is ignored), so the offset given will always be that of the leftmost pixel in the first span. The alignment of the initial pixel of all spans

after the first is determined by adding the source stride (from the **srcFormat** register) to the alignment of the previous span.

For blts, the **dstXY** should be the starting pixel of destination rectangle as shown in the table below. For line and polyline modes, the **dstXY** will be the endpoint of the first line segment.

In polygon mode, the **dstXY** register is used to store the last vertex sent for the right side of the polygon. If command[8] is set when the command register is written in polygon mode, the value from **srcXY** will be copied to **dstXY**. If command[8] is cleared, **dstXY** can be written with the rightmost pixel in the top span of the polygon.

Command[15:14]	Starting X/Y
00	Upper Left-hand corner
01	Upper Right-hand corner
10	Lower Left-hand corner
11	Lower Right-hand corner

dstXY

Bit	Description
12:0	Signed X position on the destination surface
15:14	RESERVED
28:16	Signed Y position on the destination surface
31:30	RESERVED

srcXY

Bit	Description
12:0	Signed x position of the first source pixel
15:14	RESERVED
28:16	Signed y position of the first source pixel
31:30	RESERVED

7.2.9 srcFormat and dstFormat Registers

These register specify the format and strides of the source and destination surfaces

For linear surfaces, the stride of a pixel map is the number of bytes between the starting addresses of adjacent scan lines. For these surfaces, the units of the stride is always bytes, regardless of the pixel format.

For tiled surfaces, the stride is a tile-stride. It's units are tiles, and only bits(6:0) are used.

The number of bits per pixel is determined as described by the tables below. The '32 bpp' format contains 24 bits of RGB, along with a byte of unused data, the '24 bpp' is packed 24 bit color.

Data coming through the host port can be byte swizzled to allow conversion between big and little endian formats, as selected by Bit 19 and 20 of src Format register. If both byte and word swizzling are enabled, the byte swizzling occurs first, followed by word swizzling.

The source packing bits control how the stride of the source will be determined during blts. If both bits are zero, the stride is set by the stride entry. Otherwise, the stride is based off of the width of the blt being

performed, as shown in the table below. The stride will equal the number of bytes in a row of the rectangle being blted plus as many bytes as are required to get the necessary alignment. Packed source and tiled surfaces are mutually exclusive - you cannot have packed source on a tiled surface.

For YUYV422 and UYVY422 source formats, linear strides must always be a dword multiple. Thus, bits(1:0) of the **srcFormat** register must be 0.

When necessary, the blt engine will convert source pixels to the destination format.

When source pixels in 15bpp or 16bpp format are converted to 24bpp or 32bpp, color conversion is performed by replicating the msbs of each channel into the extra lsbs required. When pixels are converted from 32bpp or 24bpp formats to 15 or 16bpp, 16bpp, the extra lsbs are removed from each channel. When any non-32bpp format is converted to 32bpp, the 8msbs of each pixel (i.e. the alpha channel) are filled with zeros.

Destination pixel formats are stored as shown in the description of the **colorFore** and **colorBack** registers. RGB source formats match these, the other source formats are shown in the table below. For monochrome source, p0 represents the leftmost pixel on the screen and p31 represents the rightmost. For YUV formats, ya represents the left pixel and yb represents the pixel to the right of ya, etc. Thus, ya7 is the msb of the y channel for the left pixel and ya0 is the lsb of the y channel for the left pixel. In the diagram, the dword with the lower address (which will be quadword aligned) is shown first, followed by the dword with the higher address.

Source formats

Monochrome

p24 p25 p26 p27 p28 p29 p30 p31 p16 p17 p18 p19 p20 p21 p22 p23 p8 p9 p10 p11 p12 p13 p14 p15 p0 p1 p2 p3 p4 p5 p6 p7

UYVY 4:2:2

yb7 yb6 yb5 yb4 yb3 yb2 yb1 yb0 v7 v6 v5 v4 v3 v2 v1 v0 ya7 ya6 ya5 ya4 ya3 ya2 ya1 ya0 u7 u6 u5 u4 u3 u2 u1 u0

YUYV 4:2:2

v7 v6 v5 v4 v3 v2 v1 v0 yb7 yb6 yb5 yb4 yb3 yb2 yb1 yb0 u7 u6 u5 u4 u3 u2 u1 u0 ya7 ya6 ya5 ya4 ya3 ya2 ya1 ya0

Methods of color translation used for Blts

	1bpp src	**8bpp src**	**15bpp src**	**16bpp src**	**24bpp src**	**32bpp src**	**YUV src**
8bpp dst	color registers	direct or palette	not supported	not supported	not supported	not supported	not supported
15bpp dst	color registers	not supported	direct	lsb removal	lsb removal	lsb removal, alpha dropped	YUV => RGB
16bpp dst	color registers	not supported	msb duplication	direct	lsb removal	lsb removal, alpha dropped	YUV => RGB
24bpp dst	color registers	not supported	msb duplication	msb duplication	direct	direct, alpha dropped	YUV => RGB
32bpp dst	color registers	not supported	msb duplication, zero alpha	msb duplication, zero alpha	rgb direct, zero alpha	direct	YUV => RGB zero alpha

srcFormat

Bit	Description
13:0	Source Stride in bytes or tiles
15:14	RESERVED
19:16	Source color format: 1, 8, 16, 24, 32 bpp RGB, YUYV422, UYVY422
20	Host port byte swizzle (1=enable)
21	Host port word swizzle (1=enable)
23:22	Source packing
31:24	RESERVED

dstFormat

Bit	Description
13:0	Destination Stride in bytes or tiles
15:14	RESERVED
18:16	Destination bits per pixel: 8, 15, 16, 24, or 32
31:19	RESERVED

srcFormat [19:16]	Source Format
0	1 bpp mono
1	8 bpp palettized
3	16 bpp RGB
4	24 bpp RGB
5	32 bpp RGB
8	packed 4:2:2 YUYV
9	packed 4:2:2 UYVY

dstFormat [18:16]	Destination Bpp
1	8
3	16
4	24
5	32

srcFormat[23:22]	Packing	Stride calculation
0	Use stride register	srcFormat[13:0]
1	Byte packed	ceil(src_width * src_bpp/8)
2	Word packed	ceil(src_width * src_bpp/16)*2
3	Double-word packed	ceil(src_width * src_bpp/32)*4

7.2.10 clip0Min, clip0Max, clip1Min, and clip1Max Registers

The clip registers define the maximum and minimum x & y values of pixel that can be written in the destination pixel map. There are two sets of clip registers, however, only one set is used at a time, as determined by the clip select bit in the **command** register.

Clipping is inclusive of the minimum values, and exclusive of the maximum values. Thus if the clip select bit is zero, only pixels with x values in the range [clip0Min_x, clip0Max_x) and y values in the range [clip0Min_y, clip0Max_y) will be written.

clip0Min

Bit	Description
11:0	x minimum clip when clip select is 0

15:12	RESERVED
27:16	y minimum clip when clip select is 0
31:28	RESERVED

clip0Max

Bit	Description
11:0	x maximum clip when clip select is 0
15:12	RESERVED
27:16	y maximum clip when clip select is 0
31:28	RESERVED

clip1Min

Bit	Description
11:0	x minimum clip when clip select is 1
15:12	RESERVED
27:16	y minimum clip when clip select is 1
31:28	RESERVED

clip1Max

Bit	Description
11:0	x maximum clip when clip select is 1
15:12	RESERVED
27:16	y maximum clip when clip select is 1
31:28	RESERVED

7.2.11 colorkey Registers

These registers define the range of colors that will be transparent when color keying is enabled.

Different ROPs are selected for each pixel depending the result of that pixels colorkey test. A source pixel passes the colorkey test if it is within the inclusive range defined by the **srcColorkeyMin** and **srcColorkeyMax** registers. A destination pixel passes the colorkey test if it is within the inclusive range defined by the **dstColorkeyMin** and **dstColorkeyMax** registers.

For Pixels with 8bpp formats, the color indices are compared directly. For pixels with 16, 24, or 32bpp formats, each color channel (R, G, and B) is compared separately, and each channel must pass for the colorkey test to be passed. In the 32bpp format, the upper 8 bits are ignored during colorkey testing. Source colorkeying cannot be enabled if the source format is 1 bpp.

If colorkeying is disabled for the source or destination surfaces, that colorkey test is failed.

For further information on ROP selection by the colorkey test results, see the description of the ROP register.

The colorkey test uses the following formula:
pass = (((color>=colorkey_min) && (color<=colorkey_max)) && colorkey_enable)

srcColorkeyMin

Bit	Description
23:0	minimum color key value for source pixels

31:24	RESERVED

srcColorkeyMax

Bit	Description
23:0	maximum color key value for source pixels
31:24	RESERVED

dstColorkeyMin

Bit	Description
23:0	minimum color key value for destination pixels
31:24	RESERVED

dstColorkeyMax

Bit	Description
23:0	maximum color key value for destination pixels
31:24	RESERVED

7.2.12 rop Register

This is a set of ternary ROPs used to determine how the source, destination, and pattern pixels will be combined. The default ROP, ROP0 is stored in the command register. Which of the four ROPs will be used is determined on a per-pixel basis, based on the results of the source and destination colorkey tests, as shown in the following table:

Source Color Key Test	Destination Color Key Test	ROP
Fail	Fail	ROP 0
Fail	Pass	ROP 1
Pass	Fail	ROP 2
Pass	Pass	ROP 3

rop

Bit	Description
7:0	ROP 1
15:8	ROP 2
23:16	ROP 3

7.2.13 lineStyle register

The **lineStyle** register specifies how lines will be drawn.

The bit pattern used for line stippling can be set to repeat every 1-32 bits, as set by the bit-mask size part of this register. The bit-mask size entry gives the number of bits *minus one* that will be used from the **lineStipple** register. Thus, if you want to use 2 bits to represent a dashed line, you would set the bit-mask size to 1.

Each bit from the **lineStipple** register will determine the color or transparency of from 1-256 pixels. The repeat count determines the number of pixels along the line that will be drawn (or skipped) for each bit in

the line pattern register. The number of pixels associated with each bit of the line pattern *minus one* must be written to the repeat count entry.

The start position give the offset within the line pattern register for the first pixel drawn in a line. It consists of an integer index of the current bit in the line pattern, and a fractional offset that will determine the number of pixels that will be drawn using that bit of the pattern. The number of pixels drawn using the initial bit in the line pattern will equal the repeat count (i.e. the repeat count entry+1) minus the fractional part of the start position. The bit positions within the **lineStipple** registers are numbered starting with the lsb at 0, going up to the msb at 31.

It is illegal to set the integer part of the stipple position to be greater than the bit-mask size. It is illegal to set the fractional part to be greater than the repeat count. If either part of the stipple position is too large, the behavior of the line drawing engine is undefined.

Writing the **lineStyle** register will cause the stipple position to be loaded from the register. If the **lineStyle** register is not written to between the execution of two line commands, the stipple position at the start of the new line will be whatever if was after the completion of the last line. If the **lineStyle** register is read while the 2D engine is idle, the stipple position read will always be that which will be used in the next line operation - thus, if the **lineStyle** register has been written since the last stippled line was drawn the value written will be returned, otherwise the value that remained after the last stippled line will returned. Reading the **lineStyle** register while the 2D engine is not idle will return an indeterminable value for the stipple position.

In the following examples,. ‘x’ represents a pixel colored with **colorFore**, ‘o’ represents a pixel colored with **colorBack** or that is transparent. ‘_S_’ Shows that the line engine is starting at bit 0 in the **lineStipple** register. ‘_’ shows that the line engine is using a new bit from the **lineStipple** register.

7.2.13.1 Example

Say the bit-mask size is set to 6 (thus, the entry in the register is 5) and the line pattern is:
lineStipple <= 010111b

The pixel pattern that will be repeated is:

repeat_count	repeating pixel pattern
1	x_x_x_o_x_o_S_x_x_x_o_x_o
2	xx_xx_xx_oo_xx_oo_S_xx_xx_xx_oo_xx_oo
3	xxx_xxx_xxx_ooo_xxx_ooo_S_xxx_xxx_xxx_ooo_xxx_ooo

7.2.13.2 Example

Say the repeat count is 5 (the register entry is 4), the integer part of the start position is 7, and the fractional part of the start position is 2. The color of the first 3 pixels drawn for the line will be determined by bit 7 in the line pattern register, the next 5 pixels will be determined by bit 8, and so on.

lineStyle <= 07020904h
lineStipple <= 1010110111b

pixels generated, where x=**colorFore** and o=**colorBack**:

xxx_ooooo_xxxxx_S_xxxxx_xxxxx_xxxxx_ooooo_xxxxx_xxxxx_ooooo_xxxxx_ooooo_xxxxx_S

7.2.13.3 Pseudo code for line pixel generation

Here is the pseudo-code for determining the color of pixels generated by the line engine:

```
<bit_position> = <start_position_integer>
<pixel_position> = <start_position_fraction>

while (<need_another_pixel>) {
   if ( <line_pattern> & (1 << <bit_position>) ) {
         <new_pixel_color> = <colorFore>
   } else {
         if (<transparent>) {
            <new_pixel_color> = <transparent>
         } else {
            <new_pixel_color> = <colorBack>
         }
   }

if ( <pixel_position> == <repeat_count> ) {
         <pixel_position> = 0
         if (<bit_position> == <bit_mask_size>) {
                  <bit_position> = 0;
         } else {
                  <bit_position> = <bit_position> + 1
         }
   } else {
         <pixel_position> = <pixel_position> +1
   }

}
```

lineStyle

Bit	Description
7:0	Repeat count
12:8	Stipple size
15:13	RESERVED
23:16	Start position - fractional part
28:24	Start position - integer part
31:29	RESERVED

7.2.14 lineStipple Register

The line bit-mask register contains a mask that determines how lines will be drawn. Bits that are ones will be drawn with the color in the **colorFore** register. Bits that are zeros will be filled with the color in the **colorBack** register, or will not be filled, depending on the 'transparent' bit in the command register. The pattern in the bit mask can be set to repeat every 1-32 bits, as set by the bit-mask size part of the line style register. If the bit-mask size is set to less than 31, some of the bits of the line bit-mask will not be used, starting with the most-significant bit. For example, if the bit-mask size is set to 7, bits 0-7 of the **lineStipple** register will contain the line bit-mask.

lineStipple

Bit	Description
31:0	Line bit-mask

7.2.15 bresenhamError registers

These registers allows the user to specify the initial Bresenham error terms used when performing line drawing, polygon drawing, and stretch blts. The Bresenham error terms are signed values.

Bit 31 of each registers determines whether or not the error term given in the lower bits will be used. If this bit is 0, the line and stretch blt engines will generate the initial error term automatically. If the bit is set to 1, the error term given in bits 16-0 will be used. If a bresenham error register is used, the register should be written with bit[31] set to 0 after completion of the operation, so that subsequent operations will not be affected.

bresError0 can be used to set the initial error value for lines, for the left edge of a polygon, and for blt stretching along the y-axis.

bresError1 can be used to set the initial error value for the right edge of a polygon, and for blt stretching along the x-axis.

bresError0

Bit	Description
15:0	Signed Bresenham error term for stretch blt y, lines, and left polygon edges
30:17	RESERVED
31	Use the error term given in bits 16-0

bresError1

Bit	Description
15:0	Signed Bresenham error term for stretch blt x and right polygon edges
30:17	RESERVED
31	Use the error term given in bits 16-0

7.3 Launch Area

7.3.1 Screen-to-screen Blt Mode

Writing the launch area while in screen-to-screen blt mode results in a rectangle being copied from one area of display memory to another. The position of the source rectangle is given by the write to the launch area. The write to the launch area will be used to fill the **srcXY** register.

screenBltLaunch

Bit	Description
12:0	X position of the source rectangle
15:13	RESERVED
28:16	Y position of the source rectangle
31:29	RESERVED

7.3.2 Screen-to-screen Stretch Blt Mode

Writing the launch area while in screen-to-screen blt mode results in pixels being copied from rectangle in display memory to another of a different size. The write to the launch area will be used to fill the **srcXY** register. The x and y direction bits do not apply to stretch blits. I.e., only top-down, left-to-right stretch blits can be done.

stretchBltLaunch

Bit	Description
12:0	X position of the source rectangle
15:13	RESERVED
28:16	Y position of the source rectangle
31:29	RESERVED

7.3.3 Host-to-screen Blt Mode

In host-to-screen blt mode, writes to the launch area should contain packed pixels to be used as source data. When performing a host-to-screen blt, the blt engine does not generate source addresses. However, it is still necessary for the driver to specify the **srcFormat**, in order for the blt engine to determine how the source data is packed. The driver must also write the **srcXY** register in order to specify the first byte or bit to use from the first dword. In monochrome source mode, the 5 lsbs will specify the initial bit. In all other modes, the 2 lsbs of **srcXY** will specify the initial byte of the initial span. The alignment of the first pixel of each span after the first is determined by adding the source stride (from the **srcFormat** register) to the alignment of the previous span.

If more data is written to the launch area than is required for the host blt specified, the extra data will be discarded, or may be used in the following host blt, if it is requested while the 2D is operating on the first hblt. If too little data is written to the launch area, the hblt will be aborted, and pixels on an incomplete span at the end of the host blt may or may not be drawn.

7.3.3.1 Host Blt Example 1

In this example, the driver is drawing text to a 1024x768x16bpp screen using monochrome bitmaps of various widths. The monochrome data is packed, with each row byte aligned. First, it sets up the necessary registers before giving the data specific to the first blt:

colorBack <= the background color

colorFore <= the foreground color

dstXY <= the starting position of the first character

dstBaseAddr <= base address of the primary surface

clip0Min <= 0x00000000

clip0Max <= 0xFFFFFFFF

command <= SRC_COPY || HOST_BLT_MODE = 0xCC000003

dstFormat <= 0x00030800

srcFormat <= 0x00400000

The command mode is set to host-to-screen blt, with all other features disabled. Since colorkeying is disabled, only ROP0 is needed. The format register sets the host format to unswizzled monochrome, using byte-packing. This means that the stride will not have to be set for each blt, but will be set to the number of bytes required to store the number of pixels in the source width (Since this is not a stretch blt, the source width equals the destination width, as set later in the **dstSize** register). The clip registers are set such that the results will not be clipped. Although this is a host to screen blt, the **srcXY** register must be set in order to specify the initial alignment of the bitmask. For this example, the source data begins with the lsb of the first dword of host data, so the **srcXY** register is set to zero.

Now, the driver is ready to start the first blt. It will blt a 11x7 pixel character.

dstSize <= 0x0007000B

srcXY <= 0x00000000

launch <= 0xc0608020

launch <= 0xC460C060

launch <= 0x3B806EC0

launch <= 0x00001100

7.3.3.2 Host Blt Example 2

In this example, the driver is drawing a pixel map

colorBack <= the background color

colorFore <= the foreground color

dstXY <= the starting position of the first character

clip0Min <= 0x00000000

clip0Max <= 0xFFFFFFFF

command <= SRC_COPY || HOST_BLT_MODE = 0xCC000003

srcFormat <= 0x00240000

The command mode is set to host-to-screen blt, with all other features disabled. Since colorkeying is disabled, only ROP0 is needed. The format register sets the host format to unswizzled monochrome, using byte-packing. This means that the stride will not have to be set for each blt, but will be set to the number of bytes required to store the number of pixels in the source width (Since this is not a stretch blt, the source width equals the destination width, as set later in the **dstSize** register). The clip registers are set such that the results will not be clipped. Although this is a host to screen blt, the **srcXY** register must be set in order to specify the initial alignment of the bitmask. For this example, the source data begins with the lsb of the first dword of host data, so the **srcXY** register is set to zero.

Now, the driver is ready to start the first blt. It will blt a 11x7 pixel character.

dstXY <= 0x0007000B

srcXY <= 0x00000000

launch <= 1st 2 rows

launch <= 2nd 2 rows

launch <= 3rd 2 rows

launch <= last row

hostBltLaunch

Bit	Description
31:0	Source pixel data

7.3.4 Host-to-screen Stretch Blt Mode

Writing the launch area while in host-to-screen blt mode results in the pixels written to the launch area being stretched onto the destination rectangle. Pixel data for Host-to-screen stretch blts is written just as for non-stretched host-to-screen blts, except when the destination height differs from the source height. In this case, the host must replicate or decimate the source spans to match the number of destinations spans required.

hostStretchLaunch

Bit	Description
31:0	Source pixel data

7.3.5 Rectangle Fill Mode

Rectangle fill mode is similar to screen-to-screen blt mode, but in this mode, the **colorFore** register is used as source data rather than data from display memory. The size of the rectangle is determined by the **dstSize** register. The write to the launch area gives the position of the destination rectangle, which is used to fill the **dstXY** register.

rectFillLaunch

Bit	Description
12:0	X position of the destination rectangle
15:13	RESERVED
28:16	Y position of the destination rectangle
31:29	RESERVED

7.3.6 Line Mode

Writing the launch area while in line mode will write the launch data to the **dstXY** register and draw a line from **srcXY** to **dstXY**. After the line has been drawn, **dstXY** is copied to **srcXY**. In line mode, all pixels in the line will be drawn (as specified by the line style register), including both the start and endpoint.

The ROP used for lines can use the pattern and the destination, but not source data. **colorFore** will be used in the ROP in place of source data. Source colorkeying must be turned off, destination colorkeying is allowed.

7.3.6.1 Line drawing example

srcXY <= 0x00020003 // line start-point = (3, 2)

lineStipple <= 0x00000006 // bit mask is 110 binary

lineStyle <= 0x02010202 // start position = 2 1/3, repeat count = 2, bit-mask size=2

colorBack <= BLACK

colorFore <= GREY

command <= LINE_MODE || OPAQUE

launch <= 0x000c0016 // line end-point = (22,12)

The line drawn will appear as shown below:

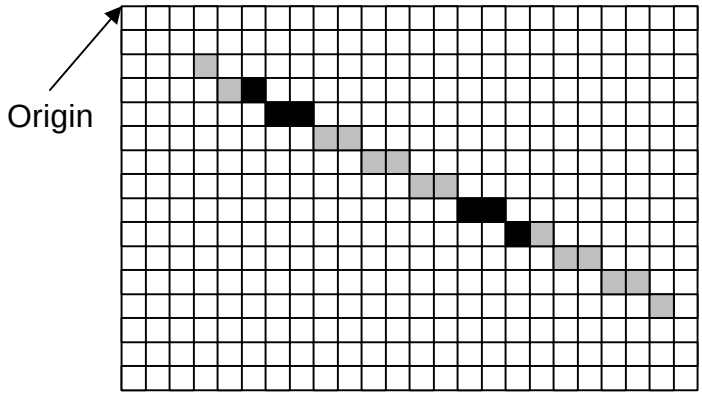


Figure 1

lineLaunch

Bit	Description
12:0	X position of the line endpoint
15:13	RESERVED
28:16	Y position of the line endpoint
31:29	RESERVED

7.3.7 Polyline Mode

Writing the launch area while in line mode will write the launch data to the **dstXY** register and draw a line from **srcXY** to **dstXY**. After the line has been drawn, **dstXY** is copied to **srcXY**. In polyline mode, the endpoint of the line (the pixel at dstXY) will not be written. This ensures that each pixel in a non-overlapping polyline will be written only once.

The ROP used for lines can use the pattern and the destination, but not source data. **colorFore** will be used in the ROP in place of source data. Source colorkeying must be turned off, destination colorkeying is allowed.

polylineLaunch

Bit	Description
12:0	X position of the line endpoint

15:13	RESERVED
28:16	Y position of the line endpoint
31:29	RESERVED

7.3.8 Polygon Fill Mode

The polygon fill mode can be used to draw simple polygons. A polygon may be drawn using the method described below if no horizontal span intersects more than two non-horizontal polygon edges. Polygons are drawn by first determining the top vertex - that is the vertex with the lowest y coordinate. The coordinates of this vertex should be written to the **srcXY** register. If multiple vertices share the lowest y coordinate, any vertex with the lowest y coordinate may be used as the starting point. If **command**[8] is set when the **command** register is written when **command**[3:0] indicates polygon mode, the value in the **srcXY** register will be copied to the **dstXY** register. The value in the **srcXY** register determines the starting point for the left side of the polygon, while the value in the **dstXY** register determines the starting point for the right side of the polygon. If bit[8] of the command register is not set, the starting position of the right side of the polygon can be set by writing to the **dstXY** register.

Once the starting vertex is set, as well as the desired colors, ROP, pattern, and options for the polygon fill, the polygon can be drawn by writing polygon vertices to the launch area. When multiple vertices share the lowest y coordinate, the starting vertex chosen will determine which of those vertices are on the 'right' edge of the polygon and which are on the 'left' edge. Pixels with the same y value as the starting point are on the left edge if they are to the left of the starting point.

For optimum performance, software should determine the leftmost and rightmost of all vertices that share the lowest y coordinate. The coordinates of the leftmost vertex should be written to **srcXY** and the coordinates of the rightmost vertex should be written to **dstXY**. When the **command** register is written, **command**[8] (the 'start command' bit) should be low.

In Polygon fill mode, polygon vertices should be written to the launch area in order of increasing y value. Whenever 2 vertices share the same y value, the leftmost vertex *must* be written first. The driver should keep track of the last y value sent for the left and right sides. If the y value for the last vertex sent for the left side is *less than or equal to* the last y value sent for the right side, the next vertex on the left side should be written to the launch area. Otherwise, the next vertex for the right side should be written to the launch area.

The ROP used for filling polygons can use the pattern and the destination, but not source data. **colorFore** will be used in the ROP in place of source data. Source colorkeying must be turned off, destination colorkeying is allowed.

Pixels that are on the line that forms the left edge of the polygon will be drawn. Pixels that fall on the line that forms the right edge of the polygon will not be drawn. For Horizontal edges, pixels on a horizontal polygon edge that is on the 'top' of the polygon (i.e. above the edge is outside the polygon and below the edge is inside the polygon) will be drawn, while pixels on a horizontal polygon edge that is on the bottom of the polygon will not be drawn.

7.3.8.1 Polygon drawing example

As an example of polygon drawing, say we are drawing the polygon shown in figure 2. Traversing the vertex list in counterclockwise order gives the following list of vertices:

(4, 1) (2, 4) (3, 6) (1, 6) (2,8) (5, 11) (8,8) (13,8) (11,6) (11,3) (10,1)

Figures 2a through 2m show the steps in drawing the polygon. Filled circles are vertices of the left polygon edge. Open circles are vertices of the right polygon edge. Pixels that are drawn at the end of each step are shaded in the figures.

The polygon engine keeps track of four vertices at a time. The top vertex of the current left polygon edge (L0), the bottom vertex of the current left polygon edge (L1), the top vertex of the current right polygon edge (R0), and the bottom vertex of the current right polygon edge (R1). The values of these variables at each step in drawing the polygon are shown in the figures. The arrows in the figures indicate when a variable changes between the start of the step and the end of pixel filling for that step.

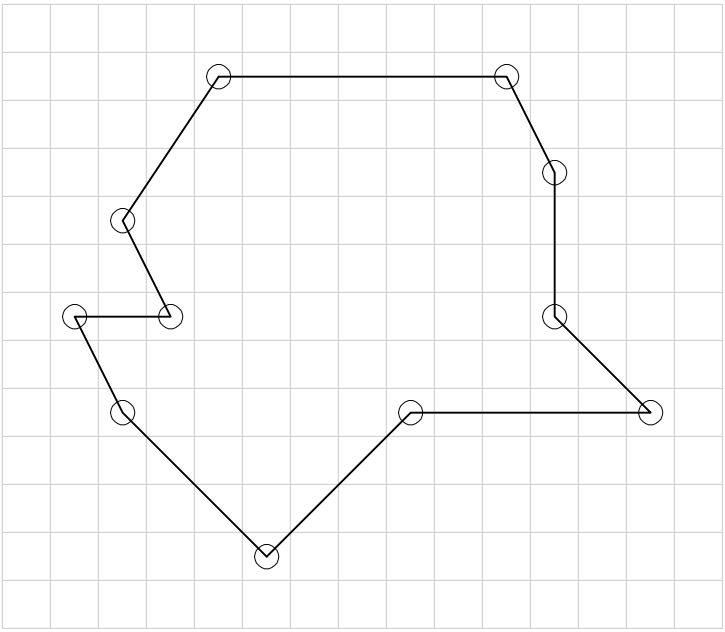


Figure 2

First, all required registers must be written, including the **dstFormat** register to specify the drawing surface, color or pattern registers, and the command register. Write the coordinates of the starting vertex (4, 1) to the **srcXY** register:

srcXY <= 0x00010004

command <= POLYGON_MODE || INITIATE_COMMAND

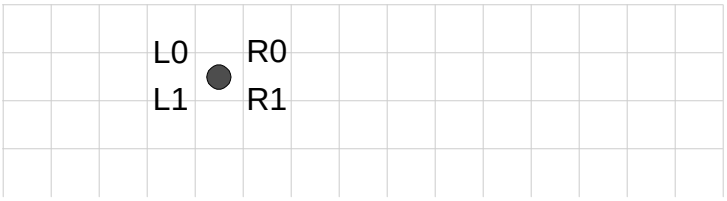


Figure 2a

R1.y>=L1.y, so we have to write the next vertex for the left edge (2, 4):

launch <= 0x00040002

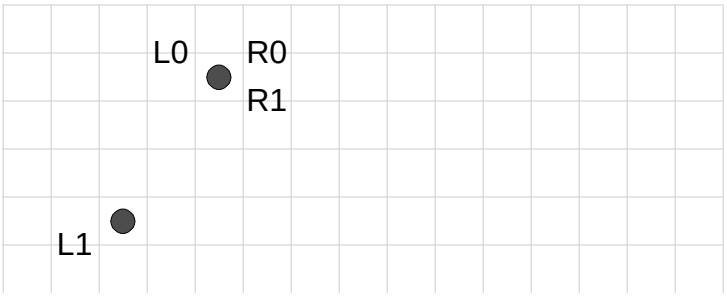


Figure 2b

R1.y<L1.y, so we write the next vertex for the right edge (10, 2). The drawing engine now has edges for both the left and right edges. So, it will draw all spans up to min(R1.y, L1.y). Because R1.y=R0.y, no pixels will be drawn, but R0 will be updated to vertex R1:

launch <= 0x0001000a

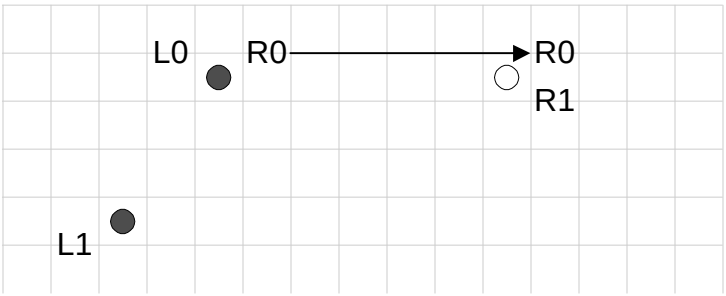


Figure 2c

R1.y<L1.y, so we again write the next vertex on the right polygon edge (11, 3). Pixels on all spans from max(L0.y, R0.y) to min(L1.y, R1.y)-1 will be drawn, as shown below. Because R1.y<L1.y, R0 is updated to R1.

launch <= 0x0003000b

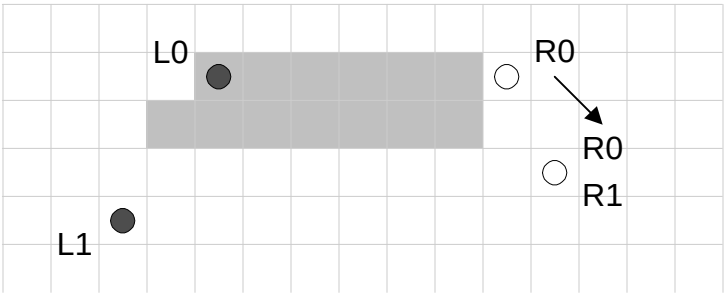


Figure 2d

R1.y<L1.y, so we write the next vertex on the right edge (11, 6). Again, pixels on all spans from max(L0.y, R0.y) to min(L1.y, R1.y)-1 will be drawn. This time R1.y>L1.y, however, so L0 is updated to L1.

launch <= 0x0006000b

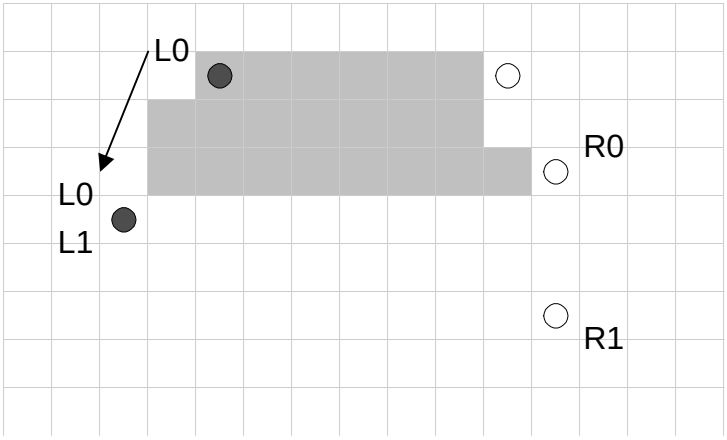


Figure 2e

R1.y>=L1.y, so we write the next vertex on the left edge (3, 6). L1.y=R1.y, so R0 is updated to R1 and L0 is updated to L1.

launch <= 0x00060003

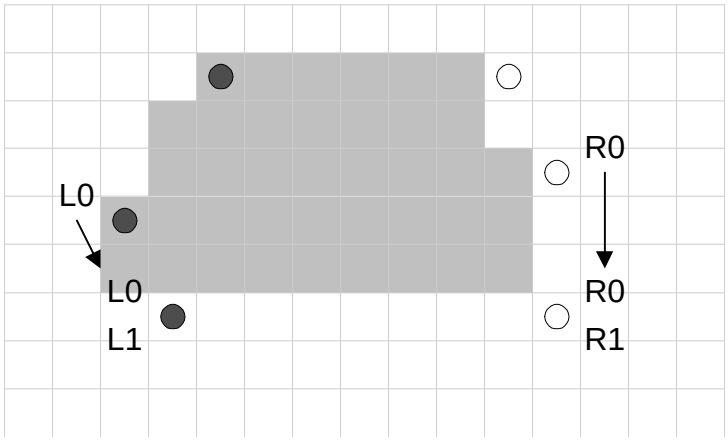


Figure 2f

R1.y>=L1.y, so we write the next vertex on the left edge (1, 6). L1.y=R1.y, so R0 is updated to R1 and L0 is updated to L1. R1 did not change, so updating R0 to R1 has no effect.

launch <= 0x00060001

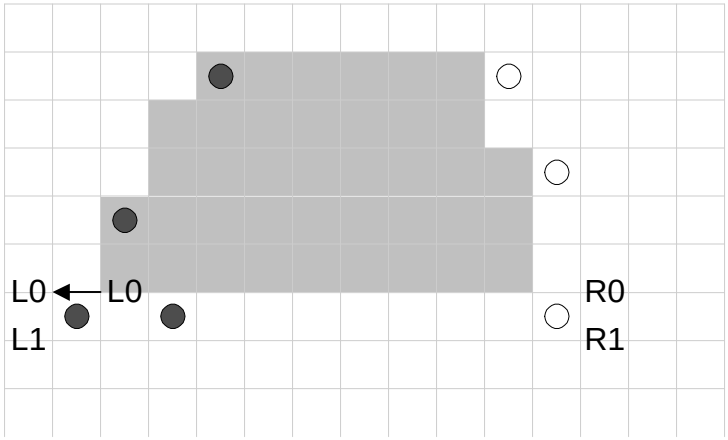


Figure 2g

R1.y>=L1.y, so we again write the next vertex on the left edge (2, 8). L1.y>R1.y, so R0 is updated to R1, again with no effect.

launch <= 0x00080002

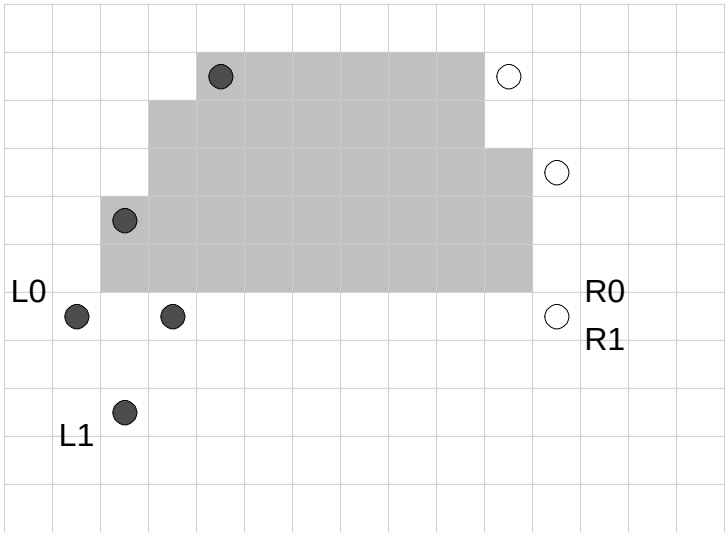


Figure 2h

R1.y<L1.y, so we write the next vertex on the right edge (11, 8). L1.y=R1.y, so R0 is updated to R1, and L0 is updated to L1.

launch <= 0x0008000b

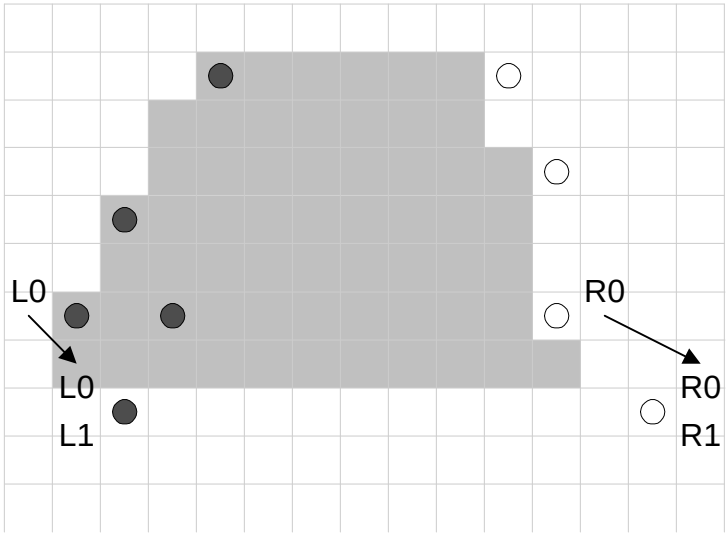


Figure 2i

R1.y>=L1.y, so we write the next vertex on the left edge (5, 11). L1.y>R1.y, so R0 is updated to R1.

launch <= 0x000b0005

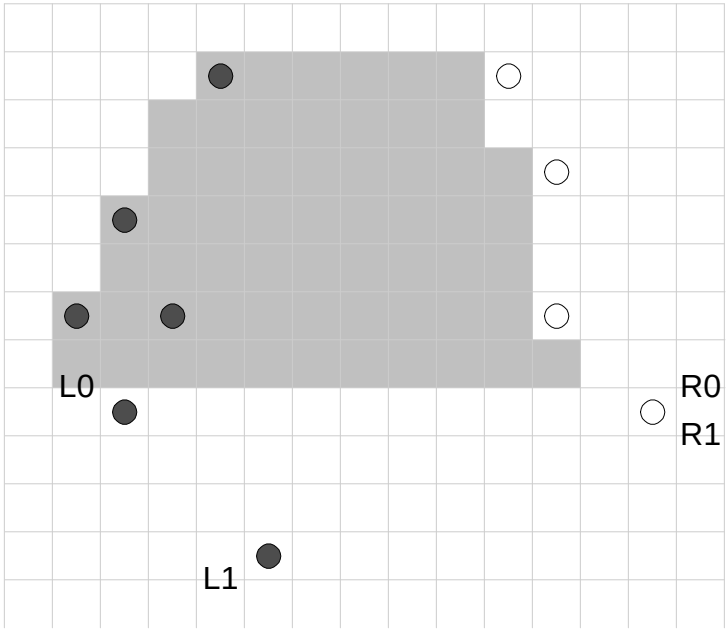


Figure 2j

R1.y<L1.y, so we write the next vertex on the right edge (8, 8). L1.y>R1.y, so R0 is updated to R1, but no pixels are drawn.

launch <= 0x00080008

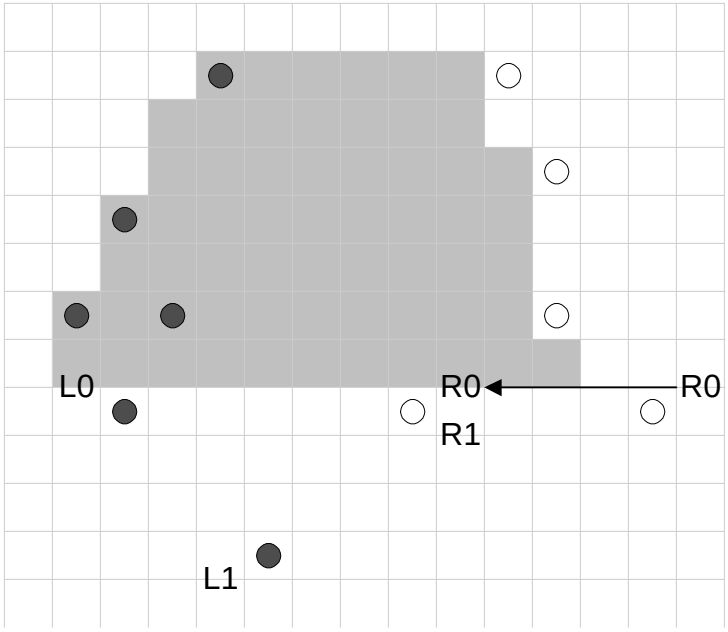


Figure 2k

R1.y<L1.y, so we write the next vertex on the right edge. This is the final vertex in the polygon, which doesn't have a horizontal span at the bottom, so this vertex is the same as the last vertex for the left edge (5, 11). L1.y=R1.y, so R0 is updated to R1, and L0 is updated to L1. No pixels on the final span are drawn (this would be true even if L1.x did not equal R1.x). If the launch area is written again before any registers are written the polygon engine will begin a new polygon starting at (5,11).

launch <= 0x000b0005

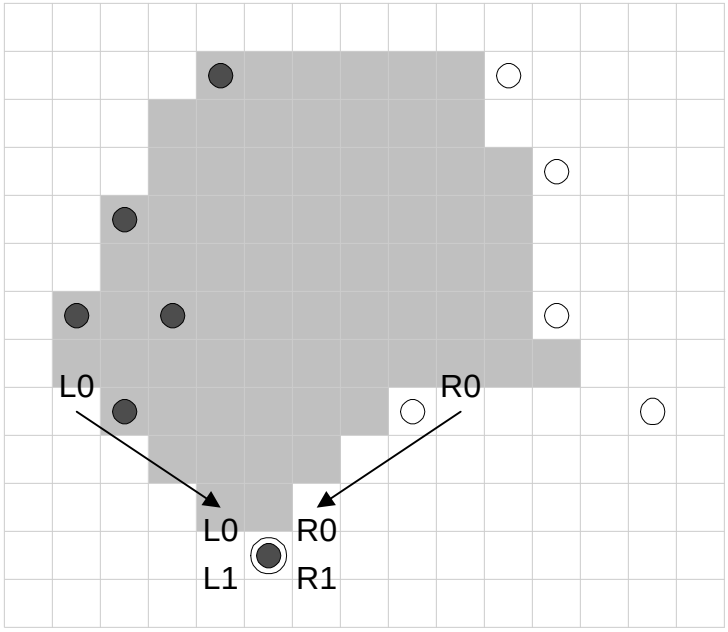


Figure 2m

polygonLaunch

Bit	Description
12:0	X position of a polygon vertex
15:13	RESERVED
28:16	Y position of a polygon vertex
31:29	RESERVED

7.4 Miscellaneous 2D

7.4.1 Write Sgram Mode Register

Executing this command causes the value in srcBaseAddr[10:0] to be set as the sgram mode register via a special bus cycle in the memory controller.

SGRAM mode register

Bit	Description
2:0	burst length
3	burst type (0=sequential, 1=interleave)
6:4	CAS latency
8:7	test mode
9	write burst length (0=burst, 1=single bit).
10	sgram-defined.

7.4.2 Write Sgram Color Register

Executing this command causes the value in srcBaseAddr[31:0] to be set as the sgram color register via a special bus cycle in the memory controller. Since H3 has a 128-bit wide bus, the register is replicated across the four sets of sgram memories.

7.4.3 Write Sgram Mask Register

Executing this command causes the value in srcBaseAddr[31:0] to be set as the sgram mask register via a special bus cycle in the memory controller. Since H3 has a 128-bit wide bus, the register is replicated across the four sets of sgram memories.

8. 3D Memory Mapped Register Set

A 4Mbyte (22-bit) FBI memory mapped register address is divided into the following fields:

AltMap	Swizzle	Wrap	Chip	Register	Byte
1	1	6	4	8	2

The **chip** field selects one or more of the Avenger chips (FBI and/or TREX) to be accessed. Each bit in the **chip** field selects one chip for writing, with FBI controlled by the lsb of the **chip** field, and TREX#2 controlled by the msb of the **chip** field. Note the **chip** field value of 0x0 selects all chips. The following table shows the **chip** field mappings: The current generation of Avenger only supports 1 TREX, so only bits 1:0 are valid, future generation of Avenger will support additional TREX cores.

Chip Field	SST-1 Chip Accessed
0000	FBI + all TREX chips
0001	FBI
0010	TREX #0
0011	FBI + TREX #0
0100	TREX #1
0101	FBI + TREX #1
0110	TREX #0 + TREX #1
0111	FBI + TREX #0 + TREX #1
1000	TREX #2
1001	FBI + TREX #2
1010	TREX #0 + TREX #2
1011	FBI + TREX #0 + TREX #2
1100	TREX #1 + TREX #2
1101	FBI + TREX #1 + TREX #2
1110	TREX #0 + TREX #1 + TREX #2
1111	FBI + all TREX chips

Note that TREX #0 is always connected to FBI in the system level diagrams of section 3, and TREX #1 is attached to TREX #0, etc. By utilizing the different **chip** fields, software can precisely control the data presented to individual chips which compose the Avenger graphics subsystem. Note that for reads, the

chip field is ignored, and read data is always read from FBI. The **register** field selects the register to be accessed from the table below. All accesses to the memory mapped registers must be 32-bit accesses. No byte (8-bit) or halfword (16-bit) accesses are allowed to the memory mapped registers, so the **byte** (2-bit) field of all memory mapped register accesses must be 0x0. As a result, to modify individual bits of a 32-bit register, the entire 32-bit word must be written with valid bits in all positions.

The table below shows the Avenger register set. The register set shown below is the address map when triangle registers address aliasing (remapping) is disabled(**fbiinit3**(0)=0). When The **chip** column illustrates which registers are stored in which chips. For the registers which are stored in TREX, the % symbol specifies that the register is unconditionally written to TREX regardless of the chip address. Similarly, the * symbol specifies that the register is only written to a given TREX if specified in the chip address. The **R/W** column illustrates the read/write status of individual registers. Reading from a register which is "write only" returns undefined data. Also, reading from a register that is TREX specific returns undefined data.. Reads from all other memory mapped registers only contain valid data in the bits stored by the registers, and undefined/reserved bits in a given register must be masked by software. The **sync** column indicates whether the graphics processor must wait for the current command to finish before loading a particular register from the FIFO. A "yes" in the **sync** column means the graphics processor will flush the data pipeline before loading the register -- this will result in a small performance degradation when compared to those registers which do not need synchronization. The **FIFO** column indicates whether a write to a particular register will be pushed into the PCI bus FIFO. Care must be taken when writing to those registers not pushed into the FIFO in order to prevent race conditions between FIFOed and non-FIFOed registers. Also note that reads are not pushed into the PCI bus FIFO, and reading FIFOed registers will return the current value of the register, irrespective of pending writes to the register present in the FIFO.

Memory Base 0: Offset 0x0200000

Register Name	Address	Reg Num	Bits	Chip	R/ W	Sync? /Fifo?	Description
status	0x000(0)	0x0	31:0	FBI	R	No / n/a	Avenger Status
intrCtrl	0x004(4)	0x1	31:0	FBI	R/W	No / No	Interrupt Status and Control
vertexAx	0x008(8)	0x2	15:0	FBI+TREX%	W	No / Yes	Vertex A x-coordinate location (12.4 format)
vertexAy	0x00c(12)	0x3	15:0	FBI+TREX%	W	No / Yes	Vertex A y-coordinate location (12.4 format)
vertexBx	0x010(16)	0x4	15:0	FBI+TREX%	W	No / Yes	Vertex B x-coordinate location (12.4 format)
vertexBy	0x014(20)	0x5	15:0	FBI+TREX%	W	No / Yes	Vertex B y-coordinate location (12.4 format)
vertexCx	0x018(24)	0x6	15:0	FBI+TREX%	W	No / Yes	Vertex C x-coordinate location (12.4 format)
vertexCy	0x01c(28)	0x7	15:0	FBI+TREX%	W	No / Yes	Vertex C y-coordinate location (12.4 format)
startR	0x020(32)	0x8	23:0	FBI	W	No / Yes	Starting Red parameter (12.12 format)
startG	0x024(36)	0x9	23:0	FBI	W	No / Yes	Starting Green parameter (12.12 format)
startB	0x028(40)	0xA	23:0	FBI	W	No / Yes	Starting Blue parameter (12.12 format)
startZ	0x02c(44)	0xB	31:0	FBI	W	No / Yes	Starting Z parameter (20.12 format)
startA	0x030(48)	0xC	23:0	FBI	W	No / Yes	Starting Alpha parameter (12.12 format)
startS	0x034(52)	0xD	31:0	TREX*	W	No / Yes	Starting S/W parameter (14.18 format)
startT	0x038(56)	0xE	31:0	TREX*	W	No / Yes	Starting T/W parameter (14.18 format)
startW	0x03c(60)	0xF	31:0	FBI+TREX*	W	No / Yes	Starting 1/W parameter (2.30 format)
dRdX	0x040(64)	0x10	23:0	FBI	W	No / Yes	Change in Red with respect to X (12.12 format)
dGdX	0x044(68)	0x11	23:0	FBI	W	No / Yes	Change in Green with respect to X (12.12 format)
dBdX	0x048(72)	0x12	23:0	FBI	W	No / Yes	Change in Blue with respect to X (12.12 format)
dZdX	0x04c(76)	0x13	31:0	FBI	W	No / Yes	Change in Z with respect to X (20.12 format)
dAdX	0x050(80)	0x14	23:0	FBI	W	No / Yes	Change in Alpha with respect to X (12.12 format)
dSdX	0x054(84)	0x15	31:0	TREX*	W	No / Yes	Change in S/W with respect to X (14.18 format)
dTdX	0x058(88)	0x16	31:0	TREX*	W	No / Yes	Change in T/W with respect to X (14.18 format)
dWdX	0x05c(92)	0x17	31:0	FBI+TREX*	W	No / Yes	Change in 1/W with respect to X (2.30 format)
dRdY	0x060(96)	0x18	23:0	FBI	W	No / Yes	Change in Red with respect to Y (12.12 format)
dGdY	0x064(100)	0x19	23:0	FBI	W	No / Yes	Change in Green with respect to Y (12.12 format)

dBdY	0x068(104)	0x1A	23:0	FBI	W	No / Yes	Change in Blue with respect to Y (12.12 format)
dZdY	0x06c(108)	0x1B	31:0	FBI	W	No / Yes	Change in Z with respect to Y (20.12 format)
dAdY	0x070(112)	0x1C	23:0	FBI	W	No / Yes	Change in Alpha with respect to Y (12.12 format)
dSdY	0x074(116)	0x1D	31:0	TREX*	W	No / Yes	Change in S/W with respect to Y (14.18 format)
dTdY	0x078(120)	0x1E	31:0	TREX*	W	No / Yes	Change in T/W with respect to Y (14.18 format)
dWdY	0x07c(124)	0x1F	31:0	FBI+TREX*	W	No / Yes	Change in 1/W with respect to Y (2.30 format)
triangleCMD	0x080(128)	0x20	31	FBI+TREX%	W	No / Yes	Execute TRIANGLE command (floating point)
reserved	0x084(132)	0x21	n/a	n/a	W	n/a	
fvertexAx	0x088(136)	0x22	31:0	FBI+TREX%	W	No / Yes	Vertex A x-coordinate location (floating point)
fvertexAy	0x08c(140)	0x23	31:0	FBI+TREX%	W	No / Yes	Vertex A y-coordinate location (floating point)
fvertexBx	0x090(144)	0x24	31:0	FBI+TREX%	W	No / Yes	Vertex B x-coordinate location (floating point)
fvertexBy	0x094(148)	0x25	31:0	FBI+TREX%	W	No / Yes	Vertex B y-coordinate location (floating point)
fvertexCx	0x098(152)	0x26	31:0	FBI+TREX%	W	No / Yes	Vertex C x-coordinate location (floating point)
fvertexCy	0x09c(156)	0x27	31:0	FBI+TREX%	W	No / Yes	Vertex C y-coordinate location (floating point)
fstartR	0x0a0(160)	0x28	31:0	FBI	W	No / Yes	Starting Red parameter (floating point)
fstartG	0x0a4(164)	0x29	31:0	FBI	W	No / Yes	Starting Green parameter (floating point)
fstartB	0x0a8(168)	0x2A	31:0	FBI	W	No / Yes	Starting Blue parameter (floating point)
fstartZ	0x0ac(172)	0x2B	31:0	FBI	W	No / Yes	Starting Z parameter (floating point)
fstartA	0x0b0(176)	0x2C	31:0	FBI	W	No / Yes	Starting Alpha parameter (floating point)
fstartS	0x0b4(180)	0x2D	31:0	TREX*	W	No / Yes	Starting S/W parameter (floating point)
fstartT	0x0b8(184)	0x2E	31:0	TREX*	W	No / Yes	Starting T/W parameter (floating point)
fstartW	0x0bc(188)	0x2F	31:0	FBI+TREX*	W	No / Yes	Starting 1/W parameter (floating point)
fdRdX	0x0c0(192)	0x30	31:0	FBI	W	No / Yes	Change in Red with respect to X (floating point)
fdGdX	0x0c4(196)	0x31	31:0	FBI	W	No / Yes	Change in Green with respect to X (floating point)
fdBdX	0x0c8(200)	0x32	31:0	FBI	W	No / Yes	Change in Blue with respect to X (floating point)
fdZdX	0x0cc(204)	0x33	31:0	FBI	W	No / Yes	Change in Z with respect to X (floating point)
fdAdX	0x0d0(208)	0x34	31:0	FBI	W	No / Yes	Change in Alpha with respect to X (floating point)
fdSdX	0x0d4(212)	0x35	31:0	TREX*	W	No / Yes	Change in S/W with respect to X (floating point)
fdTdX	0x0d8(216)	0x36	31:0	TREX*	W	No / Yes	Change in T/W with respect to X (floating point)
fdWdX	0x0dc(220)	0x37	31:0	FBI+TREX*	W	No / Yes	Change in 1/W with respect to X (floating point)
fdRdY	0x0e0(224)	0x38	31:0	FBI	W	No / Yes	Change in Red with respect to Y (floating point)
fdGdY	0x0e4(228)	0x39	31:0	FBI	W	No / Yes	Change in Green with respect to Y (floating point)
fdBdY	0x0e8(232)	0x3A	31:0	FBI	W	No / Yes	Change in Blue with respect to Y (floating point)
fdZdY	0x0ec(236)	0x3B	31:0	FBI	W	No / Yes	Change in Z with respect to Y (floating point)
fdAdY	0x0f0(240)	0x3C	31:0	FBI	W	No / Yes	Change in Alpha with respect to Y (floating point)
fdSdY	0x0f4(244)	0x3D	31:0	TREX*	W	No / Yes	Change in S/W with respect to Y (floating point)
fdTdY	0x0f8(248)	0x3E	31:0	TREX*	W	No / Yes	Change in T/W with respect to Y (floating point)
fdWdY	0x0fc(252)	0x3F	31:0	FBI+TREX*	W	No / Yes	Change in 1/W with respect to Y (floating point)
ftriangleCMD	0x100(256)	0x40	31	FBI+TREX%	W	No / Yes	Execute TRIANGLE command (floating point)
fbzColorPath	0x104(260)	0x41	27:0	FBI+TREX%	R/W	No / Yes	FBI Color Path Control
fogMode	0x108(264)	0x42	5:0	FBI	R/W	No / Yes	Fog Mode Control
alphaMode	0x10c(268)	0x43	31:0	FBI	R/W	No / Yes	Alpha Mode Control
fbzMode	0x110(272)	0x44	20:0	FBI	R/W	Yes / Yes	RGB Buffer and Depth-Buffer Control
lfbMode	0x114(276)	0x45	16:0	FBI	R/W	Yes / Yes	Linear Frame Buffer Mode Control
clipLeftRight	0x118(280)	0x46	31:0	FBI	R/W	Yes / Yes	Left and Right of Clipping Register
clipTopBottom	0x11c(284)	0x47	31:0	FBI	R/W	Yes / Yes	Top and Bottom of Clipping Register
nopCMD	0x120(288)	0x48	0	FBI+TREX%	W	Yes/Yes	Execute NOP command
fastfillCMD	0x124(292)	0x49	na	FBI	W	Yes/Yes	Execute FASTFILL command
swapbufferCMD	0x128(296)	0x4A	8:0	FBI	W	Yes/Yes	Execute SWAPBUFFER command
fogColor	0x12c(300)	0x4B	23:0	FBI	W	Yes / Yes	Fog Color Value
zaColor	0x130(304)	0x4C	31:0	FBI	W	Yes / Yes	Constant Alpha/Depth Value
chromaKey	0x134(308)	0x4D	23:0	FBI	W	Yes / Yes	Chroma Key Compare Value
chromaRange	0x138(312)	0x4E	27:0	FBI	W	Yes / Yes	Chroma Range Compare Values, Modes & Enable
userIntrCMD	0x13c(316)	0x4F	9:0	FBI	W	Yes / Yes	Execute USERINTERRUPT command
stipple	0x140(320)	0x50	31:0	FBI	R/W	Yes / Yes	Rendering Stipple Value
color0	0x144(324)	0x51	31:0	FBI	R/W	Yes / Yes	Constant Color #0
color1	0x148(328)	0x52	31:0	FBI	R/W	Yes / Yes	Constant Color #1

fbiPixelsIn	0x14c(332)	0x53	23:0	FBI	R	n/a	Pixel Counter (Number pixels processed)
fbiChromaFail	0x150(336)	0x54	23:0	FBI	R	n/a	Pixel Counter (Number pixels failed Chroma test)
fbiZfuncFail	0x154(340)	0x55	23:0	FBI	R	n/a	Pixel Counter (Number pixels failed Z test)
fbiAfuncFail	0x158(344)	0x56	23:0	FBI	R	n/a	Pixel Counter (Number pixels failed Alpha test)
fbiPixelsOut	0x15c(348)	0x57	23:0	FBI	R	n/a	Pixel Counter (Number pixels drawn)
fogTable	0x160(352) to 0x1dc(476)	0x58 to 0x77	31:0	FBI	W	Yes / Yes	Fog Table
reserved	0x1e0(480) to 0x1e8(488)	0x78 to 0x7A	na				
colBufferAddr	0x1ec(492)	0x7B	23:0	FBI	R/W	Yes / Yes	Color Buffer Base Address
colBufferStride	0x1f0(496)	0x7C	15:0	FBI	R/W	Yes / Yes	Color Buffer Stride, Memory type
auxBufferAddr	0x1f4(500)	0x7D	23:0	FBI	R/W	Yes / Yes	Auxiliary Buffer Base Address
auxBufferStride	0x1f8(504)	0x7E	15:0	FBI	R/W	Yes / Yes	Auxiliary Buffer Stride, Memory type
reserved	0x1fc(508)	0x7F	n/a	n/a	n/a	n/a	
clipLeftRight1	0x200(512)	0x80	31:0	FBI	R/W	Yes / Yes	Secondary Left/Right Clipping Register
clipTopBottom1	0x204(516)	0x81	31:0	FBI	R/W	Yes / Yes	Secondary Top/Bottom Clipping Register
reserved	0x208(520) to 0x24b(587)	0x82 to 0x92	n/a	n/a	n/a	n/a	
swapPending	0x24c(588)	0x93	na	FBI	W	No / No	Swap buffer pending
leftOverlayBuf	0x250(592)	0x94	31:0	FBI	W	No / Yes	Left Overlay address
rightOverlayBuf	0x254(596)	0x95	31:0	FBI	W	No / Yes	Right Overlay address
fbiSwapHistory	0x258(600)	0x96	31:0	FBI	R	n/a	Swap History Register
fbiTrianglesOut	0x25c(604)	0x97	23:0	FBI	R	n/a	Triangle Counter (Number triangles drawn)
sSetupMode	0x260(608)	0x98	19:0	FBI	W	No / Yes	Triangle setup mode
sVx	0x264(612)	0x99	31:0	FBI+TMU*	W	No / Yes	Triangle setup X
sVy	0x268(616)	0x9A	31:0	FBI+TMU*	W	No / Yes	Triangle setup Y
sARGB	0x26c(620)	0x9B	31:0	FBI+TMU*	W	No / Yes	Triangle setup Alpha, Red, Green, Blue
sRed	0x270(624)	0x9C	31:0	FBI	W	No / Yes	Triangle setup Red value
sGreen	0x274(628)	0x9D	31:0	FBI	W	No / Yes	Triangle setup Green value
sBlue	0x278(632)	0x9E	31:0	FBI	W	No / Yes	Triangle setup Blue value
sAlpha	0x27c(636)	0x9F	31:0	FBI	W	No / Yes	Triangle setup Alpha value
sVz	0x280(640)	0xA0	31:0	FBI	W	No / Yes	Triangle setup Z
sWb	0x284(644)	0xA1	31:0	FBI+TMU*	W	No / Yes	Triangle setup Global W
sWtmu0	0x288(648)	0xA2	31:0	TMU*	W	No / Yes	Triangle setup Tmu0 & Tmu1 W
sS/W0	0x28c(652)	0xA3	31:0	TMU*	W	No / Yes	Triangle setup Tmu0 & Tmu1 S/W
sT/W0	0x290(656)	0xA4	31:0	TMU*	W	No / Yes	Triangle setup Tmu0 & Tmu1 T/W
sWtmu1	0x294(660)	0xA5	31:0	TMU1	W	No / Yes	Triangle setup Tmu1 only W
sS/Wtmu1	0x298(664)	0xA6	31:0	TMU1	W	No / Yes	Triangle setup Tmu1 only S/W
sT/Wtmu1	0x29c(668)	0xA7	31:0	TMU1	W	No / Yes	Triangle setup Tmu1 only T/W
sDrawTriCMD	0x2a0(672)	0xA8	31:0	FBI+TMU*	W	No / Yes	Triangle setup (Draw)
sBeginTriCMD	0x2a4(676)	0xA9	31:0	FBI	W	No / Yes	Triangle setup Start New triangle
reserved	0x2a8(680) to 0x2fc(764)	0xAA to 0xBF	n/a	n/a	n/a	n/a	
textureMode	0x300(768)	0xC0	30:0	TREX*	W	No / Yes	Texture Mode Control
tLOD	0x304(772)	0xC1	23:0	TREX*	W	No / Yes	Texture LOD Settings
tDetail	0x308(776)	0xC2	21:0	TREX*	W	No / Yes	Texture Detail Settings
texBaseAddr	0x30c(780)	0xC3	31:0	TREX*	W	No / Yes	Texture Base Address
texBaseAddr_1	0x310(784)	0xC4	23:0	TREX*	W	No / Yes	Texture Base Address (supplemental LOD 1)
texBaseAddr_2	0x314(788)	0xC5	23:0	TREX*	W	No / Yes	Texture Base Address (supplemental LOD 2)
texBaseAddr_3_8	0x318(792)	0xC6	23:0	TREX*	W	No / Yes	Texture Base Address (supplemental LOD 3-8)
texStride	0x31c(796)	0xC7					
trexInit1	0x320(800)	0xC8	31:0	TREX*	W	Yes / Yes	TREX Hardware Initialization (register 1

nccTable0	0x324(804) to 0x350(848)	0xC9 to 0xD4	31:0 or 26:0	TREX*	W	Yes / Yes	Narrow Channel Compression Table 0 (12 entries)
nccTable1	0x354(852) to 0x380(896)	0xD5 tp 0xE0	31:0 or 26:0	TREX*	W	Yes / Yes	Narrow Channel Compression Table 1 (12 entries)
reserved	0x384(900) to 0x3fc(1020)	0xE1 to 0xFF	n/a	n/a	n/a	n/a	

The triangle parameter registers are aliased to a different address mapping to improve PCI bus throughput. The upper bit of the **wrap** field in the pci address is 0x1 (**pci_ad**[21]=1), the following table shows the addresses for the triangle parameter registers.

Register Name	Address	Reg Num	Bits	Chip	R/ W	Sync? /Fifo?	Description
status	0x000(0)	0x0	31:0	FBI	R/W	No / Yes	SST-1 Status
intrCtrl	0x004(4)	0x1	19:0	FBI	R/W	No / No	Interrupt Status and Control
vertexAx	0x008(8)	0x2	15:0	FBI+TREX%	W	No / Yes	Vertex A x-coordinate location (12.4 format)
vertexAy	0x00c(12)	0x3	15:0	FBI+TREX%	W	No / Yes	Vertex A y-coordinate location (12.4 format)
vertexBx	0x010(16)	0x4	15:0	FBI+TREX%	W	No / Yes	Vertex B x-coordinate location (12.4 format)
vertexBy	0x014(20)	0x5	15:0	FBI+TREX%	W	No / Yes	Vertex B y-coordinate location (12.4 format)
vertexCx	0x018(24)	0x6	15:0	FBI+TREX%	W	No / Yes	Vertex C x-coordinate location (12.4 format)
vertexCy	0x01c(28)	0x7	15:0	FBI+TREX%	W	No / Yes	Vertex C y-coordinate location (12.4 format)
startR	0x020(32)	0x8	23:0	FBI	W	No / Yes	Starting Red parameter (12.12 format)
dRdX	0x024(36)	0x9	23:0	FBI	W	No / Yes	Change in Red with respect to X (12.12 format)
dRdY	0x028(40)	0xA	23:0	FBI	W	No / Yes	Change in Red with respect to Y (12.12 format)
startG	0x02c(44)	0xB	23:0	FBI	W	No / Yes	Starting Green parameter (12.12 format)
dGdX	0x030(48)	0xC	23:0	FBI	W	No / Yes	Change in Green with respect to X (12.12 format)
dGdY	0x034(52)	0xD	23:0	FBI	W	No / Yes	Change in Green with respect to Y (12.12 format)
startB	0x038(56)	0xE	23:0	FBI	W	No / Yes	Starting Blue parameter (12.12 format)
dBdX	0x03c(60)	0xF	23:0	FBI	W	No / Yes	Change in Blue with respect to X (12.12 format)
dBdY	0x040(64)	0x10	23:0	FBI	W	No / Yes	Change in Blue with respect to Y (12.12 format)
startZ	0x044(68)	0x11	31:0	FBI	W	No / Yes	Starting Z parameter (20.12 format)
dZdX	0x048(72)	0x12	31:0	FBI	W	No / Yes	Change in Z with respect to X (20.12 format)
dZdY	0x04c(76)	0x13	31:0	FBI	W	No / Yes	Change in Z with respect to Y (12.12 format)
startA	0x050(80)	0x14	23:0	FBI	W	No / Yes	Starting Alpha parameter (12.12 format)
dAdX	0x054(84)	0x15	23:0	FBI	W	No / Yes	Change in Alpha with respect to X (12.12 format)
dAdY	0x058(88)	0x16	23:0	FBI	W	No / Yes	Change in Alpha with respect to Y (12.12 format)
startS	0x05c(92)	0x17	31:0	TREX*	W	No / Yes	Starting S/W parameter (14.18 format)
dSdX	0x060(96)	0x18	31:0	TREX*	W	No / Yes	Change in S/W with respect to X (14.18 format)
dSdY	0x064(100)	0x19	31:0	TREX*	W	No / Yes	Change in S/W with respect to Y (14.18 format)
startT	0x068(104)	0x1A	31:0	TREX*	W	No / Yes	Starting T/W parameter (14.18 format)
dTdX	0x06c(108)	0x1B	31:0	TREX*	W	No / Yes	Change in T/W with respect to X (14.18 format)
dTdY	0x070(112)	0x1C	31:0	TREX*	W	No / Yes	Change in T/W with respect to Y (14.18 format)
startW	0x074(116)	0x1D	31:0	FBI+TREX*	W	No / Yes	Starting 1/W parameter (2.30 format)
dWdX	0x078(120)	0x1E	31:0	FBI+TREX*	W	No / Yes	Change in 1/W with respect to X (2.30 format)
dWdY	0x07c(124)	0x1F	31:0	FBI+TREX*	W	No / Yes	Change in 1/W with respect to Y (2.30 format)
triangleCMD	0x080(128)	0x20	31	FBI+TREX%	W	No / Yes	Execute TRIANGLE command (sign bit)
reserved	0x084(132)	0x21	n/a	n/a	W	n/a	
fvertexAx	0x088(136)	0x22	31:0	FBI+TREX%	W	No / Yes	Vertex A x-coordinate location (floating point)
fvertexAy	0x08c(140)	0x23	31:0	FBI+TREX%	W	No / Yes	Vertex A y-coordinate location (floating point)
fvertexBx	0x090(144)	0x24	31:0	FBI+TREX%	W	No / Yes	Vertex B x-coordinate location (floating point)
fvertexBy	0x094(148)	0x25	31:0	FBI+TREX%	W	No / Yes	Vertex B y-coordinate location (floating point)
fvertexCx	0x098(152)	0x26	31:0	FBI+TREX%	W	No / Yes	Vertex C x-coordinate location (floating point)
fvertexCy	0x09c(156)	0x27	31:0	FBI+TREX%	W	No / Yes	Vertex C y-coordinate location (floating point)

fstartR	0x0a0(160)	0x28	31:0	FBI	W	No / Yes	Starting Red parameter (floating point)
fdRdX	0x0a4(164)	0x29	31:0	FBI	W	No / Yes	Change in Red with respect to X (floating point)
fdRdY	0x0a8(168)	0x2A	31:0	FBI	W	No / Yes	Change in Red with respect to Y (floating point)
fstartG	0x0ac(172)	0x2B	31:0	FBI	W	No / Yes	Starting Green parameter (floating point)
fdGdX	0x0b0(176)	0x2C	31:0	FBI	W	No / Yes	Change in Green with respect to X (floating point)
fdGdY	0x0b4(180)	0x2D	31:0	FBI	W	No / Yes	Change in Green with respect to Y (floating point)
fstartB	0x0b8(184)	0x2E	31:0	FBI	W	No / Yes	Starting Blue parameter (floating point)
fdBdX	0x0bc(188)	0x2F	31:0	FBI	W	No / Yes	Change in Blue with respect to X (floating point)
fdBdY	0x0c0(192)	0x30	31:0	FBI	W	No / Yes	Change in Blue with respect to Y (floating point)
fstartZ	0x0c4(196)	0x31	31:0	FBI	W	No / Yes	Starting Z parameter (floating point)
fdZdX	0x0c8(200)	0x32	31:0	FBI	W	No / Yes	Change in Z with respect to X (floating point)
fdZdY	0x0cc(204)	0x33	31:0	FBI	W	No / Yes	Change in Z with respect to Y (floating point)
fstartA	0x0d0(208)	0x34	31:0	FBI	W	No / Yes	Starting Alpha parameter (floating point)
fdAdX	0x0d4(212)	0x35	31:0	FBI	W	No / Yes	Change in Alpha with respect to X (floating point)
fdAdY	0x0d8(216)	0x36	31:0	FBI	W	No / Yes	Change in Alpha with respect to Y (floating point)
fstartS	0x0dc(220)	0x37	31:0	TREX*	W	No / Yes	Starting S/W parameter (floating point)
fdSdX	0x0e0(224)	0x38	31:0	TREX*	W	No / Yes	Change in S/W with respect to X (floating point)
fdSdY	0x0e4(228)	0x39	31:0	TREX*	W	No / Yes	Change in S/W with respect to Y (floating point)
fstartT	0x0e8(232)	0x3A	31:0	TREX*	W	No / Yes	Starting T/W parameter (floating point)
fdTdX	0x0ec(236)	0x3B	31:0	TREX*	W	No / Yes	Change in T/W with respect to X (floating point)
fdTdY	0x0f0(240)	0x3C	31:0	TREX*	W	No / Yes	Change in T/W with respect to Y (floating point)
fstartW	0x0f4(244)	0x3D	31:0	FBI+TREX*	W	No / Yes	Starting 1/W parameter (floating point)
fdWdX	0x0f8(248)	0x3E	31:0	FBI+TREX*	W	No / Yes	Change in 1/W with respect to X (floating point)
fdWdY	0x0fc(252)	0x3F	31:0	FBI+TREX*	W	No / Yes	Change in 1/W with respect to Y (floating point)
ftriangleCMD	0x100(256)	0x40	31	FBI+TREX%	W	No / Yes	Execute TRIANGLE command (floating point)

8.1 status Register

The **status** register provides a way for the CPU to interrogate the graphics processor about its current state and FIFO availability. The **status** register is read only, but writing to **status** clears any Avenger generated PCI interrupts.

Bit	Description
5:0	PCI FIFO freespace (0x3f=FIFO empty). Default is 0x3f.
6	Vertical retrace (0=Vertical retrace active, 1=Vertical retrace inactive). Default is 1.
7	FBI graphics engine busy (0=engine idle, 1=engine busy). Default is 0.
8	TREX busy (0=engine idle, 1=engine busy). Default is 0.
9	Avenger busy (0=idle, 1=busy). Default is 0.
10	2D busy (0=idle, 1=busy). Default is 0.
11	Reserved
27:12	reserved
30:28	Swap Buffers Pending. Default is 0x0.
31	PCI Interrupt Generated. Default is 0x0. (not currently implemented).

Bits(5:0) show the number of entries available in the internal host FIFO. The internal host FIFO is 64 entries deep. The FIFO is empty when bits(5:0)=0x3f. Bit(6) is the state of the monitor vertical retrace signal, and is used to determine when the monitor is being refreshed. Bit(7) of **status** is used to determine if the graphics engine of FBI is active. Note that bit(7) only determines if the graphics engine of FBI is busy -- it does not include information as to the status of the internal PCI FIFOs. Bit(8) of **status** is used to determine if TREX is busy. Note that bit(8) of **status** is set if any unit in TREX is not idle -- this includes the graphics engine and all internal TREX FIFOs. Bit(9) of **status** determines if all units in the Avenger system (including graphics engines, FIFOs, etc.) are idle. Bit(9) is set when any internal unit in Avenger is active (e.g. graphics is being rendered or any FIFO is not empty). When the Memory FIFO is enabled, bits(27:12) show the number of entries available in the Memory FIFO. Depending upon the amount of frame buffer memory available, a maximum of 65,536 entries may be stored in the Memory FIFO. The Memory FIFO is empty when bits(27:12)=0xffff. Bits (30:28) of **status** track the number of outstanding SWAPBUFFER commands. When a SWAPBUFFER command is received from the host cpu, bits (30:28) are incremented -- when a SWAPBUFFER command completes, bits (30:28) are decremented. Bit(31) of status is used to monitor the status of the PCI interrupt signal. If Avenger generates a vertical retrace interrupt (as defined in **pciInterrupt**), bit(31) is set and the PCI interrupt signal line is activated to generate a hardware interrupt. An interrupt is cleared by writing to **status** with "dont-care" data. NOTE THAT BIT(31) IS CURRENTLY NOT IMPLEMENTED IN HARDWARE, AND WILL ALWAYS RETURN 0X0.

8.2 intrCtrl Register

The **intrCtrl** register controls the interrupt capabilities of Avenger. Bits 1:0 enable video horizontal sync signal generation of interrupts. Generated horizontal sync interrupts are detected by the CPU by reading bits 7:6 of **intrCtrl**. Bits 3:2 enable video vertical sync signal generation of interrupts. Generated vertical sync interrupts are detected by the CPU by reading bits 9:8 of **intrCtrl**. Bit 4 of **intrCtrl** enables generation of interrupts when the frontend PCI FIFO is full. Generated PCI FIFO Full interrupts are detected by the CPU by reading bit 10 of **intrCtrl**. PCI FIFO full interrupts are genered when **intrCtrl** bit 4 is set and the number of free entries in the frontend PCI FIFO drops below the value specified in **fbiInit0** bits(10:6). Bit 5 of **intrCtrl** enables the user interrupt command USERINTERRUPT generation of interrupts. Generated user interrupts are detected by the CPU by reading bit 11 of **intrCtrl**. The tag associated with a generated user interrupt is stored in bits 19:12 of **intrCtrl**.

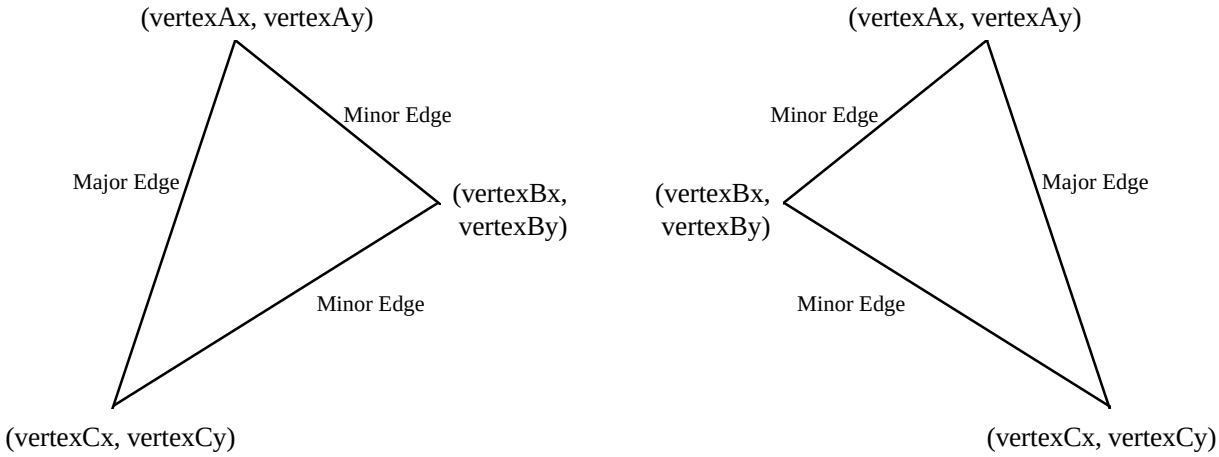
Generated interrupts are cleared by writing a 0 to the bit signaling a particular interrupt was generated and writing a 1 to **interCtrl** bit(31). For example, a PCI FIFO full generated interrupt is cleared by writing a 0 to bit 10 of **intrCtrl**, and a generated user interrupt is cleared by writing a 0 to bit 11 of **intrCtrl**. For both cases, bit 31 of **intrCtrl** must be written with the value 1 to clear the external PCI interrupt. Care must be taken when clearing interrupts not to accidentally overwrite the interrupt mask bits (bits 5:0) of **intrCtrl**) which enable generation of particular interrupts.

Note that writes to the **intrCtrl** register are not pushed on the PCI frontend FIFO, so writes to **intrCtrl** are processed immediately. Since **intrCtrl** is not FIFO'ed, writes to **intrCtrl** may be processed out-of-order with respect to other queued writes in the PCI and memory-backed FIFOs.

Bit	Description
0	Horizontal Sync (rising edge) interrupts enable (1=enable). Default is 0.
1	Horizontal Sync (falling edge) interrupts enable (1=enable). Default is 0.
2	Vertical Sync (rising edge) interrupts enable (1=enable). Default is 0.
3	Vertical Sync (falling edge) interrupts enable (1=enable). Default is 0.
4	PCI FIFO Full interrupts enable (1=enable). Default is 0.
5	User Interrupt Command interrupts enable (1=enable). Default is 0.
6	Horizontal Sync (rising edge) interrupt generated (1=interrupt generated).
7	Horizontal Sync (falling edge) interrupt generated (1=interrupt generated).
8	Vertical Sync (rising edge) interrupt generated (1=interrupt generated).
9	Vertical Sync (falling edge) interrupt generated (1=interrupt generated).
10	PCI FIFO Full interrupt generated (1=interrupt generated).
11	User Interrupt Command interrupt generated (1=interrupt generated).
19:12	User Interrupt Command Tag. Read only.
20	Hole counting interupts enable (1=enable). Default is 0.
21	VMI interrupts enable. (1=enable). Default is 0.
22	Hole counting interrupt generated (1=interrupt generated).
23	VMI interrupt generated (1=interrupt generated).
29:24	reserved
30	VGA Interrupt generated (1=interrupt generated).
31	External pin pci_inta value, active low (0=PCI interrupt is active, 1=PCI interrupt is inactive)

8.3 vertex and fvertex Registers

The **vertexAx**, **vertexAy**, **vertexBx**, **vertexBy**, **vertexCx**, **vertexCy**, **fvertexAx**, **fvertexAy**, **fvertexBx**, **fvertexBy**, **fvertexCx**, and **fvertexCy** registers specify the x and y coordinates of a triangle to be rendered. There are three vertices in an Avenger triangle, with the **AB** and **BC** edges defining the minor edge and the **AC** edge defining the major edge. The diagram below illustrates two typical triangles:



The **fvertex** registers are floating point equivalents of the **vertex** registers. Avenger automatically converts both the **fvertex** and **vertex** registers into an internal fixed point notation used for rendering.

vertexAx, **vertexAy**, **vertexBx**, **vertexBy**, **vertexCx**, **vertexCy**

Bit	Description
15:0	Vertex coordinate information (fixed point two's complement 12.4 format)

fvertexAx, **fvertexAy**, **fvertexBx**, **fvertexBy**, **fvertexCx**, **fvertexCy**

Bit	Description
31:0	Vertex coordinate information (IEEE 32-bit single-precision floating point format)

8.4 startR, startG, startB, startA, fstartR, fstartG, fstartB, and fstartA Registers

The **startR**, **startG**, **startB**, **startA**, **fstartR**, **fstartG**, **fstartB**, and **fstartA** registers specify the starting color information (red, green, blue, and alpha) of a triangle to be rendered. The **start** registers must contain the color values associated with the **A** vertex of the triangle. The **fstart** registers are floating point equivalents of the **start** registers. Avenger automatically converts both the **start** and **fstart** registers into an internal fixed point notation used for rendering.

startR, startG, startB, startA

Bit	Description
23:0	Starting Vertex-A Color information (fixed point two's complement 12.12 format)

fstartR, fstartG, fstartB, fstartA

Bit	Description
31:0	Starting Vertex-A Color information (IEEE 32-bit single-precision floating point format)

8.5 startZ and fstartZ registers

The **startZ** and **fstartZ** registers specify the starting Z information of a triangle to be rendered. The **startZ** registers must contain the Z values associated with the **A** vertex of the triangle. The **fstartZ** register is a floating point equivalent of the **startZ** registers. Avenger automatically converts both the **startZ** and **fstartZ** registers into an internal fixed point notation used for rendering.

startZ

Bit	Description

31:0	Starting Vertex-A Z information (fixed point two's complement 20.12 format)

fstartZ

Bit	Description
31:0	Starting Vertex-A Z information (IEEE 32-bit single-precision floating point format)

8.6 startS, startT, fstartS, and fstartT Registers

The **startS**, **startT**, **fstartS**, and **fstartT** registers specify the starting S/W and T/W texture coordinate information of a triangle to be rendered. The **start** registers must contain the texture coordinates associated with the **A** vertex of the triangle. Note that the S and T coordinates used by Avenger for rendering must be divided by W prior to being sent to Avenger (i.e. Avenger iterates S/W and T/W prior to perspective correction). During rendering, the iterated **S** and **T** coordinates are (optionally) divided by the iterated **W** parameter to perform perspective correction. The **fstart** registers are floating point equivalents of the **start** registers. Avenger automatically converts both the **start** and **fstart** registers into an internal fixed point notation used for rendering.

startS, startT

Bit	Description
31:0	Starting Vertex-A Texture coordinates (fixed point two's complement 14.18 format)

fstartS, fstartT

Bit	Description
31:0	Starting Vertex-A Texture coordinates (IEEE 32-bit single-precision floating point format)

8.7 startW and fstartW registers

The **startW** and **fstartW** registers specify the starting 1/W information of a triangle to be rendered. The **startW** registers must contain the W values associated with the **A** vertex of the triangle. Note that the **W** value used by Avenger for rendering is actually the reciprocal of the 3D-geometry-calculated W value (i.e. Avenger iterates 1/W prior to perspective correction). During rendering, the iterated **S** and **T** coordinates are (optionally) divided by the iterated **W** parameter to perform perspective correction. The **fstartW** register is a floating point equivalent of the **startW** registers. Avenger automatically converts both the **startW** and **fstartW** registers into an internal fixed point notation used for rendering.

startW

Bit	Description
31:0	Starting Vertex-A W information (fixed point two's complement 2.30 format)

fstartW

Bit	Description
31:0	Starting Vertex-A W information (IEEE 32-bit single-precision floating point format)

8.8 dRdX, dGdX, dBdX, dAdX, fdRdX, fdGdX, fdBdX, and fdAdX Registers

The **dRdX**, **dGdX**, **dBdX**, **dAdX**, **fdRdX**, **fdGdX**, **fdBdX**, and **fdAdX** registers specify the change in the color information (red, green, blue, and alpha) with respect to X of a triangle to be rendered. As a triangle is rendered, the **d?dX** registers are added to the the internal color component registers when the pixel drawn moves from left-to-right, and are subtracted from the internal color component registers when the pixel drawn moves from right-to-left. The **fd?dX** registers are floating point equivalents of the **d?dX**

registers. Avenger automatically converts both the **d?dX** and **fd?dX** registers into an internal fixed point notation used for rendering.

dRdX, dGdX, dBdX, dAdX

Bit	Description
23:0	Change in color with respect to X (fixed point two's complement 12.12 format)

fdRdX, fdGdX, fdBdX, fdAdX

Bit	Description
31:0	Change in color with respect to X (IEEE 32-bit single-precision floating point format)

8.9 dZdX and fdZdX Registers

The **dZdX** and **fdZdX** registers specify the change in Z with respect to X of a triangle to be rendered. As a triangle is rendered, the **dZdX** register is added to the the internal Z register when the pixel drawn moves from left-to-right, and is subtracted from the internal Z register when the pixel drawn moves from right-to-left. The **fdZdX** registers are floating point equivalents of the **dZdX** registers. Avenger automatically converts both the **dZdX** and **fdZdX** registers into an internal fixed point notation used for rendering.

dZdX

Bit	Description
31:0	Change in Z with respect to X (fixed point two's complement 20.12 format)

fdZdX

Bit	Description
31:0	Change in Z with respect to X (IEEE 32-bit single-precision floating point format)

8.10 dSdX, dTdX, fdSdX, and fdTdX Registers

The **dXdX**, **dTdX**, **fdSdX**, and **fdTdX** registers specify the change in the S/W and T/W texture coordinates with respect to X of a triangle to be rendered. As a triangle is rendered, the **d?dX** registers are added to the the internal S and T registers when the pixel drawn moves from left-to-right, and are subtracted from the internal S/W and T/W registers when the pixel drawn moves from right-to-left. Note that the delta S/W and T/W values used by Avenger for rendering must be divided by W prior to being sent to Avenger (i.e. Avenger uses $\Delta S/W$ and $\Delta T/W$). The **d?dX** registers are floating point equivalents of the **fd?dX** registers. Avenger automatically converts both the **d?dX** and **fd?dX** registers into an internal fixed point notation used for rendering.

dSdX, dTdX

Bit	Description
31:0	Change in S and T with respect to X (fixed point two's complement 14.18 format)

fdSdX, fdTdX

Bit	Description
31:0	Change in Z with respect to X (IEEE 32-bit single-precision floating point format)

8.11 dWdX and fdWdX Registers

The **dWdX** and **fdWdX** registers specify the change in 1/W with respect to X of a triangle to be rendered. As a triangle is rendered, the **dWdX** register is added to the the internal 1/W register when the pixel drawn moves from left-to-right, and is subtracted from the internal 1/W register when the pixel drawn

moves from right-to-left. The **fdWdX** registers are floating point equivalents of the **dWdX** registers. Avenger automatically converts both the **dWdX** and **fdWdX** registers into an internal fixed point notation used for rendering.

dWdX

Bit	Description
31:0	Change in W with respect to X (fixed point two's complement 2.30 format)

fdWdX

Bit	Description
31:0	Change in W with respect to X (IEEE 32-bit single-precision floating point format)

8.12 dRdY, dGdY, dBdY, dAdY, fdRdY, fdGdY, fdBdY, and fdAdY Registers

The **dRdY**, **dGdY**, **dBdY**, **dAdY**, **fdRdY**, **fdGdY**, **fdBdY**, and **fdAdY** registers specify the change in the color information (red, green, blue, and alpha) with respect to Y of a triangle to be rendered. As a triangle is rendered, the **d?dY** registers are added to the the internal color component registers when the pixel drawn in a positive Y direction, and are subtracted from the internal color component registers when the pixel drawn moves in a negative Y direction. The **fd?dY** registers are floating point equivalents of the **d?dY** registers. Avenger automatically converts both the **d?dY** and **fd?dY** registers into an internal fixed point notation used for rendering.

dRdY, dGdY, dBdY, dAdY

Bit	Description
23:0	Change in color with respect to Y (fixed point two's complement 12.12 format)

fdRdY, fdGdY, fdBdY, fdAdY

Bit	Description
31:0	Change in color with respect to Y (IEEE 32-bit single-precision floating point format)

8.13 dZdY and fdZdY Registers

The **dZdY** and **fdZdY** registers specify the change in Z with respect to Y of a triangle to be rendered. As a triangle is rendered, the **dZdY** register is added to the the internal Z register when the pixel drawn moves in a positive Y direction, and is subtracted from the internal Z register when the pixel drawn moves in a negative Y direction. The **fdZdY** registers are floating point equivalents of the **dZdY** registers. Avenger automatically converts both the **dZdY** and **fdZdY** registers into an internal fixed point notation used for rendering.

dZdY

Bit	Description
31:0	Change in Z with respect to Y (fixed point two's complement 20.12 format)

fdZdY

Bit	Description
31:0	Change in Z with respect to Y (IEEE 32-bit single-precision floating point format)

8.14 dSdY, dTdY, fdSdY, and fdTdY Registers

The **dYdY**, **dTdY**, **fdSdY**, and **fdTdY** registers specify the change in the S/W and T/W texture coordinates with respect to Y of a triangle to be rendered. As a triangle is rendered, the **d?dY** registers are added to the the internal S/W and T/W registers when the pixel drawn moves in a positive Y direction,

and are subtracted from the internal S/W and T/W registers when the pixel drawn moves in a negative Y direction. Note that the delta S/W and T/W values used by Avenger for rendering must be divided by W prior to being sent to Avenger (i.e. Avenger uses ΔS/W and ΔT/W). The **d?dY** registers are floating point equivalents of the **fd?dY** registers. Avenger automatically converts both the **d?dY** and **fd?dY** registers into an internal fixed point notation used for rendering.

dSdY, dTdY

Bit	Description
31:0	Change in S and T with respect to Y (fixed point two's complement 14.18 format)

fdSdY, fdTdY

Bit	Description
31:0	Change in Z with respect to Y (IEEE 32-bit single-precision floating point format)

8.15 dWdY and fdWdY Registers

The **dWdY** and **fdWdY** registers specify the change in 1/W with respect to Y of a triangle to be rendered. As a triangle is rendered, the **dWdY** register is added to the the internal 1/W register when the pixel drawn moves in a positive Y direction, and is subtracted from the internal 1/W register when the pixel drawn moves in a negative Y direction. The **fdWdY** registers are floating point equivalents of the **dWdY** registers. Avenger automatically converts both the **dWdY** and **fdWdY** registers into an internal fixed point notation used for rendering.

dWdY

Bit	Description
31:0	Change in W with respect to Y (fixed point two's complement 2.30 format)

fdWdY

Bit	Description
31:0	Change in W with respect to Y (IEEE 32-bit single-precision floating point format)

8.16 triangleCMD and ftriangleCMD Registers

The **triangleCMD** and **ftriangleCMD** registers execute the triangle drawing command. Writes to **triangleCMD** or **ftriangleCMD** initiate rendering a triangle defined by the **vertex**, **start**, **d?dX**, and **d?dY** registers. Note that the **vertex**, **start**, **d?dX**, and **d?dY** registers must be setup prior to writing to **triangleCMD** or **ftriangleCMD**. The value stored to **triangleCMD** or **ftriangleCMD** is the area of the triangle being rendered -- this value determines whether a triangle is clockwise or counter-clockwise geometrically. If bit(31)=0, then the triangle is oriented in a counter-clockwise orientation (i.e. positive area). If bit(31)=1, then the triangle is oriented in a clockwise orientation (i.e. negative area). To calculate the area of a triangle, the following steps are performed:

1. The vertices (A, B, and C) are sorted by the Y coordinate in order of increasing Y (i.e. A.y <= B.y <= C.y)
2. The area is calculated as follows:

   AREA = ((dxAB * dyBC) - (dxBC * dyAB)) / 2

   where

   dxAB = A.x - B.x
   dyBC = B.y - C.y
   dxBC = B.x - C.x
   dyAB = A.y - B.y

Note that Avenger only requires the sign bit of the area to be stored in the **triangleCMD** and **ftriangleCMD** registers -- bits(30:0) written to **triangleCMD** and **ftriangleCMD** are ignored.

triangleCMD

Bit	Description
31	Sign of the area of the triangle to be rendered

ftriangleCMD

Bit	Description
31	Sign of the area of the triangle to be rendered (IEEE 32-bit single-precision floating point format)

8.17 nopCMD Register

Writing any data to the **nopCMD** register executes the NOP command. Executing a NOP command flushes the graphics pipeline. When a **nopCMD** is exectuted with Bit(0)==1, the following counters get cleared: **fbiPixelsIn**, **fbiChromaFail**, **fbiZfuncFail**, **fbiAfuncFail**, and **fbiPixelsOut** registers. When a **nopCMD** is executed with Bit(1)==1, fbiTrianglesOut is cleared. When a **nopCMD** is executed with non of its bits set, none of the aforementioned counters will be modified.

Bit	Description
0	Clear **fbiPixelsIn**, **fbiChromaFail**, **fbiZfuncFail**, **fbiAfuncFail**, and **fbiPixelsOut** registers (1=clear registers)
1	Clear **fbiTrianglesOut** (1 = clear register).

8.18 fastfillCMD Register

Writing any data to the **fastfill** register executes the FASTFILL command. The FASTFILL command is used to clear the RGB and depth buffers as quickly as possible. Prior to executing the FASTFILL command, the **clipLeftRight** and **clipLowYHighY** are loaded with a rectangular area which is the desired area to be cleared. Note that **clip** registers define a rectangular area which is inclusive of the **clipLeft** and **clipLowY** register values, but exclusive of the **clipRight** and **clipHighY** register values. The **fastfillCMD** register is then written to initiate the FASTFILL command after the **clip** registers have been loaded. FASTFILL will optionally clear the color buffers with the RGB color specified in the **color1** register, and also optionally clears the depth buffer with the depth value taken from the **zaColor** register. Note that since **color1** is a 24-bit value, either dithering or bit truncation must be used to translate the 24-bit value into the native 16-bit frame buffer -- dithering may be employed optionally as defined by bit(8) of **fbzMode**. Disabling clearing of the color or depth buffers is accomplished by modifying the rgb/depth mask bits(10:9) in **fbzMode**. This allows individual or combined clearing of the RGB and depth buffers.

When using SGRAM, **fastfillCMD[0]** overrides **fbzMode[8],** and forces dithering off, allowing the color plane to be filled using SGRAM blockwrites. When using SDRAM, dithering behavior is determined solely by **fbzMode[8]**.

Special Notes: Avenger's **fastfillCMD** is not fully functional with SDRAM. The aux-plane fill portion of the comand doesn't work. To fastfill a surface with SDRAM, point the **colBufAddr** and **colBufStride** registers to the surface, and then fill it, using only the color-portion of the fastfill command (i.e. set the RGB mask=1 (**fbzMode[9]**, but clear the za-mask (**fbzmode[10]**)).

Bit	Description
0	Disable dithering during fastfill (1 = disable dithering).

8.19 swapbufferCMD Register

Writing to the **swapbufferCMD** register executes the SWAPBUFFER command. If the data written to **swapbufferCMD** bit(0)=0, then the frame buffer swapping is not synchronized with vertical retrace. If frame buffer swapping is not synchronized with vertical retrace, then visible frame "tearing" may occur. If **swapbufferCMD** bit(0)=1 then the frame buffer swapping is synchronized with vertical retrace. Synchronizing frame buffer swapping with vertical retrace eliminates the aforementioned frame "tearing." When a **swapbufferCMD** is received in the front-end PCI host FIFO, the swap buffers pending field in the status register is incremented. Conversely, when an actual frame buffer swapping occurs, the swap buffers pending field in the **status** register (bits(30:28)) is decremented. The swap buffers pending field allows software to determine how many SWAPBUFFER commands are present in the Avenger FIFOs. Bits(8:1) of **swapbufferCMD** are used to specify the number of vertical retraces to wait before swapping the color buffers. An internal counter is incremented whenever a vertical retrace occurs, and the color buffers are not swapped until the internal vertical retrace counter is greater than the value of **swapbufferCMD** bits(8:1) -- After a swap occurs, the internal vertical retrace counter is cleared.

Setting swabufferCMD[0]=1 is used to maintain constant frame rate. **NOTE: for highest performance when syncing-to-vsync, set the swapbuffer interval (swapbufferCMD bits(8:1)) to zero**.

SwapbufferCMD bit(9) disables swapping, which has the effect of decrementing the outstanding swap count, but not performing a video pointer swap. Note that if vertical retrace synchronization is disabled for swapping buffers (**swapbufferCMD**(0)=0), then the swap buffer interval field is ignored. The **swapbufferCMD** on Avenger works similar to Voodoo Rush. The driver must write to the swapbufferPend register to increase the outstanding swap count, then write to the **swapbufferCMD** register.

To enable triple buffering, turn on the appropriate bit in dram_init_1. If triple buffering is enabled, then the graphics core will be allowed to continue given that one or fewer swaps is pending to be done by the video unit. Effectively, this allows Avenger to render up to two frames ahead of the displayed buffer.

Bit	Description
0	Synchronize frame buffer swapping to vertical retrace (1=enable)
8:1	Swap buffer interval
9	Swap buffer disable swap

8.20 fbzColorPath Register

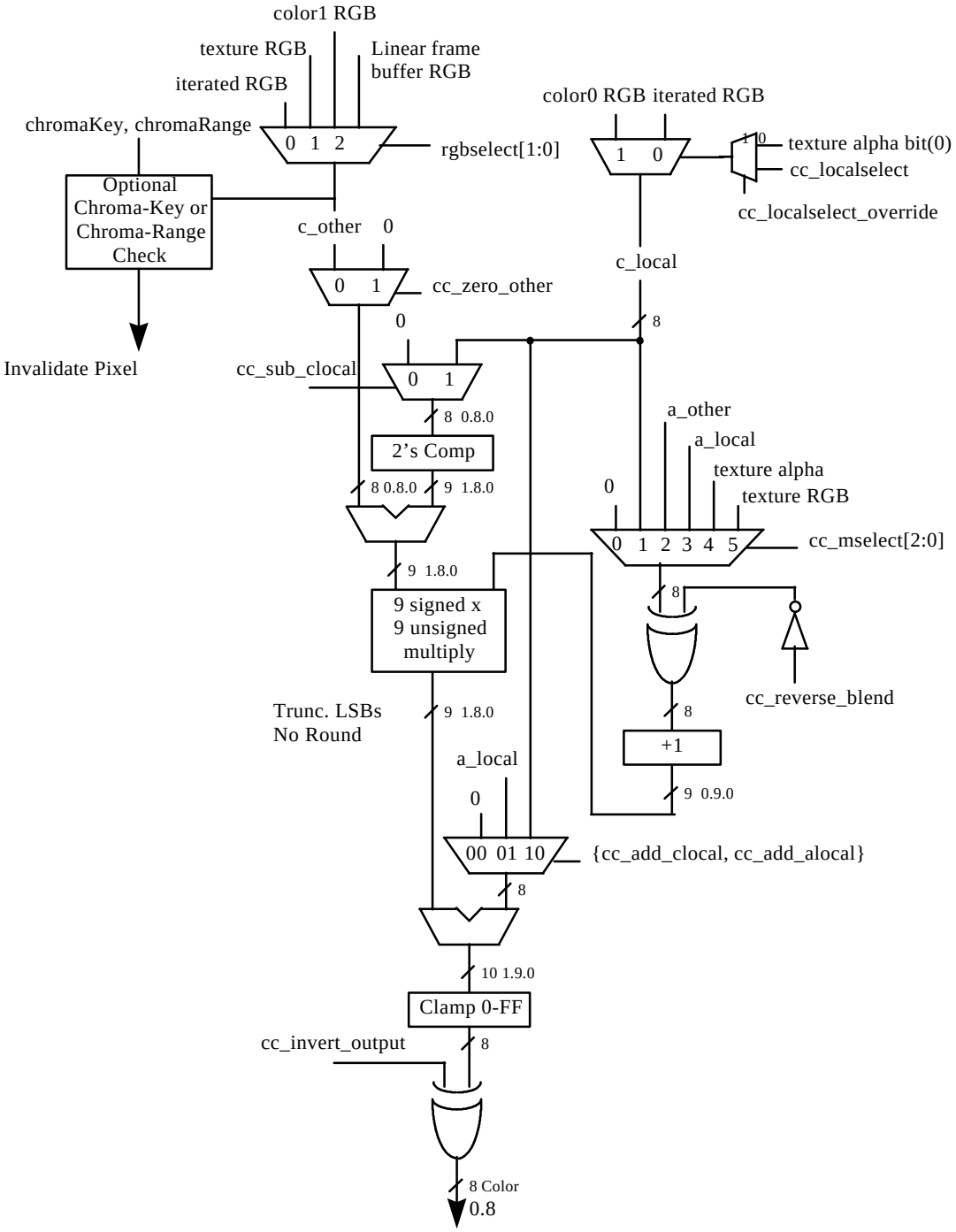
The **fbzColorPath** register controls the color and alpha rendering pixel pipelines. Bits in **fbzColorPath** control color/alpha selection and lighting. Individual bits of **fbzColorPath** are set to enable modulation, addition, etc. for various lighting effects including diffuse and specular highlights.

Bit	Description
1:0	RGB Select (0=Iterated RGB, 1=TREX Color Output, 2=**Color1** RGB, 3=Reserved)
3:2	Alpha Select (0=Iterated A, 1=TREX Alpha Output, 2=**Color1** Alpha, 3=Reserved)
4	Color Combine Unit control (cc_localselect mux control: 0=iterated RGB, 1=**Color0** RGB)
6:5	Alpha Combine Unit control (cca_localselect mux control: 0=iterated alpha, 1=**Color0** alpha, 2=iterated Z, 3=clamped iterated W)
7	Color Combine Unit control (cc_localselect_override mux control: 0=cc_localselect, 1=Texture alpha bit(7))

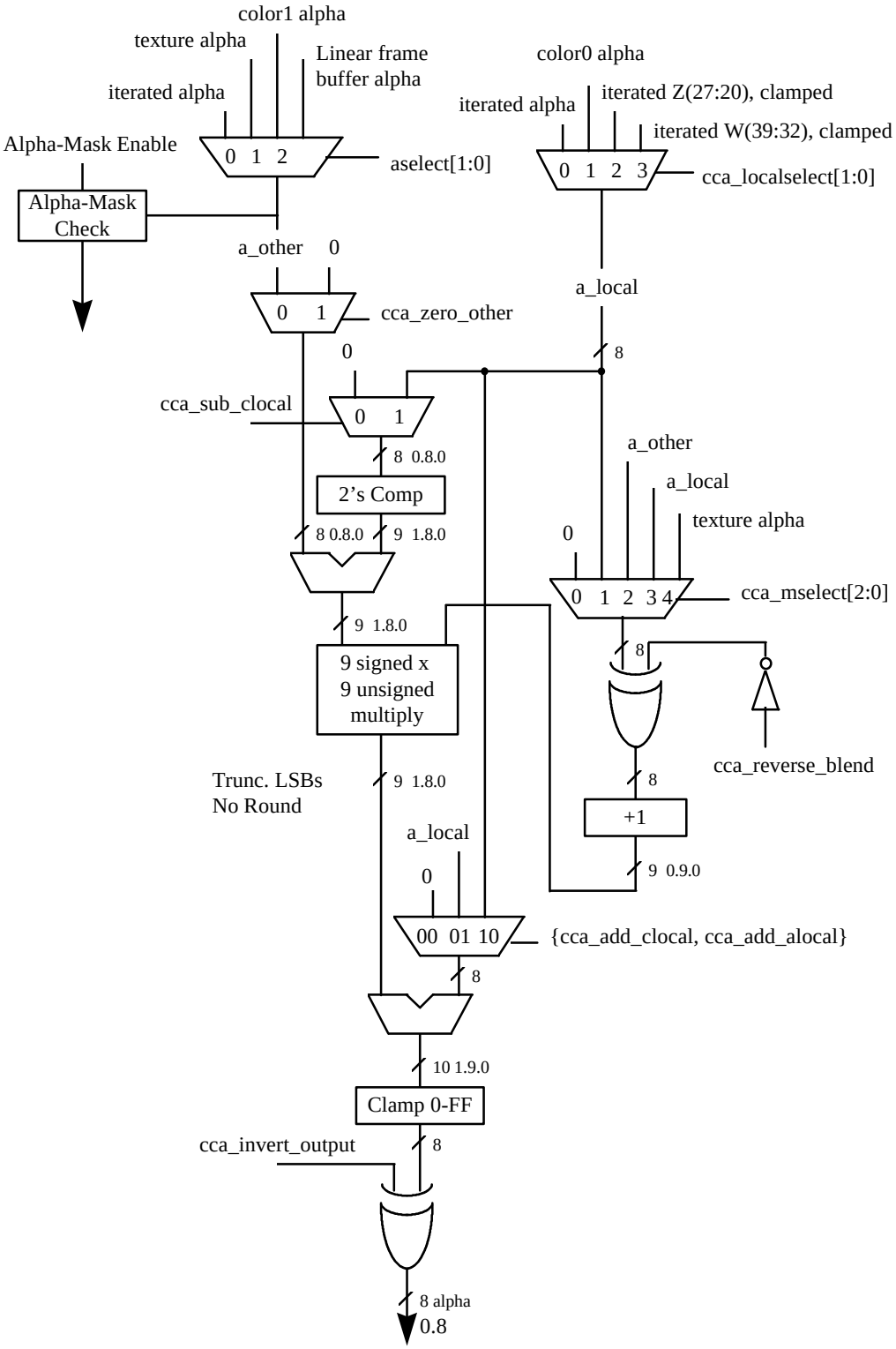
8	Color Combine Unit control (cc_zero_other mux control: 0=c_other, 1=zero)
9	Color Combine Unit control (cc_sub_clocal mux control: 0=zero, 1=c_local)
12:10	Color Combine Unit control (cc_mselect mux control: 0=zero, 1=c_local, 2=a_other, 3=a_local, 4=texture alpha, 5=texture rgb, 6-7=reserved)
13	Color Combine Unit control (cc_reverse_blend control)
14	Color Combine Unit control (cc_add_clocal control)
15	Color Combine Unit control (cc_add_alocal control)
16	Color Combine Unit control (cc_invert_output control)
17	Alpha Combine Unit control (cca_zero_other mux control: 0=a_other, 1=zero)
18	Alpha Combine Unit control (cca_sub_clocal mux control: 0=zero, 1=a_local)
21:19	Alpha Combine Unit control (cca_mselect mux control: 0=zero, 1=a_local, 2=a_other, 3=a_local, 4=texture alpha, 5-7=reserved)
22	Alpha Combine Unit control (cca_reverse_blend control)
23	Alpha Combine Unit control (cca_add_clocal control)
24	Alpha Combine Unit control (cca_add_alocal control)
25	Alpha Combine Unit control (cca_invert_output control)
26	Parameter Adjust (1=adjust parameters for subpixel correction)
27	Enable Texture Mapping (1=enable)
28	Enable RGBA, Z, and W parameter clamping (1=enable)
29	Enable anti-aliasing (1=enable)

Note that the color channels are controlled separately from the alpha channel. There are two primary color selection units: the Color Combine Unit(CCU) and the Alpha Combine Unit (ACU). Bits(1:0), bit(4), and bits(16:8) of **fbzColorPath** control the Color Combine Unit. The diagram below illustrates the Color Combine Unit controlled by the **fbzColorPath** register:

- Color Combine Unit -



Bits(3:2), bits(6:5), and bits(25:17) of **fbzColorPath** control the Alpha Combine Unit. The diagram below illustrates the Alpha Combine Unit controlled by the **fbzColorPath** register:



Bit(26) of **fbzColorPath** enables subpixel correction for all parameters. When enabled, Avenger will automatically subpixel correct the incoming color, depth, and texture coordinate parameters for triangles not aligned on integer spatial boundaries. Enabling subpixel correction decreases the on-chip triangle setup performance from 7 clocks to 16 clocks, but as the triangle setup engine is separately pipelined from the triangle rasterization engine, little if any performance penalty is seen when subpixel correction is enabled.

Important Note: When subpixel correction is enabled, the correction is performed on the **start** registers as they are passed into the triangle setup unit from the PCI FIFO. As a result, the host must pass down new starting parameter information for each new triangle -- if new starting parameter information is *not* passed down for a new triangle, the starting parameters will be subpixel corrected starting with the **start** registers already subpixel corrected for the last rendered triangle [in effect the parameters will be subpixel corrected twice, resulting in inaccuracies in the starting parameter values].

Bit(27) of **fbzColorPath** is used to enable texture mapping. If texture-mapped rendering is desired, then bit(27) of **fbzColorPath** must be set. When bit(27)=1, then data is transfered from TREX to FBI. If texture mapping is not desired (i.e. Gouraud shading, flat shading, etc.), then bit(27) may be cleared and no data is transfered from TREX to FBI.

Bit(28) of **fbzColorpath** is used to enable RGBA, Z, and W parameter clamping. When **fbzColorpath** bit(28)=1, then the RGBA triangle parameters are be clamped to [0,0xff] inclusive during triangle rasterization. Note that **fbzColorpath** bit(28) has no effect on the RGBA triangle parameters during triangle setup or sub-pixel correction. When **fbzColorpath** bit(28)=0, then the RGBA parameters are allowed to wrap according to the following formula:

```
if(rgbaIterator[23:12] == 0xfff)
      rgbaClamped[7:0] = 0x0;
else if(rgbaIterator[23:12] == 0x100)
      rgbaClamped[7:0] = 0xff;
else
      rgbaClamped[7:0] = rgbaIterator[19:12];
```

When **fbzColorpath** bit(28)=1, then the Z triangle parameter is clamped to [0,0xffff] inclusive during triangle rasterization. Note that **fbzColorpath** bit(28) has no effect on the Z triangle parameter during triangle setup or sub-pixel correction. Note also that the unclamped Z triangle iterator is used when performing floating point Z-buffering (**fbzMode** bit(21)=1). When **fbzColorpath** bit(28)=0, then the Z parameter is allowed to wrap according to the following formula:

```
if(zIterator[31:12] == 0xfffff)
      zClamped[15:0] = 0x0;
else if(zIterator[31:12] == 0x10000)
      zClamped[15:0] = 0xffff;
else
      zClamped[15:0] = zIterator[27:12];
```

When **fbzColorpath** bit(28)=1, then the W triangle parameter is clamped to [0,0xff] inclusive for use in the Alpha Combine Unit and the fog unit. Note that **fbzColorpath** bit(28) has no effect on the W triangle parameter during triangle setup or sub-pixel correction. Note also that the unclamped W triangle iterator is used when performing floating point W-buffering (**fbzMode** bit(21)=0). When **fbzColorpath** bit(28)=0, then the W parameter used as inputs to the ACU and fog units is allowed to wrap according to the following formula:

```
if(wIterator[47:32] == 0xffff)
```

```
        wClamped[7:0] = 0x0;
  else if(zIterator[47:32] == 0x0100)
        wClamped[7:0] = 0xff;
  else
        wClamped[7:0] = wIterator[39:32];
```

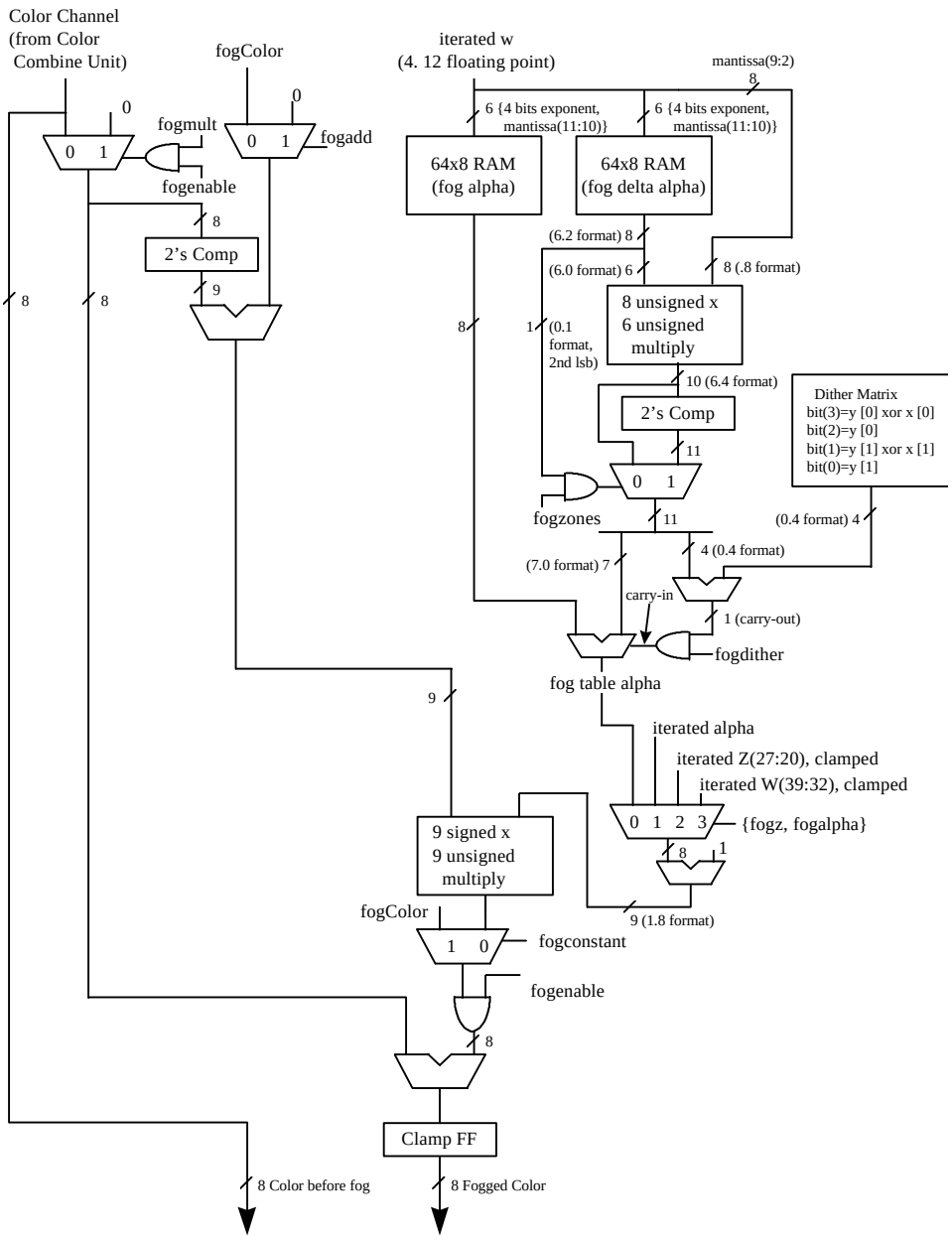
Bit(29) of **fbzColorpath** used to enable anti-aliasing. FIXME…

8.21 fogMode Register

The **fogMode** register controls the fog functionality of Avenger.

Bit	Description
0	Enable fog (1=enable)
1	Fog Unit control (fogadd control: 0=**fogColor**, 1=zero)
2	Fog Unit control (fogmult control: 0=Color Combine Unit RGB, 1=zero)
3	Fog Unit control (fogalpha control)
4	Fog Unit control (fogz control)
5	Fog Unit control (fogconstant control: 0=fog multiplier output, 1=**fogColor**)
6	Fog Unit control (fogdither control, dither the fog blending component)
7	Fog Unit control (fogzones control, enable signed fog delta)

The diagram below shows the fog unit of Avenger:



Bit(0) of **fogMode** is used to enable fog and atmospheric effects. When fog is enabled, the fog color specified in the **fogColor** register is blended with the source pixels as a function of the **fogTable** values and iterated W. Avenger supports a 64-entry lookup table (**fogTable**) to support atmospheric effects such as fog and haze. When enabled, the MSBs of a normalized floating point representation of (1/W) is used to index into the 64-entry fog table. The ouput of the lookup table is an "alpha" value which represents the level of blending to be performed between the static fog/haze color and the incoming pixel color. 8 lower order bits of the floating point (1/W) are used to blend between multiple entries of the lookup table to reduce fog "banding." The fog lookup table is loaded by the Host CPU, so various fog equations, colors, and effects can be supported.

The following table shows the mathematical equations for the supported values of bits(2:1) of **fogMode** when bits(5:3)=0:

Bit(0) - Enable Fog	Bit(1) - fogadd mux control	Bit(2) - fogmult mux control	Fog Equation
0	ignored	ignored	Cout = Cin
1	0	0	Cout = Afog*Cfog + (1-Afog)*Cin
1	0	1	Cout = Afog*Cfog
1	1	0	Cout = (1-Afog)*Cin
1	1	1	Cout = 0

where:

Cout = Color output from Fog block
Cin = Color input from Color Combine Unit Module
Cfog = **fogColor** register
AFog = alpha value calculated from Fog table

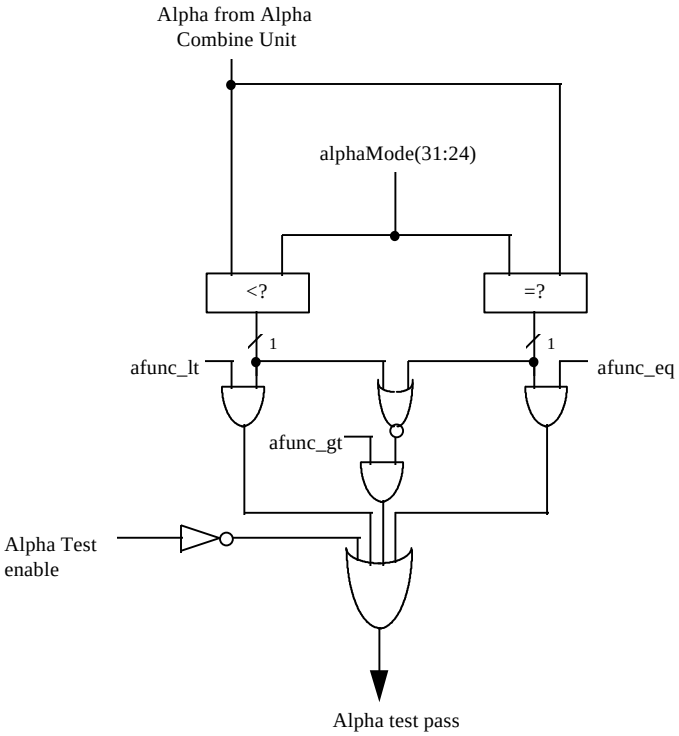
When bit(3) of **fogMode** is set, the integer part of the iterated alpha component is used as the fog alpha instead of the calculated fog alpha value from the fog table. When bit(4) of **fogMode** is set, the upper 8 bits of the iterated Z component are used as the fog alpha instead of the calculated fog alpha value from the fog table. If both bit(3) and bit(4) are set, then bit(4) takes precedence, and the upper 8 bits of the iterated Z component are used for the fog alpha value. Bit(5) of **fogMode** takes precedence over bits(4:3) and enables a constant value(**fogColor**) to be added to incoming source color.

8.22 alphaMode Register

The **alphaMode** register controls the alpha blending and anti-aliasing functionality of Avenger.

Bit	Description
0	Enable alpha function (1=enable)
3:1	Alpha function (see table below)
4	Enable alpha blending (1=enable)
5	reserved
7:8	reserved
11:8	Source RGB alpha blending factor (see table below)
15:12	Destination RGB alpha blending factor (see table below)
19:16	Source alpha-channel alpha blending factor (see table below)
23:20	Destination alpha-channel alpha blending factor (see table below)
31:24	Alpha reference value

Bits(3:1) specify the alpha function during rendering operations. The alpha function and test pipeline is shown below:



When **alphaMode** bit(0)=1, an alpha comparison is performed between the incoming source alpha and bits(31:24) of **alphaMode**. Section 5.18.1 below further describes the alpha function algorithm.

Bit(4) of **alphaMode** enables alpha blending. When alpha blending is enabled, the blending function is performed to combine the source color with the destination pixel. The blending factors of the source and destinations pixels are individually programmable, as determined by bits(23:8). Note that the RGB and alpha color channels may have different alpha blending factors. Section 5.18.2 below further describes alpha blending.

Bit(5) of **alphaMode** is reserved.

8.22.1 Alpha function

When the alpha function is enabled (**alphaMode** bit(0)=1), the following alpha comparison is performed:

AlphaSrc AlphaOP *AlphaRef*

where *AlphaSrc* represents the alpha value of the incoming source pixel, and *AlphaRef* is the value of bits(31:24) of **alphaMode**. A source pixel is written into an RGB buffer if the alpha comparison is true and writing into the RGB buffer is enabled (**fbzMode** bit(9)=1. If the alpha function is enabled and the alpha comparison is false, the **fbiAfuncFail** register is incremented and the pixel is invalidated in the pixel pipeline and no drawing occurs to the color or depth buffers. The supported alpha comparison functions (AlphaOPs) are shown below:

Value	AlphaOP Function
0	never
1	less than
2	equal
3	less than or equal
4	greater than

5	not equal
6	greater than or equal
7	always

8.22.2 Alpha Blending

When alpha blending is enabled (**alphaMode** bit(4)=1), incoming source pixels are blended with destination pixels. The alpha blending function for the RGB color components is as follows:

$$D_{new} \Leftarrow (S \cdot \alpha) + (D_{old} \cdot \beta)$$

where

D_{new}	The new destination pixel being written into the frame buffer
S	The new source pixel being generated
D_{old}	The old (current) destination pixel about to be modified
α	The source pixel alpha blending function.
β	The destination pixel alpha blending function.

The alpha blending function for the alpha components is as follows:

$$A_{new} \Leftarrow (AS \cdot \alpha d) + (A_{old} \cdot \beta d)$$

where

A_{new}	The new destination alpha being written into the alpha buffer
AS	The new source alpha being generated
A_{old}	The old (current) destination alpha about to be modified
αd	The source alpha alpha-blending function.
βd	The destination alpha alpha-blending function.

Note that the source and destination pixels may have different associated alpha blending functions. Also note that RGB color components and the alpha components may have different associated alpha blending functions. The alpha blending factors of the RGB color components are defined in bits(15:8) of **alphaMode**, while the alpha blending factors of the alpha component is specified in bits(23:16) of **alphaMode**. The following table lists the alpha blending functions supported:

Alpha Blending Function	Alpha Blending Function Pneumonic	Alpha Blending Function Description
0x0	AZERO	Zero
0x1	ASRC_ALPHA	Source alpha
0x2	A_COLOR	Color
0x3	ADST_ALPHA	Destination alpha
0x4	AONE	One
0x5	AOMSRC_ALPHA	1 - Source alpha
0x6	AOM_COLOR	1 - Color
0x7	AOMDST_ALPHA	1 - Destination alpha
0x8-0xe		Reserved
0xf (source alpha blending function)	ASATURATE	MIN(Source alpha, 1 - Destination alpha)
0xf (destination alpha blending function)	A_COLORBEFOREFOG	Color before Fog Unit

When the value 0x2 is selected as the destination alpha blending factor, the source pixel color is used as the destination blending factor. When the value 0x2 is selected as the source alpha blending factor, the destination pixel color is used as the source blending factor. Note also that the alpha blending function 0xf is different depending upon whether it is being used as a source or destination alpha blending function. When the value 0xf is selected as the destination alpha blending factor, the source color before the fog unit (“unfogged” color) is used as the destination blending factor -- this alpha blending function is useful for multi-pass rendering with atmospheric effects. When the value 0xf is selected as the source

alpha blending factor, the alpha-saturate anti-aliasing algorithm is selected -- this MIN function performs polygonal anti-aliasing for polygons which are drawn front-to-back.

8.23 lfbMode Register

The **lfbMode** register controls linear frame buffer accesses.

Bit	Description
3:0	Linear frame buffer write format (see table below)
5:4	Reserved
7:6	Reserved
8	Enable Avenger pixel pipeline-processed linear frame buffer writes (1=enable)
10:9	Linear frame buffer RGBA lanes (see tables below)
11	16-bit word swap linear frame buffer writes (1=enable)
12	Byte swizzle linear frame buffer writes (1=enable)
13	LFB access Y origin (0=top of screen is origin, 1=bottom of screen is origin)
14	Linear frame buffer write access W select (0=LFB selected, 1=**zacolor**[15:0]).
15	Reserved
16	Reserved

The following table shows the supported Avenger linear frame buffer write formats:

Value	Linear Frame Buffer Write Format
	16-bit formats
0	16-bit RGB (5-6-5)
1	16-bit RGB (x-5-5-5)
2	16-bit ARGB (1-5-5-5)
3	Reserved
	32-bit formats
4	24-bit RGB (x-8-8-8)
5	32-bit ARGB (8-8-8-8)
7:6	Reserved
11:8	Reserved
12	16-bit depth, 16-bit RGB (5-6-5)
13	16-bit depth, 16-bit RGB (x-5-5-5)
14	16-bit depth, 16-bit ARGB (1-5-5-5)
15	16-bit depth, 16-bit depth

When accessing the linear frame buffer, the cpu accesses information from the starting linear frame buffer (LFB) address space (see section 4 on Avenger address space) plus an offset which determines the <x,y> coordinates being accessed. Bits(3:0) of **lfbMode** define the format of linear frame buffer writes.

When writing to the linear frame buffer, **lfbMode** bit(8)=1 specifies that LFB pixels are processed by the normal Avenger pixel pipeline -- this implies each pixel written must have an associated depth and alpha value, and is also subject to the fog mode, alpha function, etc. If bit(8)=0, pixels written using LFB access bypass the normal Avenger pixel pipeline and are written to the specified buffer unconditionally and the values written are unconditionally written into the color/depth buffers except for optional color dithering [depth function, alpha blending, alpha test, and color/depth write masks are all bypassed when bit(8)=0]. If bit(8)=0, then only the buffers that are specified in the particular LFB format are updated. Also note

that if **lfbMode** bit(8)=0 that the color and Z mask bits in **fbzMode**(bits 9 and 10) are ignored for LFB writes. For example, if LFB modes 0-2, or 4 are used and bit(8)=0, then only the color buffers are updated for LFB writes (the depth buffer is unaffected by all LFB writes for these modes, regardless of the status of the Z-mask bit **fbzMode** bit 10). However, if LFB modes 12-14 are used and bit(8)=0, then both the color and depth buffers are updated with the LFB write data, irrespective of the color and Z mask bits in **fbzMode**. If LFB mode 15 is used and bit(8)=0, then only the depth buffer is updated for LFB writes (the color buffers are unaffected by all LFB writes in this mode, regardless of the status of the color mask bits in **fbzMode**).

If **lfbMode** bit(8)=0 and a LFB write format is selected which contains an alpha component (formats 2, 5, and 14) and the alpha buffer is enabled, then the alpha component is written into the alpha buffer. Conversely, if the alpha buffer is not enabled, then the alpha component of LFB writes using formats 2, 5, and 14 when bit(8)=0 are ignored. Note that anytime LFB formats 2, 5, and 14 are used when bit(8)=0 that blending and/or chroma-keying using the alpha component is not performed since the pixel-pipeline is bypassed when bit(8)=0.

If **lfbMode** bit(8)=0 and LFB write format 14 is used, the component that is ignored is determined by whether the alpha buffer is enabled -- If the alpha buffer is enabled and LFB write format 14 is used with bit(8)=0, then the depth component is ignored for all LFB writes. Conversely, if the alpha buffer is disabled and LFB write format is used with bit(8)=0, then the alpha component is ignored for all LFB writes.

If **lfbMode** bit(8)=1 and a LFB write access format does not include depth or alpha information (formats 0-5), then the appropriate depth and/or alpha information for each pixel written is taken from the **zaColor** register. Note that if bit(8)=1 that the LFB write pixels are processed by the normal Avenger pixel pipeline and thus are subject to the per-pixel operations including clipping, dithering, alpha-blending, alpha-testing, depth-testing, chroma-keying, fogging, and color/depth write masking.

Bits(10:9) of **lfbMode** specify the RGB channel format (color lanes) for linear frame buffer writes. The table below shows the Avenger supported RGB lanes:

Value	RGB Channel Format
0	ARGB
1	ABGR
2	RGBA
3	BGRA

Bit(11) of **lfbMode** defines the format of 2 16-bit data types passed with a single 32-bit writes. For linear frame buffer formats 0-2, two 16-bit data transfers can be packed into one 32-bit write -- bit(11) defines which 16-bit shorts correspond to which pixels on screen. The table below shows the pixel packing for packed 32-bit linear frame buffer formats 0-2:

lfbMode bit(11)	Screen Pixel Packing
0	Right Pixel(host data 31:16), Left Pixel(host data 15:0)
1	Left Pixel(host data 31:16), Right Pixel(host data 15:0)

For linear frame buffer formats 12-14, bit(11) of **lfbMode** defines the bit locations of the 2 16-bit data types passed. The table below shows the data packing for 32-bit linear frame buffer formats 12-14:

lfbMode bit(11)	Screen Pixel Packing
0	Z value(host data 31:16), RGB value(host data 15:0)

1	RGB value(host data 31:16), Z value(host data 15:0)

For linear frame buffer format 15, bit(11) of **lfbMode** defines the bit locations of the 2 16-bit depth values passed. The table below shows the data packing for 32-bit linear frame buffer format 15:

lfbMode bit(11)	Screen Pixel Packing
0	Z Right Pixel(host data 31:16), Z Left Pixel(host data 15:0)
1	Z left Pixel(host data 31:16), Z Right Pixel(host data 15:0)

Note that bit(11) of **lfbMode** is ignored for linear frame buffer writes using formats 4 or 5.

Bit(12) of **lfbMode** is used to enable byte swizzling. When byte swizzling is enabled, the 4-bytes within a 32-bit word are swizzled to correct for endian differences between Avenger and the host CPU. For little endian CPUs (e.g. Intel x86 processors) byte swizzling should not be enabled, however big endian CPUs (e.g. PowerPC processors) should enable byte swizzling. For linear frame buffer writes, the bytes within a word are swizzled prior to being modified by the other control bits of **lfbMode**. When byte swizzling is enabled, bits(31:24) are swapped with bits(7:0), and bits(23:16) are swapped with bits(15:8).

Very Important Note: The order of swapping and swizzling operations for LFB writes is as follows: byte swizzling is performed first on all incoming LFB data, as defined by **lfbMode** bit(12) and irrespective of the LFB data format. After byte swizzling, 16-bit word swapping is performed as defined by **lfbMode** bit(11). Note that 16-bit word swapping is never performed on LFB data when data formats 4 and 5 are used. Also note that 16-bit word swapping is performed on the LFB data that was previously optionally swapped. Finally, after both swizzling and 16-bit word swapping are performed, the individual color channels are selected as defined in **lfbMode** bits(10:9). Note that the color channels are selected on the LFB data that was previously swizzled and/or swapped

Bit(13) of **lfbMode** is used to define the origin of the Y coordinate for all linear frame buffer writes when the pixel pipeline is bypassed (**lfbMode** bit(8)=0). Note that bit(13) of **lfbMode** does not affect rendering operations (FASTFILL and TRIANGLE commands) -- bit(17) of **fbzMode** defines the origin of the Y coordinate for rendering operations. Note also that if the pixel pipeline is enabled for linear frame buffer writes (**lfbMode** bit(8)=1), then **fbzMode** bit(17) is used to determine the location of the Y origin. When cleared, the Y origin (Y=0) for all linear frame buffer accesses is defined to be at the top of the screen. When bit(13) is set, the Y origin for all linear frame buffer accesses is defined to be at the bottom of the screen.

Bit(14) of **lfbMode** is used to select the W component used for LFB writes processed through the pixel pipeline. If bit(14)=0, then the MSBs of the fractional component of the 48-bit W value passed to the pixel pipeline for LFB writes through the pixel pipeline is the 16-bit Z value associated with the LFB write. [Note that the 16-bit Z value associated with the LFB write is dependent on the LFB format, and is either passed down pixel-by-pixel from the CPU, or is set to the constant **zaColor**(15:0)]. If bit(14)=1, then the MSBs of the fractional component of the 48-bit W value passed to the pixel pipeline for LFB writes is **zacolor**(15:0). Regardless of the setting of bit(14), when LFB writes go through the pixel pipeline, all other bits except the 16 MSBs of the fractional component of the W value are set to 0x0. Note that bit(14) is ignored if LFB writes bypass the pixel pipeline.

8.23.1 Linear Frame Buffer Writes

Linear frame buffer writes -- format 0:
When writing to the linear frame buffer with 16-bit format 0 (RGB 5-6-5), the RGB channel format specifies the RGB ordering within a 16-bit word. If the Avenger pixel pipeline is enabled for LFB

accesses (**lfbMode** bit(8)=1), then alpha and depth information for LFB format 0 is taken from the **zaColor** register. The following table shows the color channels for 16-bit linear frame buffer access format 0:

RGB Channel Format Value	16-bit Linear frame buffer access bits	RGB Channel
0	15:0	Red (15:11), Green(10:5), Blue(4:0)
1	15:0	Blue (15:11), Green(10:5), Red(4:0)
2	15:0	Red (15:11), Green(10:5), Blue(4:0)
3	15:0	Blue (15:11), Green(10:5), Red(4:0)

Linear frame buffer writes -- format 1:

When writing to the linear frame buffer with 16-bit format 1 (RGB 5-5-5), the RGB channel format specifies the RGB ordering within a 16-bit word. If the Avenger pixel pipeline is enabled for LFB accesses (**lfbMode** bit(8)=1), then alpha and depth information for LFB format 1 is taken from the **zaColor** register. The following table shows the color channels for 16-bit linear frame buffer access format 1:

RGB Channel Format Value	16-bit Linear frame buffer access bits	RGB Channel
0	15:0	Ignored(15), Red (14:10), Green(9:5), Blue(4:0)
1	15:0	Ignored(15), Blue (14:10), Green(9:5), Red(4:0)
2	15:0	Red (15:11), Green(10:6), Blue(5:1), Ignored(0)
3	15:0	Blue (15:11), Green(10:6), Red(5:1), Ignored(0)

Linear frame buffer writes -- format 2:

When writing to the linear frame buffer with 16-bit format 2 (ARGB 1-5-5-5), the RGB channel format specifies the RGB ordering within a 16-bit word. If the Avenger pixel pipeline is enabled for LFB accesses (**lfbMode** bit(8)=1), then depth information for LFB format 2 is taken from the **zaColor** register. Note that the 1-bit alpha value passed when using LFB format 2 is bit-replicated to yield the 8-bit alpha used in the pixel pipeline. The following table shows the color channels for 16-bit linear frame buffer access format 2:

RGB Channel Format Value	16-bit Linear frame buffer access bits	RGB Channel
0	15:0	Alpha(15), Red (14:10), Green(9:5), Blue(4:0)
1	15:0	Alpha(15), Blue (14:10), Green(9:5), Red(4:0)
2	15:0	Red (15:11), Green(10:6), Blue(5:1), Alpha(0)
3	15:0	Blue (15:11), Green(10:6), Red(5:1), Alpha(0)

Linear frame buffer writes -- format 3:

Linear frame buffer format 3 is an unsupported format.

Linear frame buffer writes -- format 4:

When writing to the linear frame buffer with 24-bit format 4 (RGB x-8-8-8), the RGB channel format specifies the RGB ordering within a 24-bit word. Note that the alpha/A channel is ignored for 24-bit access format 4. Also note that while only 24-bits of data is transfered for format 4, all data access must be 32-bit aligned -- packed 24-bit writes are not supported by Avenger. If the Avenger pixel pipeline is enabled for LFB accesses (**lfbMode** bit(8)=1), then alpha and depth information for LFB format 4 is taken from the **zaColor** register. The following table shows the color channels for 24-bit linear frame buffer access format 4:

RGB Channel Format Value	24-bit Linear frame buffer access bits (aligned to 32-bits)	RGB Channel
0	31:0	Ignored(31:24), Red (23:16), Green(15:8), Blue(7:0)
1	31:0	Ignored(31:24), Blue(23:16), Green(15:8), Red(7:0)
2	31:0	Red(31:24), Green(23:16), Blue(15:8), Ignored(7:0)
3	31:0	Blue(31:24), Green(23:16), Red(15:8), Ignored(7:0)

Linear frame buffer writes -- format 5:
When writing to the linear frame buffer with 32-bit format 5 (ARGB 8-8-8-8), the RGB channel format specifies the ARGB ordering within a 32-bit word. If the Avenger pixel pipeline is enabled for LFB accesses (**lfbMode** bit(8)=1), then depth information for LFB format 5 is taken from the **zaColor** register. The following table shows the color channels for 32-bit linear frame buffer access format 5.

RGB Channel Format Value	24-bit Linear frame buffer access bits (aligned to 32-bits)	RGB Channel
0	31:0	Alpha(31:24), Red (23:16), Green(15:8), Blue(7:0)
1	31:0	Alpha(31:24), Blue(23:16), Green(15:8), Red(7:0)
2	31:0	Red(31:24), Green(23:16), Blue(15:8), Alpha(7:0)
3	31:0	Blue(31:24), Green(23:16), Red(15:8), Alpha(7:0)

Linear frame buffer writes -- formats 6-11:
Linear frame buffer formats 6-11 are unsupported formats.

Linear frame buffer writes -- format 12:
When writing to the linear frame buffer with 32-bit format 12 (Depth 16, RGB 5-6-5), the RGB channel format specifies the RGB ordering within the 32-bit word. If the Avenger pixel pipeline is enabled for LFB accesses (**lfbMode** bit(8)=1), then alpha information for LFB format 12 is taken from the **zaColor** register. Note that the format of the depth value passed when using LFB format 12 must precisely match the format of the type of depth buffering being used (either 16-bit integer Z or 16-bit floating point 1/W). The following table shows the 16-bit color channels within the 32-bit linear frame buffer access format 12:

RGB Channel Format Value	16-bit Linear frame buffer access bits	RGB Channel
0	15:0	Red (15:11), Green(10:5), Blue(4:0)
1	15:0	Blue (15:11), Green(10:5), Red(4:0)
2	15:0	Red (15:11), Green(10:5), Blue(4:0)
3	15:0	Blue (15:11), Green(10:5), Red(4:0)

Linear frame buffer writes -- format 13:
When writing to the linear frame buffer with 32-bit format 13 (Depth 16, RGB x-5-5-5), the RGB channel format specifies the RGB ordering within the 32-bit word. If the Avenger pixel pipeline is enabled for LFB accesses (**lfbMode** bit(8)=1), then alpha information for LFB format 13 is taken from the **zaColor** register. Note that the format of the depth value passed when using LFB format 13 must precisely match the format of the type of depth buffering being used (either 16-bit integer Z or 16-bit floating point 1/W). The following table shows the 16-bit color channels within the 32-bit linear frame buffer access format 13:

RGB Channel Format Value	16-bit Linear frame buffer access bits	RGB Channel
0	15:0	Ignored(15), Red (14:10), Green(9:5), Blue(4:0)
1	15:0	Ignored(15), Blue (14:10), Green(9:5), Red(4:0)
2	15:0	Red (15:11), Green(10:6), Blue(5:1), Ignored(0)
3	15:0	Blue (15:11), Green(10:6), Red(5:1), Ignored(0)

Linear frame buffer writes -- format 14:

When writing to the linear frame buffer with 32-bit format 14 (Depth 16, ARGB 1-5-5-5), the RGB channel format specifies the RGB ordering within the 32-bit word. Note that the format of the depth value passed when using LFB format 14 must precisely match the format of the type of depth buffering being used (either 16-bit integer Z or 16-bit floating point 1/W). Also note that the 1-bit alpha value passed when using LFB format 14 is bit-replicated to yield the 8-bit alpha used in the pixel pipeline. The following table shows the 16-bit color channels within the 32-bit linear frame buffer access format 14:

RGB Channel Format Value	16-bit Linear frame buffer access bits	RGB Channel
0	15:0	Alpha(15), Red (14:10), Green(9:5), Blue(4:0)
1	15:0	Alpha(15), Blue (14:10), Green(9:5), Red(4:0)
2	15:0	Red (15:11), Green(10:6), Blue(5:1), Alpha(0)
3	15:0	Blue (15:11), Green(10:6), Red(5:1), Alpha(0)

Linear frame buffer writes -- format 15:

When writing to the linear frame buffer with 32-bit format 15 (Depth 16, Depth 16), the format of the depth values passed must precisely match the format of the type of depth buffering being used (either 16-bit integer Z or 16-bit floating point 1/W). If the Avenger pixel pipeline is enabled for LFB accesses (**lfbMode** bit(8)=1), then RGB color information is taken from the **color1** register, and alpha information for LFB format 15 is taken from the **zaColor** register.

8.24 fbzMode Register

The **fbzMode** register controls frame buffer and depth buffer rendering functions of the Avenger processor. Bits in **fbzMode** control clipping, chroma-keying, depth-buffering, dithering, and masking.

Bit	Description
0	Enable clipping rectangle (1=enable)
1	Enable chroma-keying (1=enable)
2	Enable stipple register masking (1=enable)
3	W-Buffer Select (0=Use Z-value for depth buffering, 1=Use W-value for depth buffering)
4	Enable depth-buffering (1=enable)
7:5	Depth-buffer function (see table below)
8	Enable dithering (1=enable)
9	RGB buffer write mask (0=disable writes to RGB buffer)
10	Depth/alpha buffer write mask (0=disable writes to depth/alpha buffer)
11	Dither algorithm (0=4x4 ordered dither, 1=2x2 ordered dither)
12	Enable Stipple pattern masking (1=enable)
13	Enable Alpha-channel mask (1=enable alpha-channel masking)
15:14	Reserved
16	Enable depth-biasing (1=enable)

17	Rendering commands Y origin (0=top of screen is origin, 1=bottom of screen is origin)
18	Enable alpha planes (1=enable)
19	Enable alpha-blending dither subtraction (1=enable)
20	Depth buffer source compare select (0=normal operation, 1=**zaColor**[15:0])
21	Depth float select (0=iterated W is used for floating point depth buffering, 1=iterated Z is used for floating point depth buffering)

Bit(0) of **fbzMode** is used to enable the clipping register. When set, clipping to the rectangle defined by the **clipLeftRight** and **clipBottomTop** registers inclusive is enabled. When clipping is enabled, the bounding clipping rectangle must always be less than or equal to the screen resolution in order to clip to screen coordinates. Also note that if clipping is not enabled, rendering may not occur outside of the screen resolution. Bit(1) of **fbzMode** is used to enable the color compare check (chroma-keying). When enabled, any source pixel matching the color specified in the **chromaKey** register is not written to the RGB buffer. The chroma-key color compare is performed immediately after texture mapping lookup, but before the color combine unit and fog in the pixel datapath.

Bit(2) of **fbzMode** is used to enable stipple register masking. When enabled, bit(12) of **fbzMode** is used to determine the stipple mode -- bit(12)=0 specifies stipple rotate mode, while bit(12)=1 specifies stipple pattern mode.

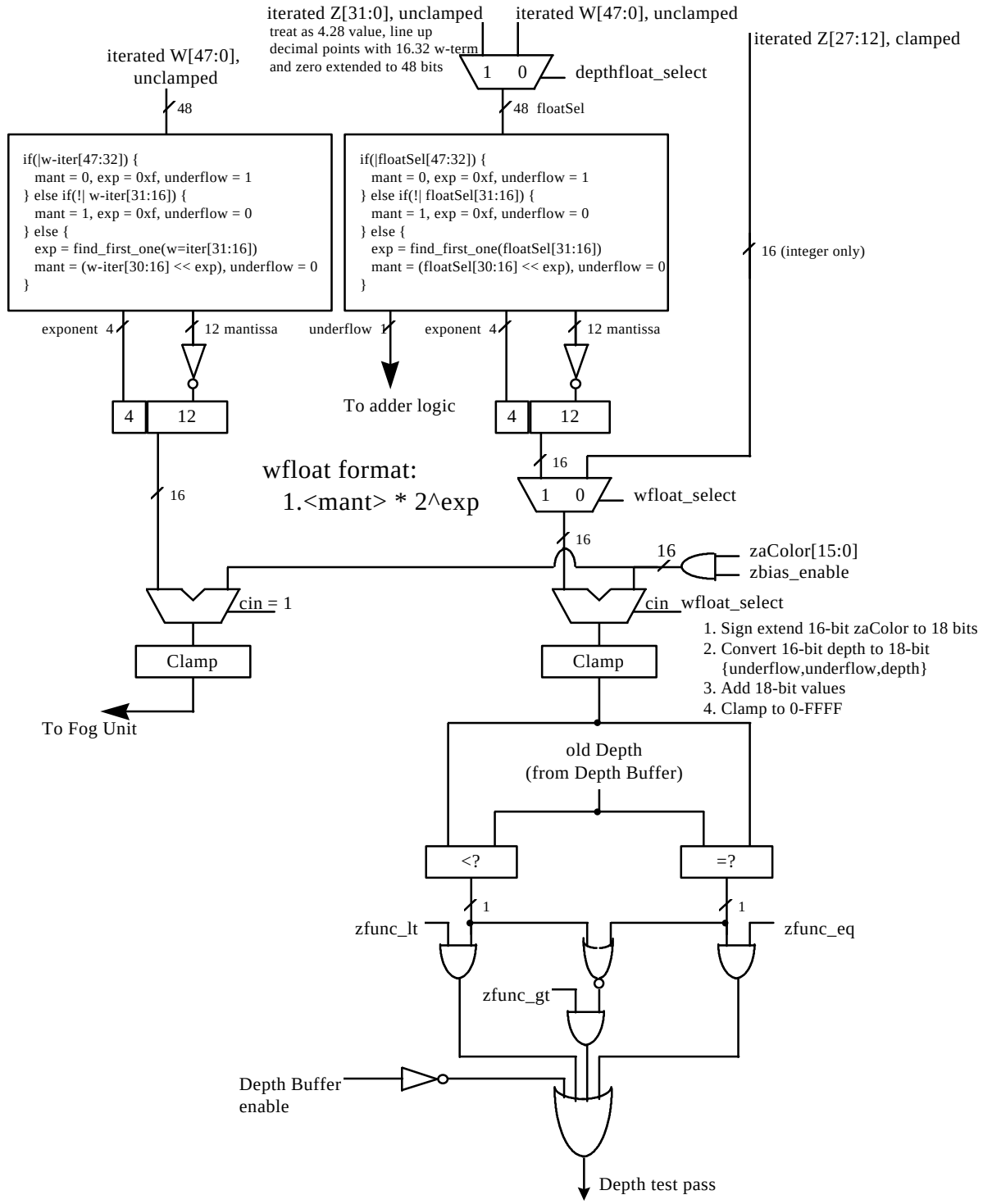
When stipple register masking is enabled and stipple rotate mode is selected, bit(31) of the **stipple** register is used to mask pixels in the pixel pipeline. For all triangle commands and linear frame buffer writes through the pixel pipeline, pixels are invalidated in the pixel pipeline if **stipple** bit(31)=0 and stipple register masking is enabled in stipple rotate mode. After an individual pixel is processed in the pixel pipeline, the **stipple** register is rotated from right-to-left, with the value of bit(0) filled with the value of bit(31). Note that the **stipple** register is rotated regardless of whether stipple masking is enabled (bit(2) in **fbzMode**) when in stipple rotate mode.

When stipple register masking is enabled and stipple pattern mode is selected, the spatial <x,y> coordinates of a pixel processed in the pixel pipeline are used to lookup a 4x8 monochrone pattern stored in the **stipple** register -- the resultant lookup value is used to mask pixels in the pixel pipeline. For all triangle commands and linear frame buffer writes through the pixel pipeline, a stipple bit is selected from the **stipple** register as follows:

```
switch(pixel_Y[1:0]) {
        case 0: stipple_Y_sel[7:0] = stipple[7:0];
        case 1: stipple_Y_sel[7:0] = stipple[15:8];
        case 2: stipple_Y_sel[7:0] = stipple[23:16];
        case 3: stipple_Y_sel[7:0] = stipple[31:24];
}
switch(pixel_X[2:0] {
        case 0: stipple_mask_bit = stipple_Y_sel[7];
        case 1: stipple_mask_bit = stipple_Y_sel[6];
        case 2: stipple_mask_bit = stipple_Y_sel[5];
        case 3: stipple_mask_bit = stipple_Y_sel[4];
        case 4: stipple_mask_bit = stipple_Y_sel[3];
        case 5: stipple_mask_bit = stipple_Y_sel[2];
        case 6: stipple_mask_bit = stipple_Y_sel[1];
        case 7: stipple_mask_bit = stipple_Y_sel[0];
}
```

If the stipple_mask_bit=0, the pixel is invalidated in the pixel pipeline when stipple register masking is enabled and stipple pattern mode is selected. Note that when stipple pattern mode is selected the **stipple** register is never rotated.

Bits(4:3) specify the depth-buffering function during rendering operations. The depth buffering pipeline is shown below:



Bit(4) of **fbzMode** is used to enable depth-buffering. When depth buffering is enabled, a depth comparison is performed for each source pixel as defined in bits(7:5). When bit(3)=0, the **z** iterator is used for the depth buffer comparison. When bit(3)=1, the **w** iterator is used for the depth buffer comparison. When bit(3)=1 enabling w-buffering, the inverse of the normalized **w** iterator is used for the

depth-buffer comparison. This in effect implements a floating-point w-buffering scheme utilizing a 4-bit exponent and a 12-bit mantissa. The inverted **w** iterator is used so that the same depth buffer comparisons can be used as with a typical z-buffer. Section 5.19.1 below further describes the depth-buffering algorithm.

Bit(8) of **fbzMode** enables 16-bit color dithering. When enabled, native 24-bit source pixels are dithered into 16-bit RGB color values with no performance penalty. When dithering is disabled, native 24-bit source pixels are converted into 16-bit RGB color values by bit truncation. When dithering is enabled, bit(11) of **fbzMode** defines the dithering algorithm -- when bit(11)=0 a 4x4 ordered dither algorithm is used, and when bit(11)=1 a 2x2 ordered dither algorithm is used to convert 24-bit RGB pixels into 16-bit frame buffer colors.

Bit(9) of **fbzMode** enables writes to the RGB buffers. Clearing bit(9) invalidates all writes to the RGB buffers, and thus the RGB buffers remain unmodified for all rendering operations. Bit(9) must be set for normal drawing into the RGB buffers. Similarly, bit(10) enables writes to the depth-buffer. When cleared, writes to the depth-buffer are invalidated, and the depth-buffer state is unmodified for all rendering operations. Bit(10) must be set for normal depth-buffered operation.

Bit(13) of **fbzMode** enables the alpha-channel mask. When enabled, bit(0) of the incoming alpha value is used to mask writes to the color and depth buffers. If alpha channel masking is enabled and bit(0) of the incoming alpha value is 0, then the pixel is invalidated in the pixel pipeline, the **fbiAfuncFail** register is incremented, and no drawing occurs to the color or depth buffers. If alpha channel masking is enabled and bit(0) of the incoming alpha value is 1, then the pixel is drawn normally subject to depth function, alpha blending function, alpha test, and color/depth masking.

Bit(16) of **fbzMode** is used to enable the Depth Buffer bias. When bit(16)=1, the calculated depth value (irrespective of Z or 1/W type of depth buffering selected) is added to bits(15:0) of **zaColor**. Depth buffer biasing is used to elimate aliasing artifacts when rendering co-planar polygons.

Bit(17) of **fbzMode** is used to define the origin of the Y coordinate for rendering operations (FASTFILL and TRIANGLE commands) and linear frame buffer writes when the pixel pipeline is bypassed for linear frame buffer writes (**lfbMode** bit(8)=0). Note that bit(17) of **fbzMode** does not affect linear frame buffer writes when the pixel pipeline is bypassed for linear frame buffer writes (**lfbMode** bit(8)=0), as in this situation bit(13) of **lfbMode** specifies the Y origin for linear frame buffer writes. When cleared, the Y origin (Y=0) for all rendering operations and linear frame buffer writes when the pixel pipeline is enabled is defined to be at the top of the screen. When bit(17) is set, the Y origin is defined to be at the bottom of the screen.

Bit(18) of **fbzMode** is used to enable the destination alpha planes. When set, the auxiliary buffer will be used as destination alpha planes. Note that if bit(18) of **fbzMode** is set that depth buffering cannot be used, and thus bit(4) of **fbzMode** (enable depth buffering) must be set to 0x0.

Bit(19) of **fbzMode** is used to enable dither subtraction on the destination color during alpha blending. When dither subtraction is enabled (**fbzMode** bit(19)=1), the dither matrix used to convert 24-bit color to 16-bit color is subtracted from the destination color before applying the alpha-blending algorithm. Enabling dither subtraction is used to enhance image quality when performing alpha-blending.

Bit(20) of **fbzMode** is used to select the source depth value used for depth buffering. When **fbzMode** bit(20)=0, the source depth value used for the depth buffer comparison is either iterated Z or iterated W (as selected by **fbzMode** bit(3)) and may be biased (as controlled by **fbzMode** bit(16)). When **fbzMode** bit(20)=1, the constant depth value defined by **zaColor**[15:0] is used as the source depth value for the

depth buffer comparison. Regardless of the state of **fbzMode** bit(20), the biased iterated Z/W is written into the depth buffer if the depth buffer function passes.

Bit(21) of **fbzMode** is used to select either the **w** iterator or the **z** iterator to be used for floating point depth buffering. Floating point depth buffering is enabled when **fbzMode** bit(4)=1. When **fbzMode** bit(21)=0, then the unclamped **w** iterator is converted to a 4.12 floating point representation and used for depth buffering. When **fbzMode** bit(21)=1, then the unclamped **z** iterator is converted into a 4.12 floating point format and used for depth buffering.

8.24.1 Depth-buffering function

When the depth-buffering is enabled (**fbzMode** bit(4)=1), the following depth comparison is performed:

DEPTHsrc DepthOP *DEPTHdst*

where *DEPTHsrc* and *DEPTHdst* represent the depth source and destination values respectively. A source pixel is written into an RGB buffer if the depth comparison is true and writing into the RGB buffer is enabled (**fbzMode** bit(9)=1). The source depth value is written into the depth buffer if the depth comparison is true and writing into the depth buffer is enabled (**fbzMode** bit(10)=1). The supported depth comparison functions (DepthOPs) are shown below:

Value	DepthOP Function
0	never
1	less than
2	equal
3	less than or equal
4	greater than
5	not equal
6	greater than or equal
7	always

8.25 stipple Register

The **stipple** register specifies a mask which is used to enable individual pixel writes to the RGB and depth buffers. See the stipple functionality description in the **fbzMode** register description for more information.

Bit	Description
31:0	stipple value

8.26 color0 Register

The **color0** register specifies constant color values which are used for certain rendering functions. In particular, bits(23:0) of **color0** are optionally used as the c_local input in the color combine unit. In addition, bits(31:24) of **color0** are optionally used as the c_local input in the alpha combine unit. See the **fbzColorPath** register description for more information.

Bit	Description
7:0	Constant Color Blue
15:8	Constant Color Green
23:16	Constant Color Red
31:24	Constant Color Alpha

8.27 color1 Register

The **color1** register specifies constant color values which are used for certain rendering functions. In particular, bits(23:0) of **color1** are optionally used as the c_other input in the color combine unit selected by bits(1:0) of **fbzColorPath**. The alpha component of **color1**(bits(31:24)) are optionally used as the a_other input in the alpha combine unit selected by bits(3:2) of **fbzColorPath**. The **color1** register bits(23:0) are also used by the FASTFILL command as the constant color for screen clears. Also, for linear frame buffer write format 15(16-bit depth, 16-bit depth), the color for the pixel pipeline is taken from **color1** if the pixel pipeline is enabled for linear frame buffer writes (**lfbMode** bit(8)=1).

Bit	Description
7:0	Constant Color Blue
15:8	Constant Color Green
23:16	Constant Color Red
31:24	Constant Color Alpha

8.28 fogColor Register

The **fogColor** register is used to specify the fog color for fogging operations. Fog is enabled by setting bit(0) in **fogMode**. See the **fogMode** and **fogTable** register descriptions for more information fog.

Bit	Description
7:0	Fog Color Blue
15:8	Fog Color Green
23:16	Fog Color Red
31:24	reserved

8.29 zaColor Register

The **zaColor** register is used to specify constant alpha and depth values for linear frame buffer writes, FASTFILL commands, and co-planar polygon rendering support. For certain linear frame buffer access formats, the alpha and depth values associated with a pixel written are the values specified in **zaColor**. See the **lfbMode** register description for more information. When executing the FASTFILL command, the constant 16-bit depth value written into the depth buffer is taken from bits(15:0) of **zaColor**. When **fbzMode** bit(16)=1 enabling depth-biasing, the constant depth value required is taken from **zaColor** bits(15:0).

Bit	Description
15:0	Constant Depth
23:16	reserved
31:24	Constant Alpha

8.30 chromaKey Register

The **chromaKey** register specifies a color which is compared with all pixels to be written into the RGB buffer. If a color match is detected between an outgoing pixel and the **chromaKey** register, and chroma-keying is enabled (bit(1)=1 in the **fbzMode** register), then the pixel is not written into the frame buffer. An outgoing pixel will still be written into the RGB buffer if chroma-keying is disabled or the **chromaKey** color does not equal the outgoing pixel color. Note that the alpha color component of an outgoing pixel is ignored in the chroma-key color match circuitry. The chroma-key comparison is performed immediately after texture lookup, but before lighting, fog, or alpha blending. See the

description of the **fbzColorPath** register for further information on the location of the chroma-key comparison circuitry. The format of **chromaKey** is a 24-bit RGB color.

Bit	Description
7:0	Chroma-key Blue
15:8	Chroma-key Green
23:16	Chroma-key Red
31:24	reserved

8.31 chromaRange Register

The **chromaRange** register specifies a 24-bit RGB color value which is comared to all pixels to be written to the color buffer. If chroma-keying is enabled (**fbzMode[1]**) and chroma-ranging is enabled (**chromaRange[28])**, the outgoing pixel color is compared to a color range formed by the colors of the **chromaKey** and **chromaRange** registers.

Each RGB color component of the **chromaKey** and **chromaRange** registers defines a chroma range for the color component The color component range includes the lower limit color from the **chromaKey** register and the upper limit color from the **chromaRange** register. Software must program the lower limits less-than or equal to the upper limits.

Each RGB color component **chromaRange** mode defines the color component range as inclusive or exclusive. Inclusive ranges prohibit colors within the range and exclusive ranges prohibit colors outside of the range.

Prohibited colors are blocked from the frame buffer based on the **chromaRange** mode. This mode may be set to “intersection” or “union”. The intersection mode blocks pixels prohibited by all of the color components and the union mode blocks pixels prohibited by any of the color components

Bit	Description
7:0	Chroma-Range Blue Upper Limit
15:8	Chroma-Range Green Upper Limit
23:16	Chroma-Range Red Upper Limit
24	Chroma-Range Blue Mode (0=inclusive; 1=exclusive)
25	Chroma-Range Green Mode (0=inclusive; 1=exclusive)
26	Chroma-Range Red Mode (0=inclusive; 1=exclusive)
27	Chroma-Range Block Mode (0=intersection; 1=union)
28	Chroma-Range Enable (0=disable; 1=enable)
31:29	reserved

8.32 userIntrCMD Register

Writing to the **userIntrCMD** register executes the USERINTERRUPT command:

Bit	Description
0	Wait for USERINTERRUPT to be cleared before continuing (1=stall graphics engine until interrupt is cleared)
1	Wait for interrupt generated by USERINTERRUPT (visible in **intrCtrl** bit(11)) to be cleared before continuing (1=stall graphics engine until interrupt is cleared)
9:2	User interrupt Tag

If the data written to **userIntrCMD** bit(0)=0, then a user interrupt is generated (**intrCtrl** bit(11) is set to 1). If the data written to **userIntrCMD** bit(1)=1, then the graphics engine stalls and waits for the

USERINTERRUPT interrupt to be cleared before continuing processing additional commands. If no USERINTERRUPT interrupt is set and the data written to **userIntrCMD** bit(1)=1, then the graphics engine will not stall and will continue to process additional commands. Software may also use combinations of intrCtrl bits(1:0) to generate different functionality.

The tag associated with a user interrupt is written to **userIntrCMD** bits 9:2. When a user interrupt is generated, the respective tag associated with the user interrupt is read from **IntrCtrl** bits 19:12.

If the USERINTERRUPT command does not stall the graphics engine (**userIntrCMD**(0)=1), then a potential race condition occurs between multiple USERINTERRUPT commands and software user interrupt processing. In particular, multiple USERINTERRUPT commands may be generated before software is able to process the first interrupt. Irrespective of how many user interrupts have been generated, the user interrupt tag field in **intrCtrl** (bits 19:12) always reflects the tag of last USERINTERRUPT command processed. As a result of this behavior, early tags from multple USERINTERRUPT commands may be lost. To avoid this behavior, software may force a single USERINTERRUPT command to be executed at a time by writing **userIntrCMD**(1:0)=0x3 and cause the graphics engine to stall until the USERINTERRUPT interrupt is cleared.

Note that bit 5 of **intrCtrl** must be set to 1 for user interrupts to be generated – writes to **userIntrCMD** when **intrCtrl**(5)=0 do not generate interrupts or cause the processing of commands to wait on clearing of the USERINTERRUPT command (regardless of the data written to **userIntrCMD**), and are thus in effect "dropped."

8.33 colBufferAddr

The **colBufferAddr** register defines the base address of the color buffer. The the address must be 16-byte aligned, so colBufferAddr[3:0] are unused.

Bit	Description
3:0	reserved
23:4	Color Buffer Base Address. Must be 16-byte aligned

*** Need to add address equation ***

8.34 colBufferStride

If the color buffer is linear (colBufferStride[15]=0) then colBufferStride[13:0] defines the linear stride of the color buffer in bytes. Linear stride must be 16-byte aligned. If the color buffer is tiled (colBufferStride[15]=1) then colBufferStride[6:0] defines the tile stride for the color buffer in tiles.

Bit	Description

13:0	if [15] = 0 then
	linear: [13:0] = linear stride in bytes
	else
	tiled: [6:0] = tile stride in tiles; [13:7] are reserved.

14	reserved
15	Memory type (0=linear; 1=tiled)

8.35 auxBufferAddr

The auxBufferAddr register defines the base address of the auxiliary buffer. The existence and enabling of the depth or the alpha auxiliary buffers is established within the **fbzMode** register. AuxBufferAddr must be 16 byte aligned, so auxBufferAddr[3:0] are unused.

Bit	Description

3:0	reserved
23:4	Auxiliary Buffer Base Address. Must be 16 byte aligned

*** Need to add address equation ***

8.36 auxBufferStride

If the aux buffer is linear (auxBufferStride[15]=0) then auxBufferStride[13:0] defines the linear stride of the aux buffer in bytes. Linear stride must be 16-byte aligned. If the aux buffer is tiled (auxBufferStride[15]=1) then auxBufferStride[6:0] defines the tile stride for the aux buffer in tiles.

Bit	Description

13:0	if [15] = 0 then
	linear: [13:0] = linear stride in bytes
	else
	tiled: [6:0] = tile stride in tiles; [13:7] are reserved.

14	reserved
15	Memory type (0=linear; 1=tiled)

8.37 clipLeftRight and clipLowYHighY Registers

The **clip** registers specify a rectangle within which all drawing operations are confined. If a pixel is to be drawn outside the clip rectangle, it will not be written into the RGB or depth buffers. Note that the specified clipping rectangle defines a valid drawing area in both the RGB and depth/alpha buffers. The values in the clipping registers are given in pixel units, and the valid drawing rectangle is inclusive of the **clipleft** and **clipLowY** register values, but exclusive of the **clipRight** and **clipHighY** register values. **clipLowY** must be less than **clipHighY**, and **clipLeft** must be less than **clipRight**. The **clip** registers can be enabled by setting bit(0) in the **fbzMode** register. When clipping is enabled, the bounding clipping rectangle must always be less than or equal to the screen resolution in order to clip to screen coordinates. Also note that if clipping is not enabled, rendering must not be specified to occur outside of the screen resolution.

Important Note: The **clipLowYHighY** register is defined such that y=0 always resides at the top of the monitor screen. Changing the value of the Y origin bits (**fbzMode** bit(17) or **lfbMode** bit(13)) has no affect on the **clipLowYHighY** register orientation. As a result, if the Y origin is defined to be at the bottom of the screen (by setting one of the Y origin bits), care must be taken in setting the **clipLowYHighY** register to ensure proper functionality. In the case where the Y origin is defined to be at the bottom of the screen, the value of **clipLowYHighY** is usually set as the number of scan lines in the monitor resolution minus the desired Y clipping values.

The **clip** registers are also used to define a rectangular region to be drawn during a FASTFILL command. Note that when **clipLowYHighY** is used to specify a rectangular region for the FASTFILL command, the orientation of the Y origin (top or bottom of the screen) is defined by the status of **fbzMode** bit(17). See section 7 and the **fastfillCMD** register description for more information on the FASTFILL command.

clipLeftRight Register

Bit	Description
11:0	Unsigned integer specifying right clipping rectangle edge
15:12	reserved
27:16	Unsigned integer specifying left clipping rectangle edge
31:28	reserved

clipLowYHighY Register

Bit	Description
11:0	Unsigned integer specifying high Y clipping rectangle edge
15:12	reserved
27:16	Unsigned integer specifying low Y clipping rectangle edge
31:28	reserved

8.38 fogTable Register

The **fogTable** register is used to implement fog functions in Avenger. The **fogTable** register is a 64-entry lookup table consisting of 8-bit fog blending factors and 8-bit Δfog blending values. The Δfog blending values are the difference between successive fog blending factors in **fogTable** and are used to blend between **fogTable** entries. Note that the Δfog blending factors are stored in 6.2 format, while the fog blending factors are stored in 8.0 format. For most applications, the 6.2 format Δfog blending factors will have the two LSBs set to 0x0, with the six MSBs representing the difference between successive fog blending factors. Also note that as a result of the 6.2 format for the Δfog blending factors, the difference between successive fog blending factors cannot exceed 63. When storing the fog blending factors, the sum of each fog blending factor and Δfog blending factor pair must not exceed 255. When loading **fogTable**, two fog table entries must be written concurrently in a 32-bit word. A total of 32 32-bit PCI writes are required to load the entire **fogTable** register.

fogTable[n] ($0 \le n \le 31$)

Bit	Description
7:0	FogTable[2n] ΔFog blending factor
15:8	FogTable[2n] Fog blending factor
23:16	FogTable[2n+1] ΔFog blending factor
31:24	FogTable[2n+1] Fog blending factor

8.39 fbiPixelsIn Register

The **fbiPixelsIn** register is a 24-bit counter which is incremented for each pixel processed by the Avenger triangle walking engine. **fbiPixelsIn** is incremented irrespective if the triangle pixel is actually drawn or not as a result of the depth test, alpha test, etc. **fbiPixelsIn** is used primarily for statistical information, and in essence allows software to count the number of pixels in a screen-space triangle. **fbiPixelsIn** is reset to 0x0 on power-up reset, and is reset when a '1' if written to the lsb of **nopCMD**.

Bit	Description
23:0	Pixel Counter (number of pixels processed by Avenger triangle engine)

8.40 fbiChromaFail Register

The **fbiChromaFail** register is a 24-bit counter which is incremented each time an incoming source pixel (either from the triangle engine or linear frame buffer writes through the pixel pipeline) is invalidated in the pixel pipeline because of the chroma-key color match test. If an incoming source pixel color matches the **chomaKey** register, **fbiChromaFail** is incremented. **fbiChromaFail** is reset to 0x0 on power-up reset, and is reset when a '1' if written to the lsb of **nopCMD**.

Bit	Description
23:0	Pixel Counter (number of pixels failed chroma-key test)

8.41 fbiZfuncFail Register

The **fbiZfuncFail** register is a 24-bit counter which is incremented each time an incoming source pixel (either from the triangle engine or linear frame buffer writes through the pixel pipeline) is invalidated in the pixel pipeline because of a failure in the Z test. The Z test is defined and enabled in the **fbzMode** register. **fbiZfuncFail** is reset to 0x0 on power-up reset, and is reset when a '1' if written to the lsb of **nopCMD**.

Bit	Description
23:0	Pixel Counter (number of pixels failed Z test)

8.42 fbiAfuncFail Register

The **fbiAfuncFail** register is a 24-bit counter which is incremented each time an incoming source pixel (either from the triangle engine or linear frame buffer writes through the pixel pipeline) is invalidated in the pixel pipeline because of a failure in the alpha test. The alpha test is defined and enabled in the **alphaMode** register. The **fbiAfuncFail** register is also incremented if an incoming source pixel is invalidated in the pixel pipeline as a result of the alpha masking test (bit(13) in **fbzMode**). **fbiAfuncFail** is reset to 0x0 on power-up reset, and is reset when a '1' if written to the lsb of **nopCMD**.

Bit	Description
23:0	Pixel Counter (number of pixels failed Alpha test)

8.43 fbiPixelsOut Register

The **fbiPixelsOut** register is a 24-bit counter which is incremented each time a pixel is written into a color buffer during rendering operations (rendering operations include triangle commands, linear frame buffer writes, and the FASTFILL command). Pixels tracked by **fbiPixelsOut** are therefore subject to the chroma-test, Z test, Alpha test, etc. that are part of the regular Avenger pixel pipeline. **fbiPixelsOut** is used to count the number of pixels actually drawn (as opposed to the number of pixels processed counted by **fbiPixelsIn**). Note that the RGB mask (**fbzMode** bit(9) is ignored when determining **fbiPixelsOut**. **fbiPixelsOut** is reset to 0x0 on power-up reset, and is reset when a '1' if written to the lsb of **nopCMD**.

Bit	Description
23:0	Pixel Counter (number of pixels drawn to color buffer)

8.44 clipLeftRight1, clipTopBottom1 Registers

The clip0 and clip1 registers specify two rectangular regions which restrict drawing operation. The secondary clip rectangles may be defined as inclusive or exclusive through the clipMode field of the clipTopBottom register. An inclusive rectangle allows drawing within the rectangle and an exclusive rectangle disallows drawing within the rectangle. Drawing within an excluded region of either of the clip rectangles circumvents the write of pixels into both the color and auxiliary buffers.

The clip registers define the four corners of a rectangular region in window relative pixel coordinates (native x/y rendering coordinates). The value of clipTop must be less than clipBottom and the value of clipLeft must be less than clipRight. This programming results in a rectangular region including the clipLeft and clipTop register values, but excluding the clipRight and clipBottom register values.

ClipLeftRight1 Register

Bit	Description
11:0	Unsigned integer specifying right clipping rectangle edge
15:12	reserved
27:16	Unsigned integer specifying left clipping rectangle edge
30:28	reserved
31	Clip Enable (0=disable, 1=enable

ClipLeftRight1 Register

Bit	Description
11:0	Unsigned integer specifying top clipping rectangle edge
15:12	reserved
27:16	Unsigned integer specifying bottom clipping rectangle edge
30:28	reserved
31	Clip Mode (0=inclusive, 1=exclusive

8.45 swapBufferPend Register

Writes to the **swapBufferPend** register increments the swap buffer pending count of the Avenger status register. Writes take effect immediately and are available only through direct access.

8.46 leftOverlayBuf Register

Starting address of left or Monocular buffer address for overlay display. For video overlay, the start address needs to be aligned on a 32-bit boundary for YUV 422 pixel format and a 64-bit boundary for YUV 411 pixel format. This register is sampled at the end of vertical retrace.

Bit	Description
23:0	Starting address of the overlay surface buffer 0. The address is the physical address.
30:24	Reserved
31	Bit[31] indicates if the buffer contains even or odd field in case of backend (Bob) deinterlacing. Bit[31] = 1 for even field; Bit[31] = 0 for odd field.

8.47 RightOverlayBuf Register

Starting address of right buffer address for overlay display. For video overlay, the start address needs to be aligned on a 32-bit boundary for YUV 422 pixel format and a 64-bit boundary for YUV 411 pixel format. This register is only used for stereo buffering. This register is sampled at the end of vertical retrace.

Bit	Description
23:0	Starting address of the overlay surface buffer 0. The address is the physical address.
31:24	Reserved

8.48 fbiSwapHistory Register

The **fbiSwapHistory** register keeps track of the number of vertical syncs which occur between executed swap commands. **fbiSwapHistory** logs this information for the last 8 executed swap commands. Upon completion of a swap command, **fbiSwapHistory** bits (27:0) are shifted left by four bits to form the new **fbiSwapHistory** bits (31:4), which maintains a history of the number of vertical syncs between execution of each swap command for the last 7 frames. Then, **fbiSwapHistory** bits(3:0) are updated with the number of vertical syncs which occurred between the last swap command and the just completed swap command or the value 0xf, whichever is less.

Bit	Description
3:0	Number of vertical syncs between the second most recently completed swap command and the most recently completed swap command, or the value 0xf, whichever is less for Frame N.
7:4	Vertical sync swapbuffer history for Frame N-1
11:8	Vertical sync swapbuffer history for Frame N-2
15:12	Vertical sync swapbuffer history for Frame N-3
19:16	Vertical sync swapbuffer history for Frame N-4
23:20	Vertical sync swapbuffer history for Frame N-5
27:24	Vertical sync swapbuffer history for Frame N-6
31:28	Vertical sync swapbuffer history for Frame N-7

8.49 fbiTrianglesOut Register

The **fbiTriangles** register is a 24-bit counter which is incremented for each triangle processed by the Avenger triangle walking engine. Triangles which are backface culled in the triangle setup unit do not increment **fbiTrianglesOut**. **fbiTrianglesOut** is reset to 0x0 on power-up reset, and is also reset to 0x0 when a '1' is written to **nopCMD** bit(1).

Bit	Description
23:0	Rendered triangles (total number of triangles rendered by Avenger triangle rendering engine)

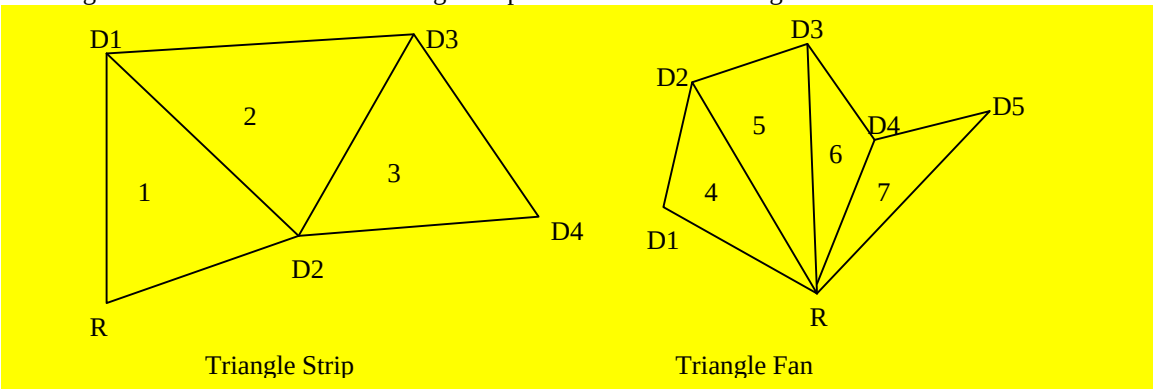
8.50 sSetupMode Register

The sSetupMode register provides a way for the CPU to only setup required parameters. When a Bit is set, that parameter will be calculated in the setup process, otherwise the value is not passed down to the triangle, and the previous value will be used. Also the definition of the triangle strip is defined in bits 21:16, where bit 16 defines fan. Culling is enabled by seting bit 17 to a value of "1", whereas bit 18 defines the culling sign. Bit 19 disables the ping pong sign inversion that happens during triangle strips.

Bit	Description
0	Setup Red, Green, and Blue
1	Setup Alpha
2	Setup Z
3	Setup Wb
4	Setup W0
5	Setup S0 and T0
6	Setup W1
7	Setup S1 and T1
15:8	reserved
16	Strip mode (0=strip, 1=fan)
17	Enable Culling (0=disable, 1=enable)
18	Culling Sign (0=positive sign, 1=negative sign)
19	Disable ping pong sign correction during triangle strips (0=normal, 1=disable)

8.51 Triangle Setup Vertex Registers

The sVx, sVy registers specify the x and y coordinates of a triangle strip to be rendered. A triangle strip, once the initial triangle has been defined, only requires a new X and Y to render consecutive triangles. The diagram below illustrates how triangle strips are sent over to Avenger.



Triangle strips and triangle fans are implemented in Avenger by common vertex information and 2 triangle commands. Vertex information is written to Avenger for a current vertex and are followed by a write to either the sBeginTriCMD or the sDrawTriCMD . For example, to render the triangle strip in the above figure, parameters X, Y, ARGB, W0, S/W, T/W for vertex R would be written followed by a write to sBeginTriCMD. Vertex D1's parameters would next be written followed by a write to the sDrawTriCMD. After D2's data has been sent, and the 2nd write to sDrawTriCMD has been completed Avenger will begin to render triangle 1. As triangle 1 is being rendered, data for vertex D3 will be sent down followed by another write to sDrawTriCMD, thus launching another triangle. Triangle fans are very similar to triangle strips. Instead of changing all three vertices, only the last 2 get modified. Triangle fans start with a sBeginTriCMD just as the triangle strip did, and send down sDrawTriCMD for every new vertex. To select triangle fan or triangle strip, you must write bit 0 of the triangle setup mode register.

SVx Register

Bit	Description
31:0	Vertex coordinate information (IEEE 32 bit single-precision floating point format)

sVy Register

Bit	Description
31:0	Vertex coordinate information (IEEE 32 bit single-precision floating point format)

8.52 sARGB Register

The ARGB register specify the color at the current vertex in a packed 32 bit value.

Bit	Description
31:24	Alpha Color
23:16	Red Color
15:8	Green Color
7:0	Blue Color

8.53 sRed Register

the sRed register is the separated red value for the current vertex.

Bit	Description
31:0	Red value at vertex (0.0 - 255.0). (IEEE 32 bit single-precision floating point format)

8.54 sGreen Register

The sGreen register is the separated green value for the current vertex.

Bit	Description
31:0	Green value at vertex (0.0 - 255.0). (IEEE 32 bit single-precision floating point format)

8.55 sBlue Register

The sBlue register is the separated blue value for the current vertex.

Bit	Description
31:0	Blue value at vertex (0.0 - 255.0). (IEEE 32 bit single-precision floating point format)

8.56 sAlpha Register

the sAlpha register is the separated alpha value for the current vertex.

Bit	Description
31:0	Alpha value at vertex (0.0 - 255.0). (IEEE 32 bit single-precision floating point format)

8.57 sVz Register

The Vz register is the Z value at the current vertex.

Bit	Description
31:0	Vertex coordinate information (IEEE 32 bit single-precision floating point format)

8.58 sWb Register

The Wb register is a global 1/W that is sent to both the FBI and all TMUs.

Bit	Description
31:0	Global 1/W. (IEEE 32 bit single-precision floating point format).

8.59 sWtmu0 Register

The sWtmu0 register is all the TMUs local 1/W value for the current vertex.

Bit	Description

31:0	Texture local 1/W. (IEEE 32 bit single-precision floating point format)

8.60 sS/W0 Register

The S/W0 register is the S coordinate of the current vertex divided by W, for all TMUs.

Bit	Description
31:0	Texture S coordinate (IEEE 32 bit single-precision floating point format)

8.61 sT/W0 Register

The T/W register s the T coordinate of the current vertex divided by W, for all TMUs.

Bit	Description
31:0	Texture T coordinate (IEEE 32 bit single-precision floating point format)

8.62 sWtmu1 Register

The sWtmu1 register is TMU1's local 1/W value for the current vertex.

Bit	Description
31:0	Texture local 1/W. (IEEE 32 bit single-precision floating point format)

8.63 sS/Wtmu1 Register

The sS/Wtmu1 register is TMU1's local S/W value for the current vertex.

Bit	Description
31:0	Texture local 1/W. (IEEE 32 bit single-precision floating point format)

8.64 sT/Wtmu1 Register

The sT/Wtmu1 register is TMU1's local T/W value for the current vertex.

Bit	Description
31:0	Texture local 1/W. (IEEE 32 bit single-precision floating point format)

8.65 sDrawTriCMD Register

The DrawTriCMD registers starts the draw process.

Bit	Description
0	Draw triangle

8.66 sBeginTriCMD Register

A write to this register begins a new triangle strip starting with the current vertex. No actual drawing is performed.

Bit	Description
0	Begin New triangle

The Folowing two figures are sample pseudo code for generating triangle strips and fans.

Setup Code

```
// packed color triangle strip setup.
write (sst->sSetupMode, PACKEDCOLOR | SETUP_XY | SETUP_RGB | SETUP_ALPHA | SETUP_ST);

// Begin triangle setup
// Vertex #0
write (sst->sVx, -30.0);
write (sst->sVy, 15.0);
write (sst->sARGB, 0xFF010203); // Color
write (sst->sSw, 4.0);
write (sst->sTw, 2.0);
write (sst->sBegintriCMD, 0);          // Begin Triangle

// vertex #1
write (sst->sVx, 5.0);
write (sst->sVy, 10.0);
write (sst->sARGB, 0x00052377);
write (sst->sSw, 30.0);
write (sst->sTw, 60.0);
write (sst->sDrawtriCMD, 0);

// Vertex #2
write (sst->sVx, 50.0);
write (sst->sVy, 100.0);
write (sst->sARGB, 0x12345678);
write (sst->sSw, 100.0);
write (sst->sTw, 200.0);
write (sst->sDrawtriCMD, 0);           // Draw first triangle

// Vertex #3
write (sst->sVx, 50.0);
write (sst->sVy, 0.0);
write (sst->sARGB, 0x87654321);
write (sst->sSw, 0.0);
write (sst->sTw, 200.0);
write (sst->sDrawtriCMD, 0);           // Draw second triangle

// Vertex #4
write (sst->sVx, 100.0);
write (sst->sVy, 100.0);
write (sst->sARGB, 0x0);
write (sst->sSw, 200.0);
write (sst->sTw, 150.0);
write (sst->sDrawtriCMD, 0);           // Draw second triangle
```

```
// Separate Color triangle fan setup
write (sst->sSetupMode, FANMODE | SETUP_XY | SETUP_RGB);

// Vertex #0
write (sst->sVx, -30.0);
write (sst->sVy, 15.0);
write (sst->sRed, 0.0);
write (sst->sGreen, 0.0);
write (sst->sBlue, 0.0);
write (sst->sBegintriCMD, 0);          // Begin Triangle

// vertex #1
write (sst->sVx, 5.0);
write (sst->sVy, 10.0);
write (sst->sRed, 255.0);
write (sst->sGreen, 0.0);
write (sst->sBlue, 0.0);
write (sst->sDrawTriCMD, 0);

// Vertex #2
write (sst->sVx, 50.0);
write (sst->sVy, 100.0);
write (sst->sRed, 0.0);
write (sst->sGreen, 255.0);
write (sst->sBlue, 0.0);
write (sst->sDrawTriCMD, 0);           // Draw first triangle

// Vertex #3
write (sst->sVx, 50.0);
write (sst->sVy, 0.0);
write (sst->sRed, 0.0);
write (sst->sGreen, 0.0);
write (sst->sBlue, 255.0);
write (sst->sDrawTriCMD, 0);           // Draw second triangle

// Vertex #4
write (sst->sVx, 100.0);
write (sst->sVy, 100.0);
write (sst->sRed, 255.0);
write (sst->sGreen, 255.0);
write (sst->sBlue, 0.0);
write (sst->sDrawTriCMD, 0);           // Draw second triangle
```

8.67 textureMode Register

The **textureMode** register controls texture mapping functionality including perspective correction, texture filtering, texture clamping, and multiple texture blending.

Bit	Name	Description
0	*tpersp_st*	Enable perspective correction for S and T iterators (0=linear interploation of S,T, force W to 1.0, 1=perspective correct, S/W, T/W)

1	*tminfilter*	Texture minification filter (0=point-sampled, 1=bilinear)
2	*tmagfilter*	Texture magnification filter (0=point-sampled, 1=bilinear)
3	*tclampw*	Clamp when W is negative (0=disabled, 1=force S=0, T=0 when W is negative)
4	*tloddither*	Enable Level-of-Detail dithering (0=no dither, 1=dither)
5	*tnccselect*	Narrow Channel Compressed (NCC) Table Select (0=table 0, 1=table 1)
6	*tclamps*	Clamp S Iterator (0=wrap, 1=clamp)
7	*tclampt*	Clamp T Iterator (0=wrap, 1=clamp)
11:8	*tformat*	Texture format (see table below)
		Texture Color Combine Unit control (RGB):
12	*tc_zero_other*	Zero Other (0=c_other, 1=zero)
13	*tc_sub_clocal*	Subtract Color Local (0=zero, 1=c_local)
16:14	*tc_mselect*	Mux Select (0=zero, 1=c_local, 2=a_other, 3=a_local, 4=LOD, 5=LOD_frac, 6-7=reserved)
17	*tc_reverse_blend*	Reverse Blend (0=normal blend, 1=reverse blend)
18	*tc_add_clocal*	Add Color Local
19	*tc_add_alocal*	Add Alpha Local
20	*tc_invert_output*	Invert Output
		Texture Alpha Combine Unit control (A):
21	*tca_zero_other*	Zero Other (0=c_other, 1=zero)
22	*tca_sub_clocal*	Subtract Color Local (0=zero, 1=c_local)
25:23	*tca_mselect*	Mux Select (0=zero, 1=c_local, 2=a_other, 3=a_local, 4=LOD, 5=LOD_frac, 6-7=reserved)
26	*tca_reverse_blend*	Reverse Blend (0=normal blend, 1=reverse blend)
27	*tca_add_clocal*	Add Color Local
28	*tca_add_alocal*	Add Alpha Local
29	*tca_invert_output*	Invert Output
30	*trilinear*	Enable trilinear texture mapping (0=point-sampled/bilinear, 1=trilinear)

tpersp_st bit of **textureMode** enables perspective correction for S and T iterators. Note that there is no performance penalty for performing perspective corrected texture mapping.

tminfilter, tmagfilter bits of **textureMode** specify the filtering operation to be performed. When point sampled filtering is selected, the texel specified by <s,t> is read from texture memory. When bilinear filtering is selected, the four closet texels to a given <s,t> are read from memory and blended together as a function of the fractional components of <s,t>. *tminfilter* is referenced when LOD>=LODmin, otherwise *tmagfilter* is referenced.

tclampw bit of **textureMode** is used when projecting textures to avoid projecting behind the source of the projection. If this bit is set, S, T are each forced to zero when W is negative. Though usually desireable, it is not necessary to set this bit when doing projected textures.

tloddither bit of **textureMode** enables Level-of-Detail (LOD) dither. Dithering the LOD calculation is useful when performing texture mipmapping to remove the LOD bands which can occur from with mipmapping without trilinear filtering. This adds an average of 3/8 (.375) to the LOD value and needs to compensated in the amount of *lodbias*.

tnccselect bit of **textureMode** selects the NCC lookup table to be used when decompressing 8-bit NCC textures.

tclamps, tclampt bits of **textureMode** enable clamping of the S and T texture iterators. When clamping is enabled, the S iterator is clamped to [0, texture width) and the T iterator is clamped to [0, texture height). When clamping is disabled, S coordinates outside of [0, texture width) are allowed to wrap into the [0, texture width) range using bit truncation. Similarly when clamping is disabled, T coordinates outside of [0, texture height) are allowed to wrap into the [0, texture height) range using bit truncation.

tformat field of **textureMode** specifies the texture format accessed by TREX. Note that the texture format field is used for both reading and writing of texture memory. The following table shows the texture formats and how the texture data is expanded into 32-bit ARGB color:

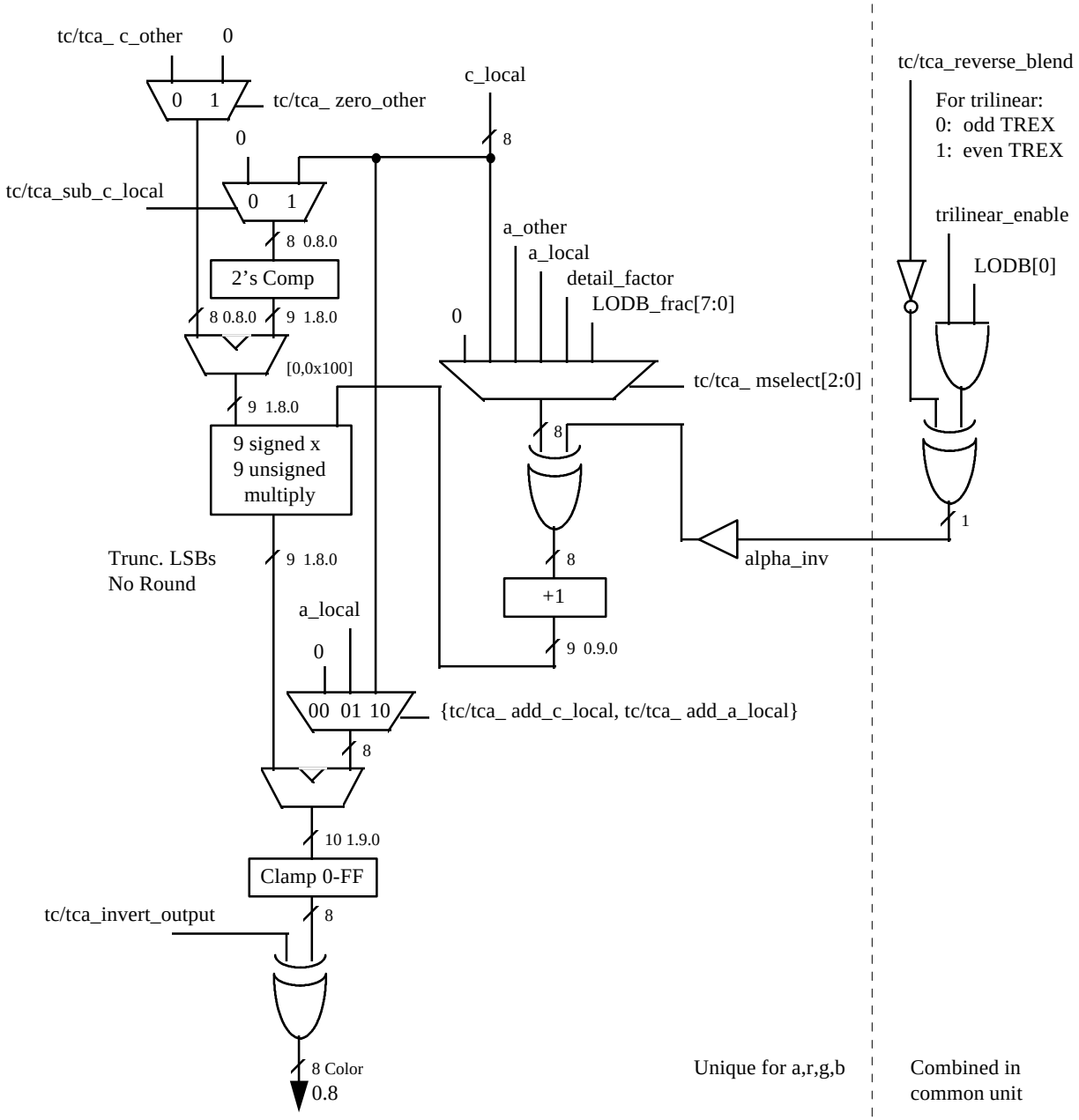
***tformat* Value**	**Texture format**	**8-bit Alpha**	**8-bit Red**	**8-bit Green**	**8-bit Blue**
0	8-bit RGB (3-3-2)	0xff	{r[2:0],r[2:0],r[2:1]}	{g[2:0],g[2:0],g[2:1]}	{b[1:0],b[1:0],b[1:0],b[1:0]}
1	8-bit YIQ (4-2-2)	See below			
2	8-bit Alpha	a[7:0]	a[7:0]	a[7:0]	a[7:0]
3	8-bit Intensity	0xff	i [7:0]	i[7:0]	i[7:0]
4	8-bit Alpha, Intensity (4-4)	{a[3:0],a[3:0]}	{i[3:0],i[3:0]}	{i[3:0],i[3:0]}	{i[3:0],i[3:0]}
5	8-bit Palette to RGB	0xff	palette r[7:0]	palette g[7:0]	palette b[7:0]
6	8 bit Palette to RGBA	{palette_r[7:2], palette_r[7:6]	{palette_r[1:0], palette_g[7:4], palette_r[1:0]}	{palette_g[3:0], palette_b[7:6], palette_g[3:2]}	{palette_b[5:0], palette_b[5:4]}
7	Reserved				
8	16-bit ARGB (8-3-3-2)	a[7:0]	{r[2:0],r[2:0],r[2:1]}	{g[2:0],g[2:0],g[2:1]}	{b[1:0],b[1:0],b[1:0],b[1:0]}
9	16-bit AYIQ (8-4-2-2)	See below			
10	16-bit RGB (5-6-5)	0xff	{r[4:0],r[4:2]}	{g[5:0],r[5:4]}	{b[4:0],b[4:2]}
11	16-bit ARGB (1-5-5-5)	{a[0],a[0],a[0],a[0], a[0],a[0],a[0],a[0]}	{r[4:0],r[4:2]}	{g[4:0],g[4:2]}	{b[4:0],b[4:2]}
12	16-bit ARGB (4-4-4-4)	{a[3:0},a[3:0]}	{r[3:0},r[3:0]}	{g[3:0},g[3:0]}	{b[3:0},b[3:0]}
13	16-bit Alpha, Intensity (8-8)	a[7:0]	i[7:0]	i[7:0]	i[7:0]
14	16-bit Alpha, Palette (8-8)	a[7:0]	palette r[7:0]	palette g[7:0]	palette b[7:0]
15	Reserved				

where a, r, g, b, and i(intensity) represent the actual values read from texture memory. The following table shows how 32-bit RGBA texture information is derived from the YIQ texture formats. This is detailed later in the nccTable description.

Texture format	**8-bit Alpha**	**8-bit Red**	**8-bit Green**	**8-bit Blue**
8-bit YIQ (4-2-2)	0xff	ncc _red[7:0]	ncc _green[7:0]	ncc _blue[7:0]
16-bit AYIQ (8-4-2-2)	a[7:0]	ncc _red[7:0]	ncc _green[7:0]	ncc _blue[7:0]

There are three Texture Color Combine Units (RGB) and one Texture Alpha Combine Unit(A), all four are identical, except for the bit fields that control them. The *tc_** fields of **textureMode** control the Texture Color Combine Units; the *tca_** fields control the Texture Alpha Combine Units. The diagram below illustrates the Texture Color Combine Unit/Texture Alpha Combine Unit:

Blend with Incoming Color



tc_ prefix applies to R,G and B channels. tca_ prefix applies to A channel.

8.68 tLOD Register

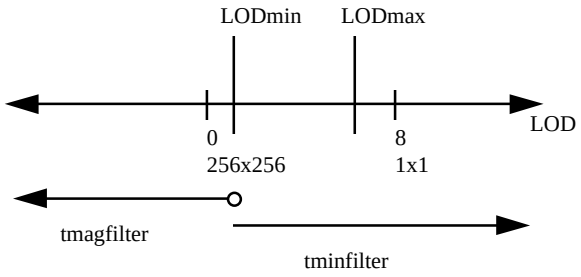
The **tLOD** register controls the texture mapping LOD calculations.

Bit	Name	Description
5:0	*lodmin*	Minimum LOD. (4.2 unsigned)

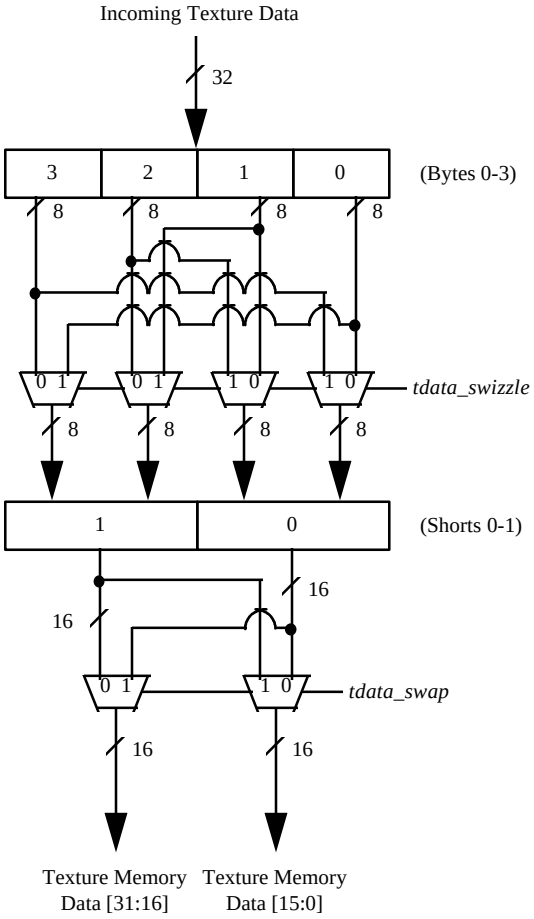
11:6	*lodmax*	Maximum LOD. (4.2 unsigned)
17:12	*lodbias*	LOD Bias. (4.2 signed)
18	*lod_odd*	LOD odd (0=even, 1=odd)
19	*lod_tsplit*	Texture is Split. (0=texture contains all LOD levels, 1=odd or even levels only, as controlled by *lod_odd*)
20	*lod_s_is_wider*	S dimension is wider, for rectilinear texture maps. This is a *don't care* for square textures. (1=S is wider than T).
22:21	*lod_aspect*	Aspect ratio. Equal to 2^n. (00 is square texture, others are rectilinear: 01 is 2x1/1x2, 10 is 4x1/1x4, 10 is 8x1/1x8)
23	*lod_zerofrac*	LOD zero frac, useful for bilinear when even and odd levels are split across two TREXs (0=normal LOD frac, 1=force fraction to 0)
24	*tmultibaseaddr*	Use multiple texbaseAddr registers
25	*tdata_swizzle*	Byte swap incoming texture data (bytes 0<->3, 1<->2).
26	*tdata_swap*	Short swap incoming texture data (shorts 0<->1).
27	*reserved*	used to be tdirect_write in Voodoo graphics.
28	*tmirrors*	Mirror texture in S dimension
29	*tmirrort*	Mirror texture in T dimension

lodbias is added to the calculated LOD value, then it is clamped to the range [*lodmin*, min(8.0, *lodmax*)]. Note that whether the LOD is clamped to *lodmin* is used to determine whether to use the minification or magnification filter, selected by the *tminfilter* and *tmagfilter* bits of **textureMode**:

LOD bias, clamp



The *tdata_swizzle* and *tdata_swap* bits in **tLOD** are used to modify incoming texture data for endian dependencies. The *tdata_swizzle* bit causes incoming texture data bytes to be byte order reversed, such that bits(31:24) are swapped with bits(7:0), and bits(23:16) are swapped with bits(15:8). Short-word swapping is performed after byte order swizzling, and is selected by the *tdata_swap* bit in **tLOD**. When enabled, short-word swapping causes the post-swizzled 16-bit shorts to be order reversed, such that bits(31:16) are swapped with bits(15:0). The following diagram shows the data manipulation functions perfomed by the *tdata_swizzle* and *tdata_swap* bits:



8.69 tDetail Register

The **tDetail** register controls the detail texture.

Bit	Name	Description
7:0	*detail _max*	Detail texture LOD clamp (8.0 unsigned)
13:8	*detail_bias*	Detail texture bias (6.0 signed)
16:14	*detail_scale*	Detail texture scale shift left
17	*rgb_tminfilter*	RGB texture minification filter(0 = point-sampled, 1 = bilinear)
18	*rgb_tmagfilter*	RGB texture magnification filter(0 = point-sampled, 1 = bilinear)
19	*a_tminfilter*	Alpha texture minification filter(0 = point-sampled, 1 = bilinear)
20	*a_tmagfilter*	Alpha texture magnification filter(0 = point-sampled, 1 = bilinear)
21	*rgb_a_separate_filter*	0 = tminfilter and tmagfilter of textureMode define the filter for RGBA 1 = rgb_tminfilter and rgb_tmagfilter define the filter for RGB, a_tminfilter and a_tmagfilter define the filter for alpha.

detail_factor is used in the Texture Combine Unit to blend between the main texture and the detail texture.
detail_factor (0.8 unsigned) = max(*detail_max*, ((*detail_bias* - LOD) << *detail_scale*)).

When *rgb_a_separate_filter* is set, *rgb_tminfilter* and *rgb_tmagfilter* are used for RGB filtering and *a_tminfilter* and *a_tmagfilter* are used for A filtering.

8.70 texBaseAddr, texBaseAddr1, texBaseAddr2, and texBaseAddr38 Registers

The **texBaseAddr** register specifies the starting texture memory address for accessing a texture. It is used for both rendering and texture downloading. Calculation of the **texBaseAddr** is described in the **Texture Memory Access** section. Selection of the base address is a function of *tmultibaseaddr* and LODBI.

texBaseAddr[23:4] indicates the base address of the texture in 16-bytes units. If the texture is tiled (**texBaseAddr**[0]=1), then **texBaseAddr**[31:25] indicate the tile stride.

texBaseAddr

Bit	Name	Description
0	***texmemtype***	Texture Memory type (0=linear, 1=tiled)
3:1	*reserved*	
23:4	***texbaseaddr***	Texture Memory Base Address, in 16-byte units, *tmultibaseaddr*==0 or LODBI==0
24	*reserved*	
31:25	***texstride***	Tile stride (0 to 127 tiles).

texBaseAddr1, texBaseAddr2, texBaseAddr38 indicate the base addresses of lods 1, 2 and 3-8 in 16 byte units, if *tmultibaseaddr*=1.

texBaseAddr1, texBaseAddr2, texBaseAddr38

Bit	Name	Description
23:4	*texbaseaddr1*	Texture Memory Base Address, *tmultibaseaddr*==1 and LODBI==1
23:4	*texbaseaddr2*	Texture Memory Base Address, *tmultibaseaddr*==1 and LODBI==2
23:4	*texbaseaddr38*	Texture Memory Base Address, *tmultibaseaddr*==1 and LODBI>=3

.

8.71 trexInit1 Register

The **trexInit1** register is used for hardware initialization and configuration of the TREX portion of H3.

Bit	Name	Description
0	*rsv_sl_int_slave*	reserved
1	*rsv_sl_int_en*	reserved
6:2	*ft_FIFO_sil*	FBI-to-TREX interface FIFO stall input level. Free space level at which stall signal is sent back to transmitting chip.
10:7	*tt_FIFO_sil*	TREX-to-TREX interface FIFO stall input level. Free space level at which stall signal is sent back to transmitting chip.
11	*reserved*	
15:12	*tf_ck_del_adj*	TREX-to-FBI interface clock delay adjust. Adjusts phase of the transmit clock.
16	*rg_ttcii_inh*	Register ttcii inhibit. when *use_rg_ttcii_inh*==1. 0=expect data from upstream TREX, 1=ignore data from upstream TREX.
17	*use_rg_ttcii_inh*	Use register ttcii inhibit to chose if data is expected from upstream TREX. 0=use clock sense result, 1=ignore clock sense result and use *rg_ttcii_inh*.
18	*send_config*	Send config. Transmit configuration to FBI through the tf_ interface instead of texel data. 0=normal, 1=send.
19	*reset_FIFOs*	Reset all of the FIFO's inside TREX. 0=run, 1=assert the reset signal.
20	*reset_graphics*	Reset all of the graphics inside TREX. 0=run, 1=assert the reset signal.
22:21	*rsv_palette_del*	reserved

25:23	*send_config_sel*	Send config select. (not revision 0) Selects which data to transmit to FBI when *send_config*==1. 000=reserved 001=reserved 010=reserved 011=trexInit1, 100=texBaseAddr[31:0], (for this function, 32 bits are retained and is non-maskable) 101,110,111=reserved.
26	*use_4bit_st_frac*	1=use 4 bits for s,t instead of 8. Default = 0.
27	*a_attr_set_only*	1=use only the A set of triangle attributes. Default = 0.
28	*nop_per_tri*	1=insert a nop per triangle. Default = 0.
29	*always_cache_inv*	1=always cache invalidate each triangle. Default = 0.
30	*always_4texel_needed*	1=always indicate that 4 texels are needed for each pixel. Default = 0.

send_config

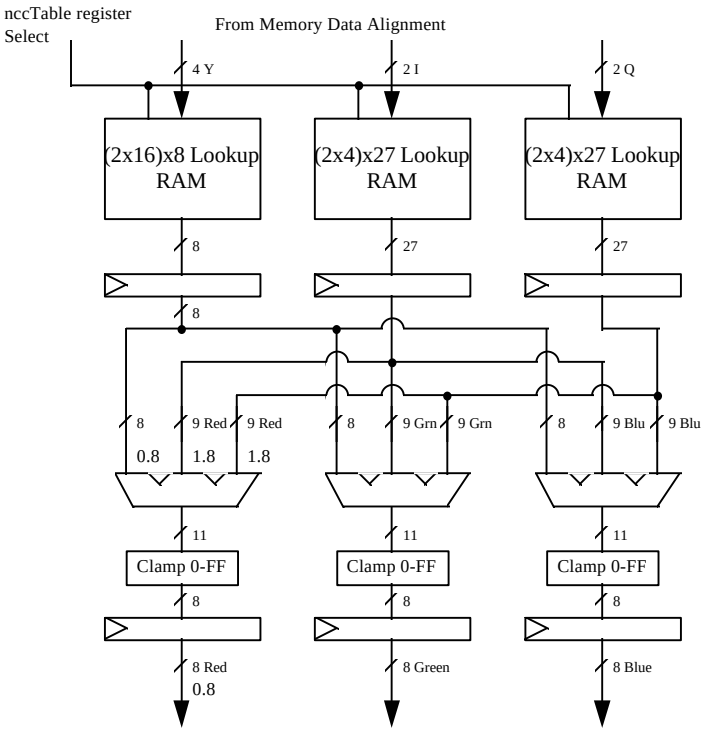
It is possible to read trexInit1 and texBaseAddr through the the send_config path, which sends these registers over to the FBI section of H3 via the graphics tf bus. When send_config = 1, tf_data[31:0] = {a[7:0], r[7:0], g[7:0], b[7:0]}. TREX's TC/TCA must be set to pass c_other.

8.72 nccTable0 and nccTable1 Registers

The **nccTable0** and **nccTable1** registers contain two Narrow Channel Compression (NCC) tables used to store lookup values for compressed textures (used in YIQ and AYIQ texture formats as specified in *tformat* of **textureMode**). Two tables are stored so that they can be swapped on a per-triangle basis when performing multi-pass rendering, thus avoiding a new download of the table. Use of either **nccTable0** or **nccTable1** is selected by the Narrow Channel Compressed (NCC) Table Select bit of **textureMode**. **nccTable0** and **nccTable1** are stored in the format of the table below, and are write only.

nccTable Address	Bits	Contents
0	31:0	{Y3[7:0], Y2[7:0], Y1[7:0], Y0[7:0]}
1	31:0	{Y7[7:0], Y6[7:0], Y5[7:0], Y4[7:0]}
2	31:0	{Yb[7:0], Ya[7:0], Y9[7:0], Y8[7:0]}
3	31:0	{Yf[7:0], Ye[7:0], Yd[7:0], Yc[7:0]}
4	26:0	{I0_r[8:0], I0_g[8:0], I0_b[8:0]}
5	26:0	{I1_r[8:0], I1_g[8:0], I1_b[8:0]}
6	26:0	{I2_r[8:0], I2_g[8:0], I2_b[8:0]}
7	26:0	{I3_r[8:0], I3_g[8:0], I3_b[8:0]}
8	26:0	{Q0_r[8:0], Q0_g[8:0], Q0_b[8:0]}
9	26:0	{Q1_r[8:0], Q1_g[8:0], Q1_b[8:0]}
10	26:0	{Q2_r[8:0], Q2_g[8:0], Q2_b[8:0]}
11	26:0	{Q3_r[8:0], Q3_g[8:0], Q3_b[8:0]}

The following figure illustrates how compressed textures are decompressed using the NCC tables:



8.73 8-bit Palette

The 8-bit palette is used for 8-bit P and 16-bit AP modes. The palette is loaded with register writes. During rendering, four texels are looked up simultaneously, each an independent 8-bit address.

Palette Write

The palette is written through the NCC table 0 I and Q register space when the MSB of the register write data is set. The NCC tables are not written when the I or Q NCC table register space is addressed and MSB of the register write data is set to 1 – Instead the data is stored in the texture palette.

Palette Load Mechanism

Register Address	LSB of P	Register Write Data (31 … 0)				
nccTable0 I0	P[0]=0	1	P[7:1]	R[7:0]	G[7:0]	B[7:0]
nccTable0 I1	P[0]=1	1	P[7:1]	R[7:0]	G[7:0]	B[7:0]
nccTable0 I2	P[0]=0	1	P[7:1]	R[7:0]	G[7:0]	B[7:0]
nccTable0 I3	P[0]=1	1	P[7:1]	R[7:0]	G[7:0]	B[7:0]
nccTable0 Q0	P[0]=0	1	P[7:1]	R[7:0]	G[7:0]	B[7:0]
nccTable0 Q1	P[0]=1	1	P[7:1]	R[7:0]	G[7:0]	B[7:0]
nccTable0 Q2	P[0]=0	1	P[7:1]	R[7:0]	G[7:0]	B[7:0]
nccTable0 Q3	P[0]=1	1	P[7:1]	R[7:0]	G[7:0]	B[7:0]

Note that the even addresses alias to the same location, as well as the odd ones. It is recommended that the table be written as 32 sets of 8 so that PCI bursts can be 8 transfers long.

8.74 Command Descriptions

8.74.1 NOP Command

The NOP command is used to flush the graphics pipeline. When a NOP command is executed, all pending commands and writes to the texture and frame buffers are flushed and completed, and the graphics engine returns to its IDLE state. While this command is used primarily for debugging and verification purposes, it is also used to clear the 3D status registers (**fbiTriangles, fbiPixelsIn**, **fbiPixelsOut**, **fbiChromaFail**, **fbiZfuncFail**, and **fbiAfuncFail**). Setting **nopCMD** bit(0)=1 clears the 3D status registers and flushes the graphics pipeline, while setting **nopCMD** bit(0)=0 has no affect on the 3D status registers but flushes the graphics pipeline. See the description of the **nopCMD** register in section 5 for more information.

8.74.2 TRIANGLE Command

TO BE COMPLETED.

8.74.3 FASTFILL Command

The FASTFILL command is used for screen clears. When the FASTFILL command is executed, the depth-buffer comparison, alpha test, alpha blending, and all other special effects are bypassed and disabled. The FASTFILL command uses the status of the RGB write mask (bit(9) of **fbzMode**) and the depth-buffer write mask (bit(10) of **fbzMode**) to access the RGB/depth-buffer memory. The FASTFILL command also uses bits (15:14) of **fbzMode** to determine which RGB buffer (front or back) is written. Prior to executing the FASTFILL command, the **clipLeftRight** and **clipLowYHighY** registers must be loaded with a rectanglar area which is desired to be cleared -- -- the **fastfillCMD** register is then written to initiate the FASTFILL command. Note that **clip** registers define a rectangular area which is inclusive of the **clipLeft** and **clipLowY** register values, but exclusive of the **clipRight** and **clipHighY** register values. Note also that the relative position of the Y origin (either top of bottom of the screen) is defined by **fbzMode** bit(17). The 24-bit color specified in the **Color1** register is written to the RGB buffer (with optional dithering as specified by bit(8) of **fbzMode**), and the depth value specified in bits(15:0) of the **zaColor** register is written to the depth buffer. See the description of the **fastfillCMD** register in section 5 for more information.

8.74.4 SWAPBUFFER Command

The SWAPBUFFER command is used to swap the drawing buffers to enable smooth animation. Since the SWAPBUFFER command is executed and queued like all other 2D and 3D commands, proper order is maintained and software does not have to poll and wait for vertical retrace to manually swap buffers – this frees the CPU to perform other functions while the graphics engine automatically waits for vertical retrace. When the SWAPBUFFER command is executed, **swapbufferCMD** bit(0) determines whether the drawing buffer swapping is synchronized with vertical retrace. Typically, it is desired that buffer swapping be synchronized with vertical retrace to eliminate frame “tearing” typically found on single buffered displays. If vertical retrace synchronization is enabled for double buffered applications, the graphics command processor blocks on a SWAPBUFFER command until the monitor vertical retrace signal is active. If the number of vertical retraces seen exceeds the value stored in bits(8:1) of **swapbufferCMD**, then the pointer used by the monitor refresh control logic is changed to point to another drawing buffer. If vertical retrace synchronization is enabled for triple buffered applications, the graphics processor does not block on a SWAPBUFFER command. Instead, a flag is set in the monitor refresh control logic that automatically causes the data pointer to be modified in the monitor refresh control logic during the next active vertical retrace period. Using triple buffering allows rendering operations to occur without waiting for the vertical retrace active period.

The **swapbufferCMD** must be proceeded by a direct write of the **swapPend** register. A write to the **swapPend** register increments the swap buffers pending field in the **status** registers. Conversely, when an actual frame buffer swapping occurs, the swap buffers pending field in the **status** register is decremented. The swap buffers pending field allows software to determine how many SWAPBUFFER commands are present in the Avenger FIFOs. See the descript of the **swapbufferCMD** register in section 5 for more information.

Since Avenger does not have fixed color buffer locations, 2 new registers are required for buffer display. **LeftOverlayBuf** and **rightOverlayBuf** are used by the video scanout section to determine the location of the current display buffer. The sequence of writes for double buffering would consist of writing to the **leftOverlayBuf** register and optionally the **rightOverlayBuf** (for stereo operations), followed by a direct write of **swapPend**, ending with a write to **swapbufferCMD** register. The leftOverlayBuf and rightOverlayBuf registers are fifoed, allowing tripple and quad buffering.

8.74.5 USERINTERRUPT Command

The USERINTERRUPT command allows for software-generated interrupts. A USERINTERRUPT command is generated by writing to the **userIntrCMD** register. **userIntrCMD** bit(0) controls whether a write to **userIntrCMD** generates a USERINTERRUPT. Setting **userIntrCMD** bit(0)=1 generates a USERINTERRUPT. **userIntrCMD** bit(1) determines whether the graphics engine stalls on software clearing of the user interrupt. By setting **userIntrCMD** bit(1)=1, the graphics engine stalls until the USERINTERRUPT is cleared. Alternatively, setting **userIntrCMD** bit(1)=0 does not stall the graphics engine upon execution of the USERINTERRUPT command, and additional graphics commands are processed without waiting for clearing of the user interrupt. A identification, or Tag, is also associated with an individual USERINTERRUPT command, and is specified by writing an 8-bit value to **userIntrCMD** bits(9:2).

User interrupts must be enabled before writes to the **userIntrCMD** are allowed by setting **intrCtrl** bit(5)=1. Writes to **userIntrCMD** when **intrCtrl** bit(5)=0 are "dropped" and do not affect functionality. A user interrupt is detected by reading **intrCtrl** bit (11), and is cleared by setting **intrCtrl** bit(11)=0. The tag of a generated user interrupt is read from **intrCtrl** bits (19:12). See the description of the **intrCtrl** and **userIntrCMD** registers in section 5 for more information.

8.75 Linear Frame Buffer Access (* FIX THIS *)

The Avenger linear frame buffer base address is located at a 8 Mbyte offset from the **memBaseAddr** PCI configuration register and occupies 4 Mbytes of Avenger address space (see section 4 for an Avenger address map). Regardless of actual frame buffer resolution, all linear frame buffer accesses assume a 2048-pixel logical scan line width. The number of bytes per scan line depends on the format of linear frame buffer access format selected in the **lfbMode** register. Note for all accesses to the linear frame buffer, the status of bit(16) of **fbzMode** is used to determine the Y origin of data accesses. When bit(16)=0, offset 0x0 into the linear frame buffer address space is assumed to point to the upper-left corner of the screen. When bit(16)=1, offset 0x0 into the linear frame buffer address space is assumed to point to the bottom-left corner of the screen. Regardless of the status of **fbzMode** bit(16), linear frame buffer addresses increment as accesses are performed going from left-to-right across the screen. Also note that clipping is not automatically performed on linear frame buffer writes if scissor clipping is not explicitly enabled (**fbzMode** bit(0)=1). Linear frame buffer writes to areas outside of the monitor resolution when clipping is disabled result in undefined behavior.

8.75.1 Linear frame buffer Writes

The following table shows the supported linear frame buffer write formats as specified in bits(3:0) of **lfbMode**:

Value	Linear Frame Buffer Access Format
	16-bit formats
0	16-bit RGB (5-6-5)
1	16-bit RGB (x-5-5-5)
2	16-bit ARGB (1-5-5-5)
3	Reserved
	32-bit formats
4	24-bit RGB (8-8-8)
5	32-bit ARGB (8-8-8-8)
7:6	Reserved
11:8	Reserved

12	16-bit depth, 16-bit RGB (5-6-5)
13	16-bit depth, 16-bit RGB (x-5-5-5)
14	16-bit depth, 16-bit ARGB (1-5-5-5)
15	16-bit depth, 16-bit depth

When writing to the linear frame buffer with a 16-bit access format (formats 0-3 and format 15 in **lfbMode**), each pixel written is 16-bits, so there are 2048 bytes per logical scan line. Remember when utilizing 16-bit access formats, two 16-bit values can be packed in a single 32-bit linear frame buffer write -- the location of each 16-bit component in screen space is defined by bit(11) of **lfbMode**. When using 16-bit linear frame buffer write formats 0-3, the depth components associated with each pixel is taken from the **zaColor** register. When using 16-bit format 3, the alpha component associated with each pixel is taken from the 16-bit data transfered, but when using 16-bit formats 0-2 the alpha component associated with each pixel is taken from the **zaColor** register. The format of the individual color channels within a 16-bit pixel is defined by the RGB channel format field in **lfbMode** bits(12:9). See the **lfbMode** description in section 5 for a detailed description of the rgb channel format field.

When writing to the linear frame buffer with 32-bit access formats 4 or 5, each pixel is 32-bits, so there are 4096 bytes per logical scan line. Note that when utilizing 32-bit access formats, only a single pixel may be written per 32-bit linear frame buffer write. Also note that linear frame buffer writes using format 4 (24-bit RGB (8-8-8)), while 24-bit pixels, must be aligned to a 32-bit (doubleword) boundary -- packed 24-bit linear frame buffer writes are not supported by Avenger. When using 32-bit linear frame buffer write formats 4-5, the depth components associated with each pixel is taken from the **zaColor** register. When using format 4, the alpha component associated with each pixel is taken from the **zaColor** register, but when using format 5 the alpha component associated with each pixel is taken from the 32-bit data transfered. The format of the individual color channels within a 24/32-bit pixel is defined by the rgb channel format field in **lfbMode** bits(12:9).

When writing to the linear frame buffer with a 32-bit access formats 12-14, each pixel is 32-bits, so there are 4096 bytes per logical scan line. Note that when utilizing 32-bit access formats, only a single pixel may be written per 32-bit linear frame buffer write. If depth or alpha information is not transfered with the pixel, then the depth/alpha information is taken from the **zaColor** register. The format of the individual color channels within a 24/32-bit pixel is defined by the rgb channel format field in **lfbMode** bits(12:9). The location of each 16-bit component of formats 12-15 in screen space is defined by bit(11) of **lfbMode**. See the **lfbMode** description in section 5 for more information about linear frame buffer writes.

8.75.2 Linear frame buffer Reads

It is important to note that reads from the linear frame buffer bypass the PCI host FIFO (as well as the memory FIFO if enabled) but are blocking. If the host FIFO has numerous commands queued, then the read can potentially take a very long time before data is returned, as data is not read from the frame buffer until the PCI host FIFO is empty and the graphics pixel pipeline has been flushed. One way to minimize linear frame buffer read latency is to guarantee that the Avenger graphics engine is idle and the host FIFOs are empty (in the **status** register) before attempting to read from the linear frame buffer.

8.76 Programming Caveats

The following is a list of programming guidelines which are detailed elsewhere but may have been overlooked or misunderstood:

8.76.1 Memory Accesses

All Memory accesses to Avenger registers must be 32-bit word accesses only. Linear frame buffer accesses may be 32-bit or 16-bit accesses, depending upon the linear frame buffer access format specified in **lfbMode**. Byte(8-bit) accesses are only allowed to Avenger linear frame buffer.

8.76.2 Determining Avenger Idle Condition

After certain Avenger operations, and specifically after linear frame buffer acceses, there exists a potential deadlock condition between internal Avenger state machines which is manifest when determining if the Avenger subsystem is idle. To avoid this problem, always issue a NOP command before reading the **status** register when polling on the Avenger busy bit. Also, to avoid asynchronous boundary conditions when determing the idle status, always read Avenger inactive in **status** three times. A sample code segment for determining Avenger idle status is as follows:

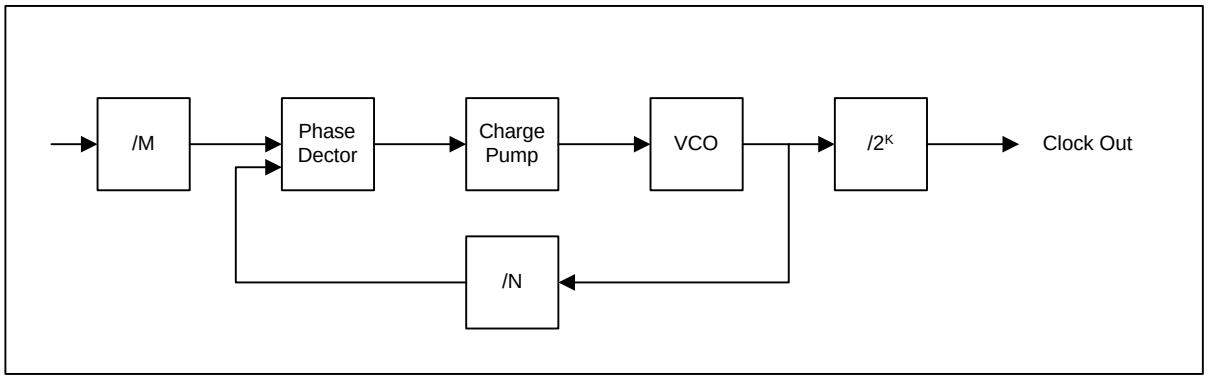
```
/**********************************************************
 * SST_IDLE:
 *    returns 0 if SST is not idle
 *    returns 1 if SST is idle
 **********************************************************/
SST_IDLE()
{
  ulong j, i;

  // Make sure SST state machines are idle
  PCI_MEM_WR(NOPCMD, 0x0);
  i = 0;
  while(1) {
    j = PCI_MEM_RD(STATUS);
    if(j & SST_BUSY)
      return(0);
    else
      i++;
    if(i > 3)
      return(1);
  }
}
```

8.76.3 Triangle Subpixel Correction

Triangle subpixel correction is performed in the on-chip triangle setup unit of Avenger. When subpixel correction is enabled (**fbzColorPath**(26)=1), the incoming starting color, depth, and texture coordinate parameters are all corrected for non-integer aligned starting triangle <x,y> coordinates. The subpixel correction in the triangle setup unit is performed as the starting color, depth, and texture coordinate parameters are read from the PCI FIFO. As a result, the exact data sent from the host CPU is changed to account for subpixel alignments. If a triangle is rendered with subpixel correction enabled, all subsequent triangles must resend starting color, depth, and texture coordinate parameters, otherwise the last triangle's subpixel corrected starting parameters are subpixel corrected (again!), and incorrect results are generated.

9. 1. PLL Registers



Register Name	Address	Bits	R/W	Description
pllCtrl0	0x40-0x43	31:0	R/W	Video Clock PLL
pllCtrl1	0x44-0x47	31:0	R/W	GRX Clock PLL
pllCtrl2	0x48-0x4b	31:0	R/W	AGP PLL.

Genlock mode: in order for the register 3da (vga register) to reflect the status of vsync correct, **vgaInit0**[1]

needs to be set

9.1 PllCtrl0, PllCtrl1 registers

These registers control the frequency of the core clock (GRX Clock) and the Video Clock.

Bit	Description
1:0	K, Post divider value
7:2	M, PLL input divider
15:8	N, PLL multiplier
16	Test. (0=normal operation, 1 = CLK is output of VCO). Default = 0.

Frequency output of PLL's is given:

fout = 14.31818 * (N + 2) / (M + 2) / (2 ^ K).

NOTE: if the deviceID==4, then the GRX clock pll's M[7:2] value is forced to 0x18 (24 decimal), which limits the maximum frequency of the grx_clk to 141MHz. Software must adjust calculations for setting the frequency accordingly.

9.2 PllCtrl2 register/Controlling AGP/PCI clocking in Avenger.

This register controls the test modes of the output of the AGP 2X PLL.

Bit	Description
1:0	State of Test Mux

The state of the PllCtrl2 register, combined with the state of the strapping bits for PCI_66MHz and PLL_BYPASS control the AGP/PCI clocks in Avenger.

Signal Definitions:		
PCI_66MHz	Strap[2]/VMI_DATA_2	Strap that sets the chip's PCI status register to report Avenger as a 66MHz device.
PLL_BYPASS	Strap[11]/VMI_ADDR_3	Strap that contributes to bypassing the AGP/PCI PLL
CLK66		Internal Avenger clock that runs all of the AGP/PCI logic.
CLK133		Internal Avenger clock used only to output AGP2X signals.

Inputs				**Resulting Internal Avenger Clocks**		**Resulting Operation Mode**
PCI 66Mhz (strap[2])	**PLL_BYPASS (strap[11])**	**PllCtrl2[1]**	**PllCtrl2[0]**	**Clk66 (Internal Avenger PCI clock)**	**Clk133 (Internal Avenger AGP2X clk)**	
0	X	x	x	pci_clk_in	pci_clk_in	**PCI, bypass PLL**
1	0	0	0	¼ pll output	½pll output	**AGP(1x or 2x), PLL ON**
1	0	0	1	¼ pci_clk_in	½pci_clk_in	Test only
1	0	1	0	pci_clk_in	½pll output	AGP, bypass PLL (1x only)
1	0	1	1	pci_clk_in	½pci_clk_in	**AGP, bypass PLL (1x only)**
1	1	x	x	¼ pci_clk_in	½pci_clk_in	Test only

VERY IMPORTANT!!: Only the operation modes that are in BOLD, above, should be used with Avenger. These modes are the following:

- PCI, bypass PLL
- AGP, PLL ON (this mode should work with PCI as well)
- AGP, bypass PLL (1X only)

10. 2.DAC Registers

Register Name	I/O Address	Bits	R/W	Description
dacMode	0x4c-0x4f	4:0	R/W	Dac Mode 2:1 or 1:1
dacAddr	0x50-0x53	8:0	R/W	Dac pallette address
dacData	0x54-0x57	23:0	R/W	Dac data register
reserved	0x58-0x5b	na		

10.1 2.1 dacMode

Bit	Description
0	Dac Mode 2:1 or 1:1
1	Enable DPMS on Vsync
2	Force Vsync value.
3	Enable DPMS on Hsync
4	Force Hsync value.

10.2 2.2 dacAddr

Bit	Description
8:0	Pallette Address

This is the 9 bit CLUT address used for programming the CLUT. Unlike the VGA mechanism, this address does not auto increment, but has access to the entire 512 entries in the CLUT.

10.3 2.3 dacData

Bit	Description
23:0	Dac color value

This is the 24 bit RGB value at the index programmed into **dacAddr**. The color values are always stored with red in bits [23:16], green in bits [15:8] and blue in bits [7:0].

11. 3.Video Registers(PCI)

Register Name	I/O Addr	Bits	R/W	Description
vidTvOutBlankVCount	0x3c	31:0	R/W	TV Out Vertical Active Start/End
vidMaxRgbDelta	0x58	23:0	R/W	Maximum delta values for video filtering
vidProcCfg	0x5c	31:0	R/W	Video Processor configuration
hwCurPatAddr	0x60	23:0	R/W	Cursor Pattern Address
hwCurLoc	0x64	26:0	R/W	X and Y location of HW cursor
hwCurC0	0x68	23:0	R/W	Hw cursor color 0
hwCurC1	0x6c	23:0	R/W	Hw cursor color 1
vidInFormat	0x70	31:0	R/W	Video In Format
vidTvOutBlankHCount	0x74	31:0	R/W	TV Out Horizontal Active Start/End
vidSerialParallelPort	0x78	31:0	R/W	Serial and Parallel Ports
vidInXDecimDeltas	0x7c	31:0	R/W	Video In horizontal decimation delta 1 & 2.
vidInDecimInitErrs	0x80	31:0	R/W	Video In horizontal and vertical decimation initial error term
vidInYDecimDeltas	0x84	31:0	R/W	Video In vertical decimation delta 1 & 2
vidPixelBufThold	0x88	17:0	R/W	Video Pixel Buffer Threshold
vidChromaMin	0x8c	31:0	R/W	Chroma Key minimum value
vidChromaMax	0x90	31:0	R/W	Chroma Key maximum value
vidInStatusCurrentLine	0x94	18:0	R	Video In Status and Current Scan line
vidScreenSize	0x98	23:0	R/W	Screen resolution
vidOverlayStartCoords	0x9c	31:0	R/W	Start Surface Coordinates [31:28] Overlay Start Screen Coordinates
vidOverlayEndScreenCoord	0xa0	23:0	R/W	Overlay End Screen Coordinates
vidOverlayDudx	0xa4	19:0	R/W	Overlay horizontal magnification factor
vidOverlayDudxOffsetSrcWidth	0xa8	31:0	R/W	Overlay horizontal magnification factor initial offset (bit 18:0) Overlay source surface width (bit 31:19)
vidOverlayDvdy	0xac	19:0	R/W	Overlay vertical magnification factor
vidOverlayDvdyOffset	0xe0	18:0	R/W	Overlay vertical magnification factor initial offset
vidDesktopStartAddr	0xe4	23:0	R/W	Desktop start address
vidDesktopOverlayStride	0xe8	31:0	R/W	Desktop and Overlay strides (linear or tile)
vidInAddr0	0xec	23:0	R/W	Video In Buffer 0 starting address
vidInAddr1	0xf0	23:0	R/W	Video In Buffer 1 starting address
vidInAddr2	0xf4	23:0	R/W	Video In Buffer 2 starting address
vidInStride	0xf8	14:0	R/W	Video In Buffer stride (linear or tile)
vidCurrOverlayStartAddr	0xfc	23:0	R	Current overlay start address in use

11.1.1 3.1.1 vidTvOutBlankVCount

If TV Out Genlock is enabled (VidInFormat[16] == 1'b1, VidInFormat[18]== 1'b1), vertical blank_n signal is de-asserted when the number of positive edges of tv_hsync after the positive edge of tv_vsync == vidTvOutBlankVCount bits[10:0].

Vertical blank_n signal is re-asserted when the number of positive edges of tv_hsync after the positive edge of tv_vsync == vidTvOutBlankVCount bits[26:16].

Output blank_n == horizontal blank_n AND vertical blank_n.

Note that the value in bits[26:16] needs to be greater than bits[10:0]. The clock cycles are based on the clock coming in through the tv_inclk pin.

Bit	Description
10:0	The number of tv_hsync LEADING edges after the LEADING edge of tv_vsync before the vertical active region starts (i.e., vertical blank becomes deasserted).
15:11	Reserved
26:16	The number of tv_hsync LEADING edges after the LEADING edge of tv_vsync before the vertical active region ends (i.e., vertical blank is re-asserted).
31:27	reserved

11.1.2 3.1.2 vidMaxRgbDelta

The vidMaxRgbDelta register specifies the threshold values allowed for the deviation of a pixel's luma and chroma components when the pixel is used in the video filter (4x1 tap filter or 2x2 box filter). If the absolute deviation of a pixel component exceeds its threshold, the particular component will be replaced by that of the center pixel before the pixel is used in the filter.

Bit	Description
5:0	Maximum blue/V/Cr delta for video filtering (unsigned)
13:8	Maximum green/U/Cb delta for video filtering (unsigned)
21:16	Maximum red/Y delta for video filtering (unsigned)

11.1.3 3.1.3 vidProcCfg Register

The **vidProcCfg** register is the general configuration register for the Video Processor. It is written by the host upon reset only.

Bit	Description
0	1: Video Processor on, VGA mode off; 0: Video Processor off, VGA mode on.
1	1: X11 cursor mode; 0: Microsoft Windows cursor mode.
2	Overlay stereo enable. 0 = disabled, 1 = enabled.
3	Interlaced video out enable. For Avenger, since interlaced video out is not supported, this bit should remain 0 all the time.
4	Half mode. 0 = disabled. 1 = enabled where desktop stride is added every other lines.
5	ChromaKeyEnable. 0 = off. 1 = on.
6	ChromaKeyResultInversion: (0 = desktop transparent if desktop color matches or falls within the chroma-key color range; 1 = desktop transparent if desktop color does not match or fall within the chroma-key range)
7	Desktop surface enable. 0 = do not fetch the desktop surface, 1 = fetch desktop surface
8	Overlay surface enable. 0 = do not fetch the overlay surface, 1 = fetch overlay surface
9	Video-in data displayed as overlay enable. 0 = do not display the video-in buffer directly as overlay. 1= use the video-in buffer address as the overlay start address (auto-flipping).
10	Desktop clut bypass. 0 = do not bypass the clut in the RAMDAC, 1 = bypass the clut
11	Overlay clut bypass. 0 = do not bypass the clut in the RAMDAC, 1 = bypass the clut
12	Desktop clut select. 0 = use the lower 256 entries of the clut. 1 = use the upper 256 entries.
13	Overlay clut select. 0 = use the lower 256 entries of the clut. 1 = use the upper 256 entries.

14	Overlay horizontal scaling enable. 0=disabled. 1=enabled. Magnification factor determined by vidOverlayDudx.
15	Overlay vertical scaling enable. 0=disabled. 1=enabled. Magnification factor determined by vidOverlayDvdy.
17:16	Overlay filter mode 00: point sampling 01: 2x2 dither subtract followed by 2x2 box filter (for 3d only) 10: 4x4 dither subtract followed by 4x1 tap filter (for 3d only) 11: bilinear scaling
20:18	Desktop pixel format 000: 8bit palettized 001: RGB565 undithered 010: RGB24 packed 011: RGB32 100: Reserved 101: Reserved 110: Reserved 111: Reserved
23:21	Overlay pixel format 000: Reserved 001: RGB565 undithered 010: Reserved 011: Reserved 100: YUV411 101: YUYV422 110: UYVY422 111: RGB565 dithered
24	Desktop Tile Space Enable. 0 = linear space, 1 = tile space
25	Overlay Tile Space Enable. 0 = linear space, 1 = tile space
26	2X mode which refreshes two screen pixels per video clock. 0 = 1X mode, 1 = 2X mode.
27	HW cursor enable. 0 = disabled, 1 = enabled.
28	Disable memory optimization for desktop requests, default=0. (0=optimization ON, 1=OFF).
29	Disable memory optimization for overlay requests, default=0. (0=optimization ON, 1= OFF).
30	Reserved.
31	Backend deinterlacing for video overlay. 0 = No deinterlacing in the backend pipe. 1 = Backend deinterlacing (Bob method). Bob method displays either the even or odd frame at a time, and interpolates two interlaced lines to get the missing field. It is not supported in 2X mode.

How to program for Backend deinterlacing (Bob method):
The only thing this option effects is that when the video processor displays the even field, it adds 0.5 to the initial vertical offset (initial dvdy offset) used by the backend bilinear scaler. Everything else is the same.
Since deinterlacing in the backend uses the bilinear scaler unit to interpolate between two interlaced lines, the host needs to enable bilinear filtering, overlay vertical scaling, overlay horizontal scaling, and set up the initial dvdy offset, dvdy, initial dudx offset and dudx correctly according to the desired magnification factor between the source video and displayed video. The suggested setting for the parameters for backend deinterlacing without horizontal magnification are: bilinear filter enable = 1, overlay vertical scaling

enable = 1, overlay horizontal scaling enable = 0, initial dvdy offset = 0.25, dvdy = 0.5. Initial dudx offset and dudx are don't cares.
Backend deinterlacing is not supported for 2x mode (2-pixel per video clk mode) since bilinear filtering is not available in 2x mode.

How to program for stereo video display:
The mainstream way of stereo support is alternating field display with quad buffers. The host writes the left overlay start address into leftOverlayBuf register and the right address into rightOverlayBuf register. These registers are described in the 3D section of the spec. Then the host executes swap buffer command, and the pair of addresses will be pushed into the overlay start address fifo between the 3D and the video. Video will switch between the two addresses for screen refresh. When the host is ready to provide the next pair of right/left buffer addresses, it executes swap buffer command again, and the new pair of addresses will be pushed into the address fifo. Only at vertical retrace and the completion of refreshing the right frame will video use the new pair of overlay addresses. The stereo_out pin will indicate the left/right field of the frame being displayed. Also, when stereo display is enabled, the swap buffer command needs to be executed with sync to vsync enabled (mid-frame swap disabled). Mid-frame swap and stereo video display are mutually exclusive features.

When dual buffer is used, there are two different options, and each has its own drawback. First, the host enables stereo video display, and writes both overlay buffer addresses into the leftOverlayBuf and rightOverlayBuf registers, and execute swap buffer command once only. In this case, video will continue to switch between the two addresses regardless of whether the next frame is ready or not. Stereo_out will indicate the left/right field.
The other method is to turn off stereo, and video will look at only the leftOverlayBuf register. In this case, the host executes a swap buffer command only when it is done rendering the next frame and has written the buffer address to the leftOverlayBuf register. Therefore, video may display in time-sequence: left -> left -> right -> left frames depending on how long a frame takes to render. However, since stereo is disabled, stereo_out will remain
low all the time.

The stereo_out pin is used to control the shutter of stereo glasses. Another alternative supported by StereoGraphics' Simuleyes is to use a white strip displayed at the bottom of the monitor to control the shuttering of the glasses. Depending on the length of the white strip (1/4 or ¾of the screen width), the glasses detect if the screen is displaying the right or left field.

Finally for alternate line display for Head Mount Displays. The requirement is left field on scanline 0 and right field on scanline 1 and so on in the display. Avenger video does not have support for this.

11.1.4 3.1.4 hwCurPatAddr Register

The **hwCurPatAddr** register stores the starting address of two monochrome cursor patterns. Each pattern is a bitmap of 64-bit wide and 64-bit high (a total of 8192 bits). The two patterns are stored in such a way that pattern 0 always resides in the lower half (least significant 64-bit) of a 128-bit word and pattern 1 the upper half. In other words, each 128-bit word consists of one line from pattern 0 and one line from pattern 1. At each horizontal retrace, the Video Processor checks to see whether the cursor location falls on the current scanline. If so, it fetches from the memory eight words of cursor patterns at a time. The eight words are then stored in the on-chip ram for use in the next eight scanlines. This reduces the number of memory accesses for cursor patterns from 64 to 8 times per screen refresh. Cursor patterns always reside in linear address space, and the linear stride is always 16 bytes. The video processor figures out the shape and color of the cursor for the current scanline according to the following table:

Bit from Pattern 0	Bit from Pattern 1	Displayed cursor (Microsoft window)	Displayed cursor (X11)
0	0	HWCurC0	Current Screen Color
0	1	HWCurC1	Current Screen Color
1	0	Current Screen color	HWCurC0
1	1	NOT current screen color	HWCurC1

Bit	Description
23:0	Physical address of where the cursor pattern resides in the memory.

11.1.5 3.1.5 hwCurLoc Register

The **hwCurLoc** register stores the x and y coordinates of the bottom right corner of the cursor. The coordinates are unsigned, and range from 0 to 2047. This allows a partial cursor to be displayed in all edges of the screen.

Bit	Description
10:0	X coordinate of the bottom right corner of the cursor. Undefined upon reset.
26:16	Y coordinate of the bottom right corner of the cursor. Undefined upon reset.

11.1.6 3.1.6 hwCurC0 Register

The **hwCurC0** register stores color 0 of the cursor.

Bit	Description
7:0	Blue value of cursor color0
15:8	Green value of cursor color0
23:16	Red value of cursor color0

11.1.7 3.1.7 hwCurC1 Register

The **hwCurC1** register stores color 1 of the cursor.

Bit	Description
7:0	Blue value of cursor color1
15:8	Green value of cursor color1
23:16	Red value of cursor color1

11.1.8 3.1.8 vidInFormat

The VidInFormat register allows the host to specify the data format of the video-in and tv-out data.

Bit	Description
0	Reserved
3:1	(VMI only) Video-In data format 110: 8bit YCbCr 4:2:2 (UYVY) 101: 8bit YCbCr 4:2:2 (YUYV) 100: 8bit YCbCr 4:1:1
4	(VMI only) Video-In de-interlacing mode. (0 = No deinterlacing applied to the video data coming in; 1 = Weave method deinterlacing, i.e. the video-in port will merge two consecutive VMI frames into one inside the frame buffer before signaling a frame is done in the vidStatus register.)

5	Vmi_vsync_in polarity. (1=active low; 0=active high (default))
6	Vmi_hsync_in polarity. (1=active low; 0=active high (default))
7	(VMI only) Vmi_vactive_in polarity. (1=active low; 0=active high (default))
8	(TV out only) G4 for posedge (1=Brooktree TV out support; 0=Chrontel) 1: Brooktree TV encoder samples at falling edge for the following data; 0: Chrontel TV encoder samples at rising edge for the following data Data[11] G0[4] Data[10] G0[3] Data[9] G0[2] Data[8] B0[7] Data[7] B0[6] Data[6] B0[5] Data[5] B0[4] Data[4] B0[3] Data[3] G0[0] Data[2] B0[2] Data[1] B0[1] Data[0] B0[0] 1: Brooktree TV encoder samples at rising edge for the following data; 0: Chrontel TV encoder samples at falling edge for the following data Data[11] R0[7] Data[10] R0[6] Data[9] R0[5] Data[8] R0[4] Data[7] R0[3] Data[6] G0[7] Data[5] G0[6] Data[4] G0[5] Data[3] R0[2] Data[2] R0[1] Data[1] R0[0] Data[0] G0[1]
10:9	(VMI only) VideoIn buffering mode select (00=single buffer, 01=double buffer, 10= triple buffer)
11	(VMI only) VideoIn buffer tile space enable. 0 = linear space, 1= tile space.
12	TvOut_vsync_in polarity. (1=active low; 0=active high (default))
13	TvOut_hsync_in polarity. (1=active low; 0=active high (default))
14	VMI interface enable.
15	TV out interface enable.
16	Genlock enable. The VMI logic of the video controller uses vmi_pixclk_in as its clock. By setting bit 16 to 1, it allows Avenger to genlock to the clock of an external VMI device or TV encoder. 0: The remaining video logic uses a separate video clock from the on-chip PLL. For VMI and TV encoder slave mode. 1: The remaining video logic uses the genlock source (as selected by vidInFormat bit[18]) to drive its clock. If the genlock source is VMI, Avenger is genlocked to vmi_pixclk_in. If the genlock source is TV encoder, Avenger is genlocked to tv_inclk. For TV encoder master mode.

17	(VMI/TV out) not_use_vga_timing_signal (Timing signals include vert_exra, display_ena, vfrontporch_active, vbackporch_active, vblank, vga_blank_n, vga_vsync, vga_hsync) 0: Use the timing signals supplied by the VGA. For VMI and TV out slave mode. 1: Do not use the timing signals from the VGA. Timing signals are either supplied by the genlock source (as selected by vidInFormat bit[18]) or generated internally by the video controller. If genlock source is VMI, vmi_vsync and vmi_hsync are input from the VMI device (Vmi_vactive is always an input to Avenger from the VMI device). If the genlock source is TV encoder, tvout_vsync and tvout_hsync are input from the TV encoder (tvout_blank is always an output from Avenger to the TV encoder device).
18	Genlock source select. (0=VMI (default), 1=TV encoder in master mode) 0: The inputs vmi_pixclk_in, vmi_vsync, and vmi_hsync are used to drive
19	TvOut select display_ena. If on, selects display_ena instead of vga_blank for driving output video. (0=off; 1=on).
20	Video-In horizontal decimation enable. (0=off; 1=on)
21	Video-In vertical decimation enable. (0=off; 1=on)
31:22	Reserved

The following configurations of external VMI and/or TV encoder devices are supported by Avenger –

Configuration	Genlock Enable VidInFormat[16]	Genlock Source VidInFormat[18]	Not_use_vga_timing_signal VidInFormat[17]
TV encoder master	1	1 (Tv encoder)	1
TV encoder slave	0	Don't care	0
VMI genlock	1	0 (VMI)	0
VMI slave*	0	Don't care	0
TV encoder master + VMI slave	1	1 (Tv encoder)	1
TV encoder slave + VMI genlock*	1	0 (VMI)	0

* While it's possible to configure the VMI device as master mode (i.e., genlock is enabled, VMI is the genlock source, and not_use_vga_timing_signal==1), it probably doesn't make sense to do so, because one pixel of input data from the VMI device requires two clocks whereas one pixel of output data to the monitor or TV encoder requires only one clock, so the timing of the input and output devices can't be aligned.

VMI field detection

Note that the polarity of the VMI Vsync, Hsync, and Vactive signals is programmable. The inactive going edge of the Vsync signal indicated whether the field is odd or even. If Hsync is active during the inactive going edge of Vsync, the field is even. If Hsync is inactive, the field is odd.

11.1.9 3.1.9 vidSerialParallelPort Register

The vidSerialParallelPort register controls the chip's I2C, DDC, GPIO, and the host port interface. Since VMI and ROM share pins for their interface, a pin can be input or output depending on which interface has control of the pin at that time. GPIO[0] is a hardwired output pin designed to be an output enable of the on-board tristate drivers. GPIO[0] is asserted low, when the VMI device has control of the shared pins, and is driving pixdata[7:0], vmi_rdy_n, and vmi_intreq_n as input to Avenger. GPIO[0] is pulled high, when ROM controls the shared pins, and pixdata[7:0], vmi_rdy_n, and vmi_intreq_n are output of Avenger.

NOTE: the ROM interface controls whether VMI or the ROM owns the shared busses. So, if there are ROM transactions, then ROM will own the bus, else VMI owns it. GPIO[0] is a reflection of the ownership.

GPIO[1] is software programmable, and is used to control the output enable of the on-board tristate drivers for vmi_pixclk, vmi_vsync, vmi_hsync, and vmi_vactive. These are the signals that should be continually driven by the external vmi device even when the ROM is using the shared pins (ROM does not use the vmi_pixclk, vmi_vsync, vmi_hsync, and vmi_vactive pins). Otherwise the internal state of the vmi controller in Avenger may be messed up. Vmi_cs_n cannot be used in lieu of GPIO[1] for this purpose because the chip select pin can be turned off by vmi parallel host interface enable bit (bit 0 below).

Bit	Written By	Description	
0	host	VMI parallel host interface enable. (0=off, 1=on); Default to 0 upon reset.	
1	host	VMI CS_N (Chip Select)	
		Mode A	Mode B
2	host	VMI DS_N (Data Strobe)	VMI RD_N (Read)
3	host	VMI RW_N (Read/ Write_n)	VMI WR_N (Write)
4	VMI	VMI data DTACK_N (Data Acknowledge)	VMI data RDY (Data Ready)
5	host	VMI Data output enable_n. (0 = enabled, 1 = disabled); Default to 1 upon reset. If set==1, this bit disables the VMI_DATA[7:0] bus from driving out.	
13:6	host/VMI	VMI Data (Input / Output)	
17:14	host	VMI Address	
		DDC interface	
18	host	DDC port enable (0 = disabled, 1 = enabled) Default to 0 upon reset.	
19	host	DDC DCK write (0 = DCK pin is driven low, 1= DCK pin is tri-stated) When this pin is tri-stated, other devices can drive this line, and the final state of the pin is reflected in bit 26. Default to 1 upon reset.	
20	host	DDC DDA write (0 = DDA pin is driven low, 1= DDA pin is tri-stated) When this pin is tri-stated, other devices can drive this line, and the final state of the pin is reflected in bit 27. Default to 1 upon reset.	
21	Monitor	DDC DCK state (read only, 0 = low, 1 = tri-stated which means no device is driving this pin)	
22	Monitor	DDC DDA state (read only, 0 = low, 1 = tri-stated which means no device is driving this pin)	
		I2C interface	
23	host	I2C port enable (0 = disabled, 1 = enabled) Default to 0 upon reset.	
24	host	I2C SCK write (0 = SCK pin is driven low, 1= SCK pin is tri-stated) When this pin is tri-stated, other devices can drive this line, and the final state of the pin is reflected in bit 21. Default to 1 upon reset.	
25	host	I2C SDA write (0 = SDA pin is driven low, 1= SDA pin is tri-stated) When this pin is tri-stated, other devices can drive this line, and the final state of the pin is reflected in bit 22. Default to 1 upon reset.	
26	VMI/ encoder	I2C SCK state (read only, 0 = low, 1 = tri-stated which means no device is driving this pin)	
27	VMI/ encoder	I2C SDA state (read only, 0 = low, 1 = tri-stated which means no device is driving this pin)	
		Misc.	
28	host	VMI reset_n (1 = normal. 0 = reset VMI device.) Default to 0 upon reset.	
29	host	output only gpio GPIO[1] output	

30	VMI/ encoder	input only gpio GPIO[2] input
31	Host	TV out reset_n (1 = normal, 0 = reset TV out device). Default to 0 upon reset.

VMI and ROM Pin Sharing (see Notes below)

Pin		Rom Access		VMI Access	
Name	**Number**	**Function**	**Type**	**Function**	**Type**
Pixdata/a0		A0	out	Y0/Cr0/Cb0*	in
Pixdata/a1		A1	out	Y1/Cr1/Cb1*	in
Pixdata/a2		A2	out	Y2/Cr2/Cb2*	in
Pixdata/a3		A3	out	Y3/Cr3/Cb3*	in
Pixdata/a4		A4	out	Y4/Cr4/Cb4*	in
Pixdata/a5		A5	out	Y5/Cr5/Cb5*	in
Pixdata/a6		A6	out	Y6/Cr6/Cb6*	in
Pixdata/a7		A7	out	Y7/Cr7/Cb7*	in
vmi_adr/a8		A8	out	vaddr0	out
vmi_adr/a9		A9	out	vaddr1	out
vmi_adr/a10		A10	out	vaddr2	out
vmi_adr/a11		A11	out	vaddr3	out
vmi_cs_n		CANNOT USE!		vmi_cs_n	out
vmi_rw		A14	out	vmi_rw_n/ vmi_wr_n	out
vmi_ds_n/a15		A15	out	vmi_ds_n/ vmi_rd_ n	out
vmi_rdy/busy		A12	out	vmi_dtack_n/ vmi_rdy_n*	in
vmi_hdata		D0	in/out	vmi_hd_0	in/out
vmi_hdata		D1	in/out	vmi_hd_1	in/out
vmi_hdata		D2	in/out	vmi_hd_2	in/out
vmi_hdata		D3	in/out	vmi_hd_3	in/out
vmi_hdata		D4	in/out	vmi_hd_4	in/out
vmi_hdata		D5	in/out	vmi_hd_5	in/out
vmi_hdata		D6	in/out	vmi_hd_6	in/out
vmi_hdata		D7	in/out	vmi_hd_7	in/out
Hsync		NOT USED		hsync	in
Vsync		NOT USED		vsync_n	in
Blank_n		NOT USED		blank_n	in
pix_clk_in		NOT USED		vid_clk_in	in
vmi_intreq_n		A13	out	vmi_int_n*	in
vmi_reset_n		NOT USED		reset_n	out
rom_oe_n		rom_oe_n	out	CANNOT USE!	
rom_we_n		rom_we_n	out	CANNOT USE!	
i2c_clk		NOT USED		i2c_clk	out
i2c_data		NOT USED		i2c_data	in/out
gp_io[0]		vmi_oe_n	out	vmi_oe_n	out
gp_io[1]		vmi_sync_oe_n	out	vmi_sync_oe_n	out
gp_io[2]		gp_in	in	gp_in	in

* means the signal may be buffered from the VMI data bus to ensure that it is not driven during ROM accesses.

TV Encoder Pins (see Notes below)

Name	Pin Number	TV Encoder Function rising/falling edge	Type
tv_data[0]		B0/G1	out
tv_data[2]		B1/R0	out
tv_data[2]		B2/R1	out
tv_data[3]		G0/R2	out
tv_data[4]		B3/G5	out
tv_data[5]		B4/G6	out
tv_data[6]		B5/G7	out
tv_data[7]		B6/R3	out
tv_data[8]		B7/R4	out
tv_data[9]		G2/R5	out
tv_data[10]		G3/R6	out
tv_data[11]		G4/R7	out
tv_clk_out		Clock_out	out
tv_hsync		Hsync	in/out
tv_vsync		Vsync	in/out
tv_blank		Blank_n	in/out
tv_inclk		Clock_in	in
tv_reset		Reset_n	out

Notes:
1. Rom access and VMI video data/host port access can only be performed separately.
2. The Type field in the tables above are referenced to Avenger.
3. Programming of the VMI or TV Encoder device can be done via I2C, e.g. setting PAL mode.
4. The TV encoder must be able to operate in Master mode where it supplies the clock, vsync, hsync, blank and Avenger outputs tv_clk_out (delayed version of tv_inclk) and synchronous data.
5. We must route a reference board to make sure the pin functions have been shared to provide a decent route.
6. The ROM cs_n is tied to GND, the oe_n and we_n are used to control read/write respectively.

11.1.10 3.1.10 vidTvOutBlankHCount

If TV Out Genlock is enabled (VidInFormat[16] == 1'b1, VidInFormat[18]== 1'b1), and Not_use_vga_timing_signal is asserted, vidTvOutBlankHCount bits[10:0] contains the number of clock cycles after the leading edge of tv_hsync before the horizontal active region starts (i.e., horizontal blank becomes deasserted).

vidTvOutBlankHCount bits[26:16] contains the number of clock cycles after leading edge of tv_hsync before the horizontal active region ends (i.e., horizontal blank is re-asserted).

Output blank_n == horizontal blank_n AND vertical blank_n.

Note that the value in bits[26:16] needs to be greater than bits[10:0]. The clock cycles are based on the clock coming in through the tv_inclk pin.

Bit	Description
10:0	The number of clock cycles after the leading edge of tv_hsync before the horizontal active region starts (i.e., horizontal blank becomes deasserted).
15:11	Reserved

26:16	The number of clock cycles after leading edge of tv_hsync before the horizontal active region ends (i.e., horizontal blank is re-asserted).
31:27	Reserved

11.1.11 3.1.11 vidInXDecimDeltas (for VMI downscaling Brensenham Engine)/ vidTvOutBlankHCount (for TV out master mode)

If VideoIn Interface is configured to VMI mode (i.e., VidInFormat[15:14] == 2'b01), vidInXDecimDeltas bits [11:0] contain the width of the destination video-in surface (width of the video overlay stored in the frame buffer) in number of pixels. VidInXDecimDeltas bits[27:16] contain the width difference between the source video-in surface (from VMI port) and destination video-in surface in number of pixels (Source - Destination)

Bit	Description
11:0	The positive (unsigned) value added to the error term when the horizontal Bresenham error term is <0. It is programmed to be the **width of the destination video-in surface in number of pixels.**
15:12	reserved
27:16	The positive (unsigned) value added to the error term when the horizontal Bresenham error term is >0. It is programmed to be the difference between the width of the source and destination video-in surfaces. **(Source - Destination) in number of pixels.**
31:28	reserved

11.1.12 3.1.12 vidInDecimInitErrs

Bit	Description
12:0	The signed (2's complement) initial value of the error term in the horizontal Bresenham accumulator
15:13	reserved
28:16	The signed (2's complement) initial value of the error term in the vertical Bresenham accumulator
31:29	reserved

11.1.13 3.1.13 vidInYDecimDeltas

If VideoIn Interface is configured to VMI mode (i.e., VidInFormat[15:14] == 2'b01), vidInYDecimDelta bits[11:0] contain the height of the destination video input window (height of the video overlay stored in the frame buffer) in number of lines. vidInYDecimDeltas contains the height difference between the source video surface (from VMI port) and destination video input window in number of lines (Source - Destination).

Bit	Description
11:0	The positive value added to the error term when the vertical Bresenham error term is <0
15:12	reserved
27:16	The positive value added to the error term when the vertical Bresenham error term is >0
31:28	reserved

Bresenham scaler for scaling down a video window in the horizontal direction:

error = vidInXDecimInitErr;

repeat until the source pixels of a video window scanline are exhausted

if (error < 0)

move to next source pixel

error = error + vidInXDecimDelta1

else

select the current source pixel as the destination pixel

move to next source pixel

error = error - vidInXDecimDelta2

Bresenham scaler for scaling down a video window in the vertical direction:

error = vidInYDecimInitErr

at each VideoIn Hsync

if (error < 0)

skip the whole line of video in data

error = error + vidInYDecimDelta1

else

select the current line of video in data

error = error - vidInYDecimDelta2

11.1.14 3.1.14 vidPixelBufThold

The vidPixelBufThold determines how many empty slots in each of the three pixel buffers will trigger refilling of the buffers.

Bit	Description
5:0	Primary pixel buffer low watermark (0x0 – 1 empty slot; 0x3f – 64 empty slots)
11:6	Secondary pixel buffer 0 low watermark (0x0 – 1 empty slot; 0x3f – 64 empty slots)
17:12	Secondary pixel buffer 1 low watermark (0x0 – 1 empty slot; 0x3f – 64 empty slots)

11.1.15 3.1.15 vidChromaKeyMin Register

The vidChromaKeyMin register contains the lower bound of the chroma key color.

Bit	Description
	8-bit desktop color format
7:0	chroma key color
31:8	Reserved
	15-bit desktop color format
4:0	Blue value of the chroma -key
9:5	Green value of the chroma -key
14:10	Red value of the chroma -key
31:15	Reserved
	16-bit desktop color format

4:0	Blue value of the chroma -key
10:5	Green value of the chroma -key
15:11	Red value of the chroma -key
31:16	Reserved
	24-bit desktop color format
7:0	Blue value of the chroma -key
15:8	Green value of the chroma -key
23:16	Red value of the chroma -key
31:24	Reserved
	32-bit desktop color format
7:0	Blue value of the chroma -key
15:8	Green value of the chroma -key
23:16	Red value of the chroma -key
31:24	Reserved

11.1.16 3.1.16 vidChromaKeyMax Register

The vidChromaKeyMax register contains the upper bound of the chroma key color. It is the same as vidChromaKeyMin if the chroma-key is a single color instead of a range.

Bit	Description
31:0	Format same as vidChromaKeyMin Register

11.1.17 3.1.17 vidInStatusCurrentLine Register

The vidInStatusCurrentLine register contains the current scan out line. As the vertical beam scans down the display this register is incremented.

Bit	Description
10:0	Current Video scan line.

The vidInStatusCurrentLine register also allows the host to read the status of the video-in port, and implement manual buffer flipping for the video-in data.

Bit	Description
17:16	Video-in buffer VMI just finishes writing to. 00=buffer 0 (as specified by vidInAddr0); 01=buffer 1 (as specified by vidInAddr1); 10=buffer 2 (as specified by vidInAddr2); 11=No buffer is ready yet, video processor is still working on the first frame
18	Even/odd field of the frame VMI just finished drawing. 1=even, 0=odd.

11.1.18 3.1.18 vidScreenSize

NOTE: Whenever the screen resolution is changed, video processor needs to be re-enabled by clearing vidProcCfg bit 0 and setting it to 1. This will reset the video processor.

Bit	Description
11:0	Width of the screen in number of pixels. If vidScreenX is specified to be bigger than 1280, 2x mode needs to be enabled.
23:12	Height of the screen in number of lines.

11.1.19 3.1.19 vidOverlayStartCoords

Bit	Description

11:0	The x-coordinate on the screen where the upper left corner of the overlay locates.
23:12	The y-coordinate on the screen where the upper left corner of the overlay locates.
25:24	The lower two bits of the x-coordinate for the first pixel (at the upper left corner) of the overlay window with respect to the beginning of the source surface. Since the overlay window may be partially occluded by the dimension of the screen, the first pixel of the window may not necessarily be the first pixel of the source surface. The lower two bits of the x-coordinate are used for undithering, and are the same for both linear and tiled address space.
27:26	The lower two bits of the y-coordinate for the first pixel (at the upper left corner) of the overlay window with respect to the beginning of the source surface. Since the overlay window may be partially occluded by the dimension of the screen, the first pixel of the window may not necessarily be the first pixel of the source surface. The lower two bits of the y-coordinate are used for undithering, and are the same for both linear and tiled address space.
31:28	reserved

11.1.20 3.1.20 vidOverlayEndScreenCoord

Bit	Description
11:0	The x-coordinate on the screen where the lower right corner of the overlay locates.
23:12	The y-coordinate on the screen where the lower right corner of the overlay locates.

11.1.21 3.1.21 vidOverlayDudx

Bit	Description
19:0	Step size in source per horizontal step in screen space for magnification. Format is 0.20.

11.1.22 3.1.22 vidOverlayDudxOffsetSrcWidth

Bit	Description
18:0	Initial offset of Dudx. Format is 0.19.
31:19	Number of bytes needed to be fetched from the source surface in order to cover a whole un-occluded scanline for the overlay (13 bits allows a max of 8K bytes for an overlay scanline). i.e., ((Overlay width in number of screen pixels * vidOverlayDudx) + vidOverlayDudxOffset)) * overlay pixel depth in bytes. For non-scaled overlay with no offset, vidOverlayDudx becomes 1, and vidOverlayDudxOffset becomes 0 in the above equation.

11.1.23 3.1.23 vidOverlayDvdy

Bit	Description
19:0	Step size in source per vertical step in screen space for magnification. Format is 0.20.

11.1.24 3.1.24 vidOverlayDvdyOffset

Bit	Description
18:0	Initial offset of Dvdy. Format is 0.19.

Example:

Given source size of 640 x 240 and have it magnified to 1024 x 768 on the screen.

Source width:
Dudx[31:19] = 640 X 2 bytes = 1280 bytes (here 16 bpp assumed)
= 500h

Dudx[19:0] = 640/1024 = 0.625 = a0000h
(Note format is 0.20 means
x x x x x x x x x x x x x x x x x x x x
| | |
| | | .....
| | 0.125
| |
| 0.25
|
0.5)

Dvdy[19:0] = 240/768 = 0.3125 = 0.25 + 0.0625 = 50000h
(Format same as dudx above)

Dudx Offset[18:0] and Dvdy Offset [18:0] = 00000h if no initial offset is needed.
If upper leftmost overlay pixel needs to be the center of
the first pixel of the overlay surface, both offsets needs to be set to 0.5
which is 40000h.

11.1.25 3.1.25 vidDesktopStartAddr

Bit	Description
23:0	Physical starting address of the desktop surface. This is a **byte-aligned** address.

11.1.26 3.1.26 vidDesktopOverlayStride

Bit	Description
14:0	If the desktop surface resides in linear space, bit[14:0] contains the linear stride of the surface in bytes. If interlaced video output mode is enabled, the linear stride is still programmed to 1x the regular stride of the surface, and will be multiplied by 2 when used. If the desktop resides in tile space, bit[14:0] contains the tile stride of the region. This is specified in number of tiles across the width of the tile address region, NOT the width of the desktop surface.
15	reserved

For video overlay, the **stride** needs to be **a multiple of 4-bytes for YUV 422** pixel format and **a multiple of 8-bytes for YUV 411** pixel format. This ensures that the right edge of the video source surface to fall on a boundary of 2 pixels for YUV 422 and 4 pixels for YUV 411. The start address for the overlay is sampled from the FIFO'ed leftOverlayBuf and rightOverlayBuf registers. **The start address needs to be aligned on a 32-bit boundary for YUV 422 pixel format and a 64-bit boundary for YUV 411 pixel format.**

Bit	Description

30:16	If the overlay surface resides in linear space, bit[30:16] contains the linear stride of the overlay surface in bytes. If interlaced video output mode is enabled, the linear stride is still programmed to 1x the regular stride of the surface, and will be multiplied by 2 when used. If the overlay resides in tile space, bit[30:16] contains the tile stride of the region. This is specified in number of tiles across the width of the tile address region, NOT the width of the overlay surface.
31	reserved

11.1.27 3.1.27 vidInAddr0

Bit	Description
23:0	Starting address of video-in buffer 0

11.1.28 3.1.28 vidInAddr1

Bit	Description
23:0	Starting address of video-in buffer 1

11.1.29 3.1.29 vidInAddr2

Bit	Description
23:0	Starting address of video-in buffer 2

11.1.30 3.1.30 vidInStride

Bit	Description
14:0	If the video-in buffers reside in linear space, this register contains the linear stride of the buffer in bytes. If interlaced video input mode is enabled, the linear stride is still programmed to 1x the regular stride, and will be multiplied by 2 before used. If the video-in buffers reside in tile space, this register contains the tile stride of the region. This is specified in number of tiles across the width of the tile address region, NOT the width of the video-in buffers.

11.1.31 3.1.31 vidCurrOverlayStartAddr

The vidCurrOverlayStartAddr register allows the host to read the start address which the video processor is using to refresh the overlay window for the current frame.

Bit	Description
23:0	Start physical address the video processor is using to refresh the overlay window. **Read only.**

11.2 3.2 Video-In Interface

11.2.1 3.2.1 Function

Video In Processor supports several connector interfaces for video data input. The following table shows the signals needed for each interface.

11.2.2 3.2.2 Signals

Ports	DDC(I2C)	VMI*	VMI (Mode A)	VMI (Mode B)
VMI Video Port		Hsync in	Hsync in	Hsync in
VMI Video Port		Vsync in	Vsync in	Vsync in
VMI Video Port		Vactive	Vactive	Vactive
VMI Video Port		P[7:0]	P[7:0]	P[7:0]
VMI Video Port		Pixclk (in)	Pixclk (in)	Pixclk in)
VMI I2C Port		SDA (in/out)		
VMI I2C Port		SCK (in/out)		
VMI Host Port			D[7:0] (in/out)	D[7:0]
VMI Host Port			A[3:0] (out)	A[3:0]
VMI Host Port			cs_n (out)	cs_n
VMI Host Port			ds_n (out)	rd_n
VMI Host Port			r/w_n (out)	wr_n
VMI Host Port			dtack_n (in)	ready
DDC Port	SDA (in/out)			
DDC Port	SCK (in/out)			
System signals		vmi_reset_n (out)	vmi_reset_n (out)	vmi_reset_n (out)
System signals		vmi_int_n (in)	vmi_int_n	vmi_int_n
System signals		vmi_present_n (in)	vmi_present_n	vmi_present_n

A. Video-In Interface:

General Description

When video data arrives through the Video-In interface, they undergo the optional decimation and filtering, packed into words of 128 bits in a FIFO before written into the memory. As writes to the memory is always aligned on a 128-bit boundary, the appropriate byte enables also need to be set with the writes. Supported pixel formats for the video-in data are YUV422 and YUV411. Both pixel formats are stored in a form of 16 bit per pixel, which means that 4 bit are unused per pixel in the case of YUV411.

Video data are stored in the Video-In frame buffers whose starting addresses are specified by the registers VidInAddr0, VidInAddr1, and VidInAddr2. VidInAddr1 and VidInAddr2 are used for double and triple buffering to avoid video tearing. However, since video is coming in at a different rate from the video refresh, switching of the video-in drawing buffers is not synchronous to the Vsync of the video refresh. At the end of each VMI frame, the vmi_int input signal will be asserted. The video processor will then switch to the next video-in frame buffer for the next VMI frame if multiple buffering is enabled. If disabled, the same video-in frame buffer will be overwritten. At the same time, the video processor also updates the VidInStatus register which indicates the VMI buffer VMI just finishes drawing (0, 1, 2), and whether the buffer contains even or odd field. An interrupt signal will signal the host for display buffer flipping for the video-in data. On the other hand, if the “Video_in data displayed as overlay enable” bit in VidProcCfg is set, the video porcessor will do the display buffer flipping automatically for the overlay provided that all

the corresponding configuration registers for the overlay is set up correctly (e.g., overlay surface enable, overlay pixel format, overlay_dudx, …… etc).

If Weave video-in deinterlaced mode is enabled, the video processor detects even/odd field from VREF(Vsync) and HREF(Hsync). If odd, the specified VidInAddr register will be used as the starting address of the video-in frame buffer. If even, VidInStride will be used as the starting address offset, and added to the specified VidInAddr. Video-in buffer will be switched at every other Vsync. VidInStride should be programmed to contain the a value which equals to 1X the regular line stride regardless of whether the video-in data is interlaced or not.

1. VMI

-data:

8-bit YCbCr interface is used. The data format is CCIR-656 YCbCr 422, and pixels arrive in the style of (Cb0[7:0] or U0[7:0]) -> Y0[7:0] -> (Cr0[7:0] or V0[7:0]) -> Y1[7:0].

Video data may be interlaced.

-timing:

Timing signals include VREF, HREF, VACTIVE, and PIXCLOCK.

VREF and HREF are active high VSYNC and HSYNC. If HREF is high during the falling edge of VREF, the field is even. If HREF is low at that time, the field is odd.

VACTIVE is a blanking signal which indicates pixel data is valid across the YCbCr bus.

11.3 3.3 Video Limitation

1. In 1x mode, 3 streams of pixel fetching will consume more memory bandwidth than available for 32-bit desktop. This means chroma-keying and bilinear filtering cannot be turned on simultaneously for 32-bit desktop.

2. In 2x mode (for any display larger than 1280 X 1024) where we refresh 2 screen pixels per cycle at 110MHz, bilinear filtering is not supported. All backend zoom (magnification) is done by point sampling (replication).

3. 1 - 10X backend zoom (magnification) with increments of 0.1X. Larger magnification is supported, but with bigger increments.

   1 to 1/16X video-in decimation (minimization) with increments of 0.015X.

4. Retain the 3-bit tap filter for RGB565 dithered as an alternative to the 2x2 box filter.

5. Interlaced video output mode is not implemented.

6. Hw cursor is 2 color only.
7. YUV 411 pixel format will be stored as unpacked in the frame buffer. This means each pixel will occupy 16 bits instead of 12 bits. This makes pixel extraction easier, but consumes more memory.

8. Video with YUV 422 format needs to be stored on a 4-byte memory boundary while YUV 411 on a 8-byte boundary. This is necessary because UV are shared between 2 pixels in 422 while UV are shared between 4 pixels in 411.

12. Command Transport Protocol

12.1 Command Transport

A command FIFO (CMDFIFO) may be established by software within frame buffer memory or AGP memory. Writes to the linear frame buffer address space are performed to build a command buffer, which is then parsed and executed by the accelerator. To accommodate host CPUs which may issue writes out-of-order (eg. Intel's Pentium Pro), one of two scenarios will occur, the CMDFIFO resides in AGP (non-local video memory) and software manages the accelerator's internal CMDFIFO depth register, or the CMDFIFO resides in frame buffer memory and the accelerator manages the internal CMDFIFO depth register.

If the CMDFIFO resides in AGP space (non-local video memory), software "BUMPS" the internal CMDFIFO depth register after N words into the AGP buffer. This allows the CPU to write to the CMDFIFO in any order, flush any pending writes in the CPU's internal write buffers and core logic chipset's internal write buffers, then update the accelerator's depth register. Since writes to the CMDFIFO will be in consecutive order, the CPU's write buffers will fill and burst into memory more efficiently, than random PCI writes.

If the CMDFIFO resides in frame buffer memory, software writes to the frame buffer in consecutive order, the CPU flushes its write buffer in any order to the accelerator. The accelerator counts the number of non written addresses, once consecutive addresses are written, the internal CMDFIFO depth register is updated to the last consecutive written address. Counting unwritten addresses allows the CPU to flush its internal write buffers in any order, but maintains the correct order in the frame buffer memory. Software must manage the circular buffer at the point where the buffer recycles to the beginning. This is done by placing a JMP instruction (CMDFIFO Packet Type 0, Func 100) at the bottom of the fifo to restart at the beginning of the CMDFIFO space.

12.1.1 CMDFIFO Management

The CMDFIFO mechanism supports 2 types of fifo management, software and hardware. When the CMDFIFO is located in frame buffer memory either software management or hardware management can be used on the CMDFIFO, unlike AGP which only supports software management of the CMDFIFO.

12.1.1.1 Software Management of CMDFIFO

Software manages the CMDFIFO "emptiness." The accelerator maintains a read pointer and a depth for the CMDFIFO. Accelerator reads from the CMDFIFO decrement the depth register and increment the read pointer. The accelerator will automatically execute data from the CMDFIFO as long as the internal CMDFIFO depth register is greater than zero. When the CPU is ready to inform the accelerator that more data is available in the CMDFIFO, the CPU writes the number of 32-bit words that have been added to the end of the CMDFIFO. The accelerator then adds the value written by the CPU to the internal depth register. The accelerator's internal registers define where the circular CMDFIFO exists in frame buffer memory by defining a beginning address for the CMDFIFO and a rollover address. By default, the CMDFIFO internal

read pointer is set to the beginning address for the CMDFIFO. Once data is stored in the CMDFIFO (and the internal depth register is incremented by the CPU), the CMDFIFO read pointer will increment as the accelerator parses and executes the CMDFIFO. When the read pointer equals the rollover address defined by initialization registers, the read pointer will jump back to the beginning CMDFIFO address. The CMDFIFO is thus programmable in size as a circular space from 1 to N 4k byte pages. Software must manage CMDFIFO "fullness" and guarantee that the CMDFIFO does not overflow. On systems like the Intel Pentium Pro, software must place a fence after the last memory write, but before the write to increase the number of new entries in the CMDFIFO.

12.1.1.2 Hardware Management of CMDFIFO

Hardware manages the CMDFIFO "emptiness." The accelerator maintains a read pointer, write pointer, and depth for the CMDFIFO. Accelerator reads from the CMDFIFO decrement the depth register and increment the read pointer. The accelerator will automatically execute data from the CMDFIFO as long as the internal CMDFIFO depth register is greater than zero. The CPU writes data into the CMDFIFO area in sequential addresses. The accelerator snoops the writes into the CMDFIFO area and examines the addresses, looking for non sequential addresses or "holes." When the accelerator gathers sequential addresses present in the CMDFIFO, the depth and write pointers are incremented. The accelerator's internal registers define where the circular CMDFIFO exists in frame buffer memory by defining a beginning address for the CMDFIFO and a rollover address. By default, the CMDFIFO internal read pointer is set to the beginning address for the CMDFIFO. Once data is stored in the CMDFIFO (and the internal depth register is incremented by the CPU), the CMDFIFO read pointer will increment as the accelerator parses and executes the CMDFIFO. When the read pointer equals the rollover address defined by initialization registers, the read pointer will jump back to the beginning CMDFIFO address. The CMDFIFO is thus programmable in size as a circular space from 1 to N 4k byte pages. Software must manage CMDFIFO "fullness" and guarantee that the CMDFIFO does not overflow. On systems like the Intel Pentium Pro, software must place a fence after the last memory write, but before the first write to the top of the CMDFIFO. Software may not write less than four 32 bit entries before performing a jump to the begining of the buffer.

12.1.2 CMDFIFO Data

All CMDFIFO data packets begin with a 32-bit packet header which defines the data which follows. There are 6 different types of CMDFIFO packet headers. Bits (2:0) of a CMDFIFO packet header define the packet header type. All CMDFIFO packet headers and data must be 32-bit words - byte and 16-bit short writes are not allowed in the CMDFIFO.

12.1.3 CMDFIFO Packet Type 0

CMDFIFO Packet Type 0 is a variable length packet, requiring a minimum single 32-bit word, to a maximum of 2 32-bit words. CMDFIFO Packet Type 0 is used to jump to the beginning of the fifo when the end of the fifo is reached. CMDFIFO Packet Type 0 also supports jumping to a secondary command stream just like a jump subroutine call (**jsr** instruction), with a CMDFIFO Packet that instructs a return as well. NOP, JSR, RET, and JMP LOCAL FRAME BUFFER functions only require a single 32-bit word CMDFIFO packet, while the JMP AGP function requires a two 32-bit word CMDFIFO packet. Bits 31:29 are reserved and must be written with 0.

CMDFIFO Packet Type 0

	31 29	28 6	5 3	2 0
word 0	Reserv	Address [24:2]	Func	000
word 1	reserved	Address [35:25]		

Code	Function
000	NOP
001	JSR
010	RET
011	JMP LOCAL FRAME BUFFER
100	JMP AGP

12.1.4 CMDFIFO Packet Type 1

CMDFIFO Packet Type 1 is a variable length packet that allows writes to either a common address, or to consecutive addresses, minimum number of words is 2 32-bit words, and maximum number of words is 65536 words. Bits 31:16 define the number of words that follow word 0 of packet type 1, and must be greater than 0. When bit 15 is a 1, data following word 0 in the packet is written in consecutive addresses starting from the register base address defined in bits 14:3. When bit 15 is a 0, data following word 0 is written to the base address. Packet header bits 14:3 define the base address of the packet, see section below. The common use of packet type 1 is host blits.

CMDFIFO Packet Type 1

	31 16	15	14 3	2 0
word 0	Number of words	incc	Register Base (See below)	001
word 1	Data			
word N	Optional Data N			

Register Base:

Avenger

11	10	9	7 0
2D/3D	reserv	chip fiel	Register Number

If bit 11 is set, then the register base is a 2D register, otherwise it is a 3D register.

CVG

11 8	7 0
Chip field	Register Number

12.1.5 CMDFIFO Packet Type 2

CMDFIFO Packet Type 2 is a variable length packet, requiring a minimum of 2 32-bit words, and a maximum of 30 32-bit words for the complete packet. The base address for CMDFIFO Packet Type 2 is defined to be offset 8 of the hardware 2D registers(clip0min). The first 32-bit word of the packet defines individual write enables for up to 29 data words to follow. From LSB o MSB of the mask, a “1” enables the write and a “0” disables the write. The sequence of up to 29 32-bit data words following the mask modify addresses equal to the implied base address plus N where mask[N] equals “1” as N ranges from 0 to 28. The total number of 32-bit data words following the mask is equal to the number of “1”s in the mask. The register mask must not be 0.

CMDFIFO Packet Type 2

	31 3	2 0
word 0	2D Register mask	010

word 1	Data
word N	Optional Data N

12.1.6 CMDFIFO Packet Type 3

CMDFIFO Packet Type 3 is a variable length packet, requiring a minimum of 3 32-bit words, and a maximum of 16 vertex data groups, where a data group is all the register writes specified in the parameter mask, for the complete packet. It is a requirement that bits 9:6 must be greater than 0. The base address for CMDFIFO Packet Type 3 is defined to be the starting address of the hardware triangle setup registers. The first 32-bit word of the packet defines 16 individual vertex data. Bits 31:29 of word 0 define 0 to 7 dummy fifo entries following the packet type 3 data. The **sSetupMode** register is written with the data in bits 27:10 of word 0. Bits 9:6 define the number of vertex writes contained in the packet, where the total packet size becomes what is defined in the parameter mask multiplied by the number of vertices. During parsing and execution of a CMDFIFO Packet Type 3, a specific action takes place based on bits 5:3. The **sSetupMode** register implies that X and Y are present in words 1 and 2. When Bit 28 when set, packed color data follows the X and Y values, otherwise independent red, green, blue, and alpha follow X and Y data. When Smode field is 0, then word 0 defines X, and word 1 defines Y.

Code 000 specifies an independent triangle packet, where an implied **sBeginTriCMD** is written after 2 **sDrawTriCMD's**. The sequence would follow, **sBeginTriCMD, sDrawTriCMD, sDrawTriCMD**, **sBeginTriCMD**, until "NumVertex" vertices has been executed.

Code 001 specifies the beginning of a triangle strip, an implicit write to **sBeginTriCMD** is issued, followed by Num Vertex **sDrawTriCMD** writes. The sequence would follow, **sBeginTriCMD, sDrawTriCMD, sDrawTriCMD, sDrawTriCMD,** until "num Vertex" vertices has been executed

Code 010 specifies the a continuance of an existing triangle strip, an implicit write to **sDrawTriCMD** is performed after one complete vertex has been parsed.

CMDFIFO Packet Type 3

	31 29	28	27 22	21 10	9 6	5 3	2 0
word 0	Num	PC	SMode	Parameter Mask	Num Vertex	CMD	011
word 1	Data						
word N	Optional Data N						

Code	Command
000	Independent Triangle
001	Start new triangle strip
010	Continue existing triangle strip
011	reserved
1xx	reserved

Bit	Description
	sParamMask field
10	Setup Red, Green, and Blue
11	Setup Alpha
12	Setup Z
13	Setup Wb
14	Setup W0
15	Setup S0 and T0

16	Setup W1
17	Setup S1 and T1
	sSetupMode field
22	Strip mode (0=strip, 1=fan)
23	Enable Culling (0=disable, 1=enable)
24	Culling Sign (0=positive sign, 1=negative sign)
25	Disable ping pong sign correction duri triangle strips (0=normal, 1=disable)

Parameter

word 1	X
word 2	Y
word 3	Red / Packed ARGB (optional)
word 4	Green (optional)
word 5	Blue (optional)
word 6	Alpha (optional)
word 7	Z (optional)
word 8	Wbroadcast (optional)
word 9	W0 Tmu 0 & Tmu1 W (optional)
word 10	S0 Tmu0 & Tmu1 S (optional)
word 11	T0 Tmu0 & Tmu1 T (optional)
word 12	W1 Tmu 1 W (optional)
word 13	S1 Tmu1 S (optional)
word 14	T1 Tmu1 T (optional)

Sequence of implied commands for Each code follows:

M = Mode register write

B = sBeginTriCMD

D = sDrawTriCMD

Code 000: MBDDBDDBDDBDD …

Code 001: MBDDDDDDDDDDD …

Code 010: MDDDDDDDDDDDD …

12.1.7 CMDFIFO Packet Type 4

CMDFIFO Packet Type 4 is a variable length packet, requiring a minimum of 2 32-bit words, and a maximum of 22 32-bit words for the complete packet. The first 3 bits 31:29 of word 0 define the number of pad words that follow the packet type 4 data. The next 14 bits of the header 28:15 define the register write mask, followed by the register base field, described later in this section. From LSB to MSB of the mask, a “1” write. The sequence of up to 22 32-bit data words following the mask modify addresses equal to the implied base address plus N where mask[N] equals “1” as N ranges from 0 to 16. The total number of 32-bit data words following the mask is equal to the number of “1”s in the mask. As a requirement, the general register mask must have a non zero value

CMDFIFO Packet Type 4

	31 29	28 15	14 3	2 0
word 0	num	General Register mask	Register Base (See below)	100
word 1	Data			
word N	Optional Data N			

Register base:

Avenger

11	10	9 8	7 0
2D/3D	Reserv	chip field	Register Number

Bit 11 denotes either 2D or 3D, when set register Number is a 2D number, otherwise Register number is a 3D register.

CVG

11 8	7 0
Chip field	Register Number

12.1.8 CMDFIFO Packet Type 5

CMDFIFO Packet Type 5 is a variable length packet, requiring a minimum of 3 32-bit words, and a maximum of 2^19 32-bit words for the complete packet Bits 31:30 define LFB type, one of linear frame buffer, planar YUV space, 3D LFB, or texture download space. Bits 29:26 in word 0 define the byte enables for word 2, and are active low true. Bits 25:22 in word 0 define the byte enables for word N. Data must be in the correct data lane, and the base address must be 32-bit aligned. CMDFIFO Packet Type 5 is used to transfer large consecutive quantities of data from the CPU to the accelerator's frame buffer with proper order with the command stream. One note, transfer to tile space is limited if tile-stride does not match PCI stride. Tile space rows are not continuous, thus each tile row must be separated into separate packets.

NOTE: when downloading into the texture download space, the aperture is 4MB wide. The lower 2M of this space is sent into the TMU0 download port, while the upper 2M is sent to the TMU1 download port. Downloads to either TMU are not guaranteed to be synchronous with each other. Please refer to the "maintaining read-reordering memory coherency" section of this spec.

Downloads to TMU0 should have a **BaseAddress[24:0]** starting at 0x000000, while downloads to TMU1 should have a **BaseAddress[24:0]** of 0x200000. When the hardware receives a CMDFIFO Packet 5 command to download into texture space, the value 0x600000 is added to the downloads generated in order to normalize the downloads to **Avenger Address Space**.

CMDFIFO Packet Type 5

	31 30	29 26	25 22	21 3	2 0
word 0	Space	Byte Enable W2	Byte Enable WN	Num Words	101
word 1	reserv		BaseAddress [24:0]		
word 2	Data				
word N	Optional Data N				

Code	Space
00	Linear FB
01	Planar YUV
10	3D LFB
11	Texture Port

12.1.9 CMDFIFO Packet Type 6

CMDFIFO Packet Type 6 is a fixed length packet, requiring 5 32-bit words for the complete packet. CMDFIFO Packet Type 6 is primarily used to transfer data from system AGP memory into frame buffer local memory. Bits 20:5 of word 0 define the transfer size in bytes of an AGP transfer. Bits 4:3 define the destination memory space LFB, Planar YUV, 3D LFB, and texture port. Word 1 bits 31:0, and word 2 bits 27:24 define the source system AGP address of the data move. Bits 23:12 define the stride, and bits 11:0 define the width of the source surface in AGP memory. Word 3 defines the destination frame buffer address, while word 4 bits 11:0 define the stride of the destination surface.

CMDFIFO Packet Type 6

	31 26	25 5	4 3	2 0
word 0	reserved	Number of transfer bytes [19:0]	type	110
word 1	AGP[31:0]			
word 2	AGP[35:32]	AGP stride [13:0]	width [13:0]	
word 3	reserved	Frame buffer offset [25:0]		
word 4	reseved	Destination stride [14:0]		

type	Space
00	Linear FB
01	Planar YUV
10	3D LFB
11	Texture Port

12.1.10 Miscellaneous

Avenger supports two full CMDFIFO streams and each individually can be located in frame buffer memory or AGP space. Each CMDFIFO has it's own base address register set, that define the starting address, memory space, and size of the CMDFIFO. The CMDFIFO registers contain a write only bump register that increments the write pointer by the amount written to the **cmdBump** register. Each CMDFIFO contains a read pointer, write pointer, and freespace count of the fifo itself, so the CPU can monitor the progress and fullness of the CMDFIFO. Ordering between the two CMDFIFO's is first come, first served.

13. AGP/CMD Transfer/Misc Registers

Memory Base 0: Offset 0x0080000

Register Name	Address	Bits	R/W	Description
AGP				
agpReqSize	0x000(0)	19:0	R/W	Size of AGP request
agpHostAddressLow	0x004(4)	31:0	R/W	Host address bits 31:0
agpHostAddressHigh	0x008(8)	31:0	R/W	Host address bits 35:32
agpGraphicsAddress	0x00C(12)	24:0	R/W	Graphics address bits 24:0

agpGraphicsStride	0x010(16)	16:0	R/W	Graphics stride
agpMoveCMD	0x014(20)	n/a	W	Begin AGP transaction
CMDFIFO 0				
cmdBaseAddr0	0x020(32)	23:0	R/W	Base Address of CMDFIFO 0
cmdBaseSize0	0x024(36)	10:0	R/W	CMDFIFO0 size
cmdBump0	0x028(40)	15:0	W	Bump CMDFIFO 0 N words
cmdRdPtrL0	0x02c(44)	31:0	R/W	CMDFIFO 0 read pointer lower 32 bits
cmdRdPtrH0	0x030(48)	3:0	R/W	CMDFIFO 0 read pointer upper 4 bits
cmdAMin0	0x034(52)	24:0	R/W	CMDFIFO 0 address min pointer
	0x038			Reserved
cmdAMax0	0x03c(60)	24:0	R/W	CMDFIFO 0 address max pointer
cmdStatus0	0x040(64)	31:0	R	Status register (debug & fault coverage)
cmdFifoDepth0	0x044(68)	19:0	R/W	CMDFIFO 0 depth value
cmdHoleCnt0	0x048(72)	15:0	R/W	CMDFIFO 0 outstanding CPU writes
CMDFIFO 1				
cmdBaseAddr1	0x050(80)	23:0	R/W	Base Address of CMDFIFO 1
cmdBaseSize1	0x054(84)	10:0	R/W	CMDFIFO1 size
cmdBump1	0x058(88)	15:0	W	Bump CMDFIFO 1 N words
cmdRdPtrL1	0x05c(92)	31:0	R/W	CMDFIFO 1 read pointer lower 32 bits
cmdRdPtrH1	0x060(96)	3:0	R/W	CMDFIFO 1 read pointer upper 4 bits
cmdAMin1	0x064(100)	24:0	R/W	CMDFIFO 1 address min pointer
	0x068(104)			Reserved
cmdAMax1	0x06c(108)	24:0	R/W	CMDFIFO 1 address max pointer
cmdStatus1	0x070(112)	31:0	R	Status register (debug & fault coverage)
cmdFifoDepth1	0x074(116)	19:0	R/W	CMDFIFO 1 freespace
cmdHoleCnt1	0x078(120)	15:0	R/W	CMDFIFO 1 outstanding CPU writes
cmdFifoThresh	0x080(96)	4:0	R/W	CMDFIFO fetch threshold
cmdHoleInt	0x084(100)	22:0	R/W	Cmd hole timeout value
Misc				
yuvBaseAddress	0x100(256)	25:0	R/W	YUV planar base address
yuvStride	0x104(260)	12:0	R/W	Y, U and V planes stride value

13.1 agpReqSize

agpReqSize defines the AGP packet transfer size. The maximum transfer size is 4-Mbyte block of data. This register is read write and has no default value.

Bit	Description
19:0	reserved. Default is 0x0.

13.2 agpHostAddressLow

During AGP transfers this address defines the source address bits 31:0 of AGP memory to fetch data from. AGP addresses are 36-bits in length and are byte aligned. The upper 4 bits reside in the **agpHostAddressHigh** register. This register is read write, and defaults to 0.

Bit	Description
31:0	Lower 32 bits of AGP memory. Default is 0x0.

13.3 agpHostAddressHigh

The **agpHostAddressHigh** defines the stride, width, and upper 4-bits of source AGP address, during AGP transfers. Stride and width are defined in quadwords. This register is read write, and defaults to 0.

Bit	Description
13:0	AGP Width. Default is 0x0.
27:14	AGP Stride. Default is 0x0.
31:28	Upper 4 bits of AGP memory. Default is 0x0.

13.4 agpGraphicsAddress

agpgraphicsAddress defines the destination frame buffer address and type of the AGP transfer. At the beginning of an AGP transfer this address is loaded into an internal address pointer that increments for each data received over AGP. This register is read write, and defaults to 0.

Bit	Description
25:0	Frame buffer offset. Default is 0x0.

13.5 agpGraphicsStride

agpGraphicsStride defines the destination stride in bytes of the AGP transfer. Stride is in multiples of bytes. This register is read write, and defaults to 0.

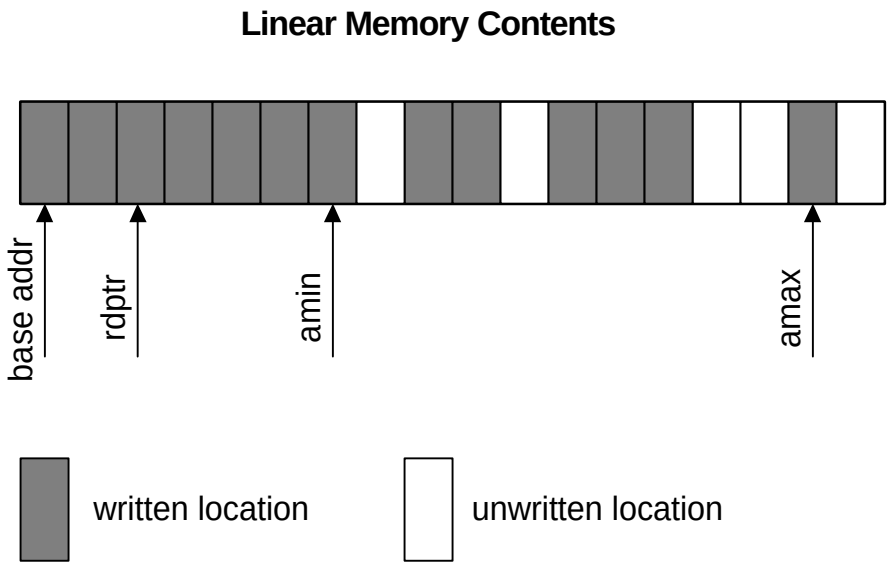
Bit	Description
14:0	Frame buffer Stride. Default is 0x0.

13.6 agpMoveCMD

agpMoveCMD starts an AGP transfer. When started **agpHostAddress** is loaded into the source pointer and **agpGraphicsAddress** is loaded into the destination pointer. The source pointer is incremented after data is fetched from AGP memory and written into frame buffer memory addresses by the destination pointer. The destination pointer is then incremented after the data has been written. This register is write only and has no default.

Bit	Description
2:0	reserved
4:3	Dest memory type (0=Linear FB, 1=planar YUV, 2=3D LFB, 3 = texture port). Default is 0x0.
5	Command stream ID. This bit defines which command fifo when using a host initiated AGP data move. Default is 0x0.

14. Command Fifo Registers



The command registers define the location, size, and fifo management method of the command fifo. The command fifo starts at the address defined in the **cmdBaseAddr**[01] register and occupies N 4k byte pages defined in the **cmdBaseSize** register. The command fifo can be located either in AGP or frame buffer memory which is defined in the **cmdBaseSize** register. **CmdRdPtr** points to the last executed entry in the command fifo. Amin is a pointer that walks through the fifo until it reaches an unwritten location. The rdptr can not access any entry beyond the amin pointer. The amax pointer is set to the furthest address location of a given write. The hole counter is basically the number of unwritten locations between the amax register and the amin register. When the hole counter is zero, the amin register is set to the value of the amax register, thus allowing the read pointer to advance to the new amin register value. The depth of the fifo is calculated by the difference between amax and rdptr.

Or, put another way (from the perspective of a driver writer):

When hole counting is enabled (hardware manages command fifo depth), the memory controller takes special action whenever a write occurs between the command fifo base and the base + size. As writes occur in this region, five variables are fiddled: readPtr, depth, aMin, aMax, and holeCount. As ordered writes happen, both aMin and aMax increment, as does depth and readPtr. In this state, the difference between aMin/aMax and the readPtr is the depth. When the depth is nonzero, the readPtr advances as

commands are read from the buffer. When/if an out-of-order write occurs, aMin stops incrementing, but aMax continues to increment as addresses written go up. The readPtr will not pass aMin, so the depth begins to decrement. Once the readPtr has caught up with aMin, the depth sits at zero. If aMax ever has to skip (due to an out-of-order write), the hole count is incremented. As out-of-order data gets written between aMin and aMax, the hole count is decremented. When the holeCount goes to zero, the difference between aMin and aMax is added to the depth, and aMin is set to be the same as aMax. This causes command processing to resume.

14.1 cmdBaseAddr0

CMDFIFO 0's base address pointer bits 23:0. **CmdBaseAddr0** contains either the entire frame buffer address of the start of CMDFIFO, or contains the AGP address of the start of CMDFIFO. This register is read write, and has no default.

Bit	Description
23:0	24-bits of CMDFIFO address [23:0] in 4k byte pages. Default is 0x0.

14.2 cmdBaseSize0

cmdBaseSize0 contains the size of the command fifo in bits 7:0 in 4k byte pages, starting from 4k. Bit 8 enables or disables command fifo 0 operation. Bit 9 defines the location of command fifo 0, a value of 0 locates the command fifo in frame buffer memory, and value of 1 locates the command fifo in AGP memory. Bit 10 disables the hole counter.

Bit	Description
7:0	Size of CmdFifo in 4k byte pages. (0=4k, 1 = 8k, etc…). Default is 0x0.
8	CMDFIFO_0 enable (0=disable, 1=enable). Default is 0x0.
9	CMDFIFO_0 resides in AGP (0=frame buffer memory, 1=AGP memory). Default is 0x0.
10	Disable hole counter (0=enable, 1=disable). Default is 0x0.

14.3 cmdBump0

cmdBump0 defines the number of words to increment the amin pointer by, when managed by software. This register is write only and has no default.

Bit	Description
15:0	Number of words to bump CMDFIFO 0's write pointer. Default is 0x0.

14.4 cmdRdPtrL0

cmdRdPtrL0 contains the lower 32-bits of the read pointer. This register is read / write and allows software to monitor the progress of the CMDFIFO. This register is read write and has no default value. At initialization, this register should be set to **cmdBaseAddr** ,expanded to a byte address.

Bit	Description
31:0	Lower 32-bits of the byte aligned CMDFIFO read pointer. Default is 0x0.

14.5 cmdRdPtrH0

cmdRdPtrH0 contains the upper 4-bits of the read pointer. This register is read write and has no default value. At initialization, this register should be set to **cmdBaseAddr**, expanded to a byte address.

Bit	Description
3:0	Upper 4-bits of the CMDFIFO read pointer. Default is 0x0.

14.6 cmdAMin0

cmdAMin0 is a 25-bit register containing the min address register. **CmdAMin** register is updated with the **cmdAMax** register when hole count is zero. This register is read write and has no default value. At initialization this register should be set to **cmdBaseAddr** - 4. The value read back from this register is 4 more than that written.

Bit	Description
24:0	Byte Aligned Address Min register, bits 0 and 1 are ignored. Default is 0x0.

14.7 cmdAMax0

cmdAMaxL0 is a 32-bit register containing the 25 bits of the max address register. **CmdAMax** register is automatically updated when an memory address greater to the existing **cmdAMax** register is written. At initialization, this register should be set to **cmdBaseAddr** - 4. The value read back from this register is 4 more than that written.

Bit	Description
24:0	Byte Aligned Address Max register, bits 0 and 1 are ignored. Default is 0x0.

14.8 cmdStatus0

cmdStatus0 is a 32 bit register that allows debuging and visability of the command fifo hardware. This register is read only, and has no impact on software development.

Bit	Description
31	AGP (packet 6) data xfer in progress
30	Unpacker is busy
29	Packet 6 decompress busy
28	Host data transfers are complete
27	JSR (fetch ctrl) active
26	Executed depth = 0
25	Prefetched depth = 0
24	On chip fifo is empty
23:17	Reserved. Default = 0
16	JSR (unpacker) active
15	Jump tag
14:12	Jump command.
11	Header Valid
10:6	Local State (remaining entries in packet)
5:3	Etended Packet Command
2:0	Packet Type

14.9 cmdFifoDepth0

cmdFifoDepth0 is a 20-bit register containing the current depth of CMDFIFO 0. Depth is the number of remaining unexecuted commands in off chip memory. The CMDFIFO is allowed to read upto, but not over the number of entries indicated by **cmdFifoDepth** register. This register is read write and has no default value.

Bit	Description
19:0	CMDFIFO 0 depth. Default is 0x0.

14.10 cmdHoleCnt0

cmdHoleCnt0 contains the number of unwritten locations between **cmdAMin** and **cmdAMax**.

Bit	Description
15:0	CMDFIFO Hole counter. Default is 0x0.

14.11 cmdBaseAddr1

cmdBaseAddrL1 is similar to **cmdBaseAddr0**, but controls CMDFIFO 1.

Bit	Description
23:0	Lower 23-bits of CMDFIFO address [31:0] in 4k pages. Default is 0x0.

14.12 cmdBaseSize1

cmdBaseSize1 is similar to **cmdBaseAddr0**, but controls CMDFIFO 1.

Bit	Description
7:0	Size of CmdFifo in 4k byte pages. Default is 0x0.
8	CMDFIFO_1 enable (0=disable, 1=enable). Default is 0x0.
9	CMDFIFO_1 resides in AGP (0=frame buffer memory, 1=AGP memory). Default is 0x0.
10	Disable hole counter (0=enable, 1=disable). Default is 0x0.

14.13 cmdBump1

cmdBump1 is similar to **cmdBump0**.

Bit	Description
15:0	Number of words to bump CMDFIFO 1's write pointer. Default is 0x0.

14.14 cmdRdPtrL1

cmdRdPtrL1 is similar to **cmdRdPtrL0**.

Bit	Description
31:0	Lower 32-bits of the CMDFIFO read pointer. Default is 0x0.

14.15 cmdRdPtrH1

cmdRdPtrH1 is similar to **cmdRdPtrH1**.

Bit	Description
3:0	Upper 4-bits of the CMDFIFO read pointer. Default is 0x0.

14.16 cmdAMin1

cmdAMin1 is similar to **cmdAMin0**

Bit	Description
24:0	Byte Aligned Address Min register for command stream 1. Default is 0x0.

14.17 cmdAMax1

cmdAMax1 is similar to **cmdAMax0**

Bit	Description
24:0	Byte Aligned Address Max register for command stream 1. Default is 0x0.

14.18 cmdStatus1

cmdStatus1 is identical to **cmdStatus0**, but is used for the second command fifo.

14.19 cmdFifoDepth1

cmdFifoDepth1 is similar to **cmdFifoDepth0**

Bit	Description
19:0	CMDFIFO 1 depth in dwords. . Default is 0x0.

14.20 cmdHoleCnt1

cmdHoleCnt1 is similar to **cmdHoleCnt0**

Bit	Description
15:0	CMDFIFO 1's hole counter. Default is 0x0.

14.21 cmdFifoThresh

cmdFifoThresh bits 4:0 contain the fetch threshold, used when the fifo freespace is below fetch threshold, then no new requests are made. Bits 8:5 contain the fifo high water mark, when fifo freespace is above the water mark, then fill requests will be generated. When the high water mark is qualified, then new requests are generated.

Bit	Description
4:0	CMDFIFO 0 and 1's fifo fetch threshold (low water mark) (Default value is 0).
8:5	CMDFIFO 0 and 1's fifo high water mark (Default value is 7).
21:9	reserved

14.22 cmdHoleInt

cmdHoleInt bits 21:0 contain the number of MCLK cycles a hole counter can have a hole before genreating an interrupt. The counter is only enabled when bit 22 of this register is set. This register should be used in combination with the **IntrCtrl** register to product PCI interrupts for flagging insufficient data.

Bit	Description
21:0	CMDFIFO 0 and 1 (holes !=0) time out value. Default is 0x0.
22	CMDFIFO Time Out Counter Enable. (0=Disable, 1 = Enable). Default is 0x0.

14.23 yuvBaseAddress

yuvBaseAddress register contains the starting frame buffer location of the yuv aperture.

Bit	Description
24:0	YUV base Address. Default is 0x0.

14.24 yuvStride

yuvStride register contains the destination stride value of the U and V planes.

Bit	Description
13:0	Y, U and V stride register. Default is 0x0.
30:14	reserved
31	Destination is tiled (0 = linear, 1 = tiled) . Default is 0x0.

15. AGP/PCI Configuration Register Set

Register Name	Addr	Bits	Description
Vendor_ID	0	15:0	3Dfx Interactive Vendor Identification
Device_ID	2	15:0	Device Identification
Command	4	15:0	PCI bus configuration
Status	6	15:0	PCI device status
Revision_ID	8	7:0	Revision Identification
Class_code	9	23:0	Generic functional description of PCI device
Cache_line_size	12	7:0	Bus Master Cache Line Size
Latency_timer	13	7:0	Bus Master Latency Timer
Header_type	14	7:0	PCI Header Type
BIST	15	7:0	Build In Self-Test Configuration
memBaseAddr0	16	31:0	Memory Base Address (Init/3D/2D regs)
memBaseAddr1	20	31:0	Memory Base Address (LFB)
ioBaseAddr	24	31:0	I/O Base Address
Reserved	28-43		Reserved
subVendorID	44-45	15:0	Subsystem Vendor ID
subSystemID	46-47	15:0	Subsystem ID
romBaseAddr	48	31:0	Expansion Rom Base Address
Capabilites Ptr	52	31:0	Pointer to start of New Capabilities structure [7:0]
Reserved	56-59		Reserved
Interrupt_line	60	7:0	Interrupt Mapping
Interrupt_pin	61	7:0	External Interrupt Connections
Min_gnt	62	7:0	Bus Master Minimum Grant Time
Max_lat	63	7:0	Bus Master Maximum Latency Time
fabID	64	2:0	Fab Identification
Reserved	68-75		Reserved
cfgStatus	76	31:0	Aliased memory-mapped **status** register
cfgScratch	80	31:0	Scratch pad register
AGP Cap_ID	84	31:0	AGP Capability identifier register (read only)
AGP status	88	31:0	AGP status register (read only)
AGP_Cmd	92	31:0	AGP command register (read / write)
ACPI Cap ID	96	31:0	ACPI Capability identifier register (read only)
ACPI cntrl/status	100	31:0	ACPI Control and Status register (read / write)
Reserved	92-255	n/a	Reserved

15.1 Vendor_ID Register

The **Vendor_ID** register is used to identify the manufacturer of the PCI device. This value is assigned by a central authority that will control issuance of the values. This register is read only.

Bit	Description
7:0	3Dfx Interactive Vendor Identification. Default is 0x121a.

15.2 Device_ID Register

The **Device_ID** register is used to identify the particular device for a given manufacturer. This register is read only.

Bit	Description
15:0	Avenger Device Identification. Default is 0x3.

15.3 Command Register

The **Command** register is used to control basic PCI bus accesses. See the PCI specification for more information. Bit 0,1 and 5 are R/W, and bits 15:6 and 4:2 are read only.

Bit	Description
0	I/O Access Enable. Default is 0.
1	Memory Access Enable (0=no response to memory cycles). Default is 0.
2	Master Enable. Default is 0.
3	Special Cycle Recognition. Default is 0.
4	Memory Write and Invalidate Enable. Default is 0.
5	Palette Snoop Enable. Default is 0.
6	Parity Error Respond Enable. Default is 0.
7	Wait Cycle Enable. Default is 0. (strapped)
8	System Error Enable. Default is 0.
15:9	reserved. Default is 0x0.

15.4 Status Register

The **Status** register is used to monitor the status of PCI bus-related events. This register is read only and is hardwired to the value 0x0.

Bit	Description
3:0	Reserved. Default is 0x0.
4	New Capabilities (AGP/ACPI). Default is 1 for AGP/ACPI (Strapped)
5	66 Mhz Capable. Default is 0 for PCI 33 Mhz 1 for AGP (Strapped)
6	UDF supported. Default is 0.
7	Fast Back-toBack capable. Default is 0. (Strapped)
8	Data Parity Reported. Default is 0.
10:9	Device Select Timing. Default is 0x0.
11	Signaled Target Abort. Default is 0.
12	Received Target Abort. Default is 0.
13	Received Master Abort. Default is 0.
14	Signaled System Error. Default is 0.
15	Detected Parity Error. Default is 0. Cleared by writing this register. This feature is used for detecting parity errors on bus transfers.

15.5 Revision_ID Register

The **Revision_ID** register is used to identify the revision number of the PCI device. This register is read only.

Bit	Description
7:0	Avenger Revision Identification. (0=na, 1=.35u, 2 = .25u)

15.6 Class_code Register

The **Class_code** register is used to identify the generic functionality of the PCI device. See the PCI specification for more information. This register is read only.

Bit	Description
23:0	Class Code. Default is 0x3.

15.7 Cache_line_size Register

The **Cache_line_size** register specifies the system cache line size in doubleword increments. It must be implemented by devices capable of bus mastering. This register is read only and is hardwired to 0x0.

Bit	Description
7:0	Cache Line Size. Default is 0x0.

15.8 Latency_timer Register

The **Latency_timer** register specifies the latency of bus master timeouts. It must be implemented by devices capable of bus mastering. This register is read only and is hardwired to 0x0.

Bit	Description
7:0	Latency Timer. Default is 0x0.

15.9 Header_type Register

The **Header_type** register defines the format of the PCI base address registers (**memBaseAddr** in Avenger). Bits 0:6 are read only and hardwired to 0x0. Bit 7 of **Header_type** specifies Avenger as a single function PCI device.

Bit	Description
6:0	Header Type. Default is 0x0.
7	Multiple-Function PCI device (0=single function, 1=multiple function). Default is 0x0.

15.10 BIST Register

The **BIST** register is implemented by those PCI devices that are capable of built-in self-test. Avenger does not provide this capability. This register is read only and is hardwired to 0x0.

Bit	Description
7:0	BIST field and configuration. Default is 0x0.

15.11 memBaseAddr0 Register

The **memBaseAddr** register determines the base address for all PCI memory mapped accesses to Avenger. Writing 0xffffffff to this register will reset it to its default state. Once **memBaseAddr** has been reset, it can be probed by software to determine the amount of memory space required for Avenger. A subsequent write to **memBaseAddr** will set the memory base address for all PCI memory accesses. See the PCI specification for more details on memory base address programming. Avenger requires 32 Mbytes of address space for memory mapped accesses. For memory mapped accesses on the 32-bit PCI bus, the contents of **memBaseAddr** are compared with the **pci_ad** bits 31..25 (upper 7 bits) to determine if Avenger is being accessed. This register is R/W.

Bit	Description
31:0	Memory Base Address. Default is 0xf8000000.

15.12 memBaseAddr1 Register

The **memBaseAddr** register determines the base address for all PCI memory mapped accesses to Avenger. Writing 0xffffffff to this register will reset it to its default state. Once **memBaseAddr** has been reset, it can be probed by software to determine the amount of memory space required for Avenger. A subsequent write to **memBaseAddr** will set the memory base address for all PCI memory accesses. See the PCI specification for more details on memory base address programming. Avenger requires 32 Mbytes of address space for memory mapped accesses. For memory mapped accesses on the 32-bit PCI bus, the contents of **memBaseAddr** are compared with the **pci_ad** bits 31..25 (upper 7 bits) to determine if Avenger is being accessed. This register is R/W.

Bit	Description
31:0	Memory Base Address. Default is 0xf8000008.

15.13 ioBaseAddr Register

The **memBaseAddr** register determines the base address for all PCI IO mapped accesses to Avenger. Writing 0xffffffff to this register will reset it to its default state. Once **ioBaseAddr** has been reset, it can be probed by software to determine the amount of io space required for Avenger. A subsequent write to **ioBaseAddr** will set the IO base address for all PCI memory accesses. See the PCI specification for more details on IO base address programming. Avenger requires 256 Bytes of address space for IO mapped accesses. For IO mapped accesses on the 32-bit PCI bus, the contents of **ioBaseAddr** are compared with the **pci_ad** bits 31..8 (upper 24 bits) to determine if Avenger is being accessed. This register is R/W.

Bit	Description
31:0	IO Base Address. Default is 0xffffff01.

15.14 subVendorID Register

The **subVendorID** register defines the board manufacturer ID. During system initialization the expansion code located at **romBaseAddr** will set this register to the appropriate value. This register is read during plug and play initialization. See the PC97 specification for more details on **subVendorID** and plug and play requirements. The default value for this register is automaticaly loaded after reset from the ROM. Bits 7:0 are stored in ROM location 0x7ff8, while bits 15:8 are stored in 0x7ff9 for a 32K ROM. Bits 7:0 are stored in ROM location 0xfff8, while bits 15:8 are stored in 0xfff9 for a 64K ROM.

Bit	Description
15:0	Subsystem Vendor ID register, Initialized by expansion prom, default is read by ROM.

15.15 subSystemID Register

The **subSystemID** register defines the board type. During system initialization, the expansion code located at **romBaseAddr** will set this register to the appropriate value. This register is read during plug and play initialization. See the PC97 specification for more details on **subSystemID** and plug and play requirements. The default value for this register is automaticaly loaded after reset from the ROM. Bits 7:0 are stored in ROM location 0x7ffa, while bits 15:8 are stored in 0x7ffb for a 32K ROM. Bits 7:0 are stored in ROM location 0xfffa, while bits 15:8 are stored in 0xfffb for a 64K ROM.

Bit	Description
15:0	Subsystem ID register, Initialized by expansion prom, default is read by ROM.

15.16 romBaseAddr Register

The **romBaseAddr** register determines the base address for all PCI ROM accesses to Avenger. Writing 0xfffffffe to this register will reset it to its default state. Once **romBaseAddr** has been reset, it can be probed by software to determine the amount of ROM space required for Avenger. A subsequent write to **romBaseAddr** will set the ROM base address for all PCI memory accesses. See the PCI specification for more details on memory base address programming. Avenger requires 32 to 64 Kbytes of address space for ROM accesses and is configured by strapping bit 2. For ROM accesses on the 32-bit PCI bus, the contents of **romBaseAddr** are compared with the **pci_ad** bits 31..16 (upper 16 bits) to determine if Avenger is being accessed. This register is R/W.

Bit	Description
31:0	Expansion Rom Base Address. Default is 0xffff8000 or 0xffff0000.

15.17 Capabilities Pointer

The Capabilities pointer register contains the offset in configuration space of beginning of the capability link list structure. This register is read only.

Bit	Description
31:0	Capabilities Pointer offset. Default is 0x00000054 if AGP is enabled via the strapping bits, otherwise it is 0x60.

15.18 Interrupt_line Register

The **Interrupt_line** register is used to map PCI interrupts to system interrupts. In a PC environment, for example, the values of 0 to 15 in this register correspond to IRQ0-IRQ15 on the system board. The value 0xff indicates no connection. This register is R/W.

Bit	Description
0:7	Interrupt Line. Default is 0x5 (IRQ5)

15.19 Interrupt_pin Register

The **Interrupt_pin** register defines which of the four PCI interrupt request lines, INTA* - INTRD*, the PCI device is connected to. This register is read only and is hardwired to 0x1.

Bit	Description
0:7	Interrupt Pin. Default is 0x1 (INTA*)

15.20 Min_gnt Register

The **Min_gnt** register specifies the burst period a PCI bus master requires. It must be implemented by devices capable of bus mastering. This register is read only and is hardwired to 0x0 since Avenger does not support bus mastering.

Bit	Description
7:0	Minimum Grant. Default is 0x0.

15.21 Max_lat Register

The **Max_lat** register specifies the maximum request frequency a PCI bus master requires. It must be implemented by devices capable of bus mastering. This register is read only and is hardwired to 0x0 since Avenger does not support bus mastering.

Bit	Description
7:0	Maximum Latency. Default is 0x0.

15.22 fabID Register

Identification code of the manufacturing plant.

Bit	Description
3:0	Manufacturing fab identification. Read only. (1 = TSMC)
31:4	Scratch pad. (read / write)

15.23 cfgStatus Register

The **cfgStatus** register is an alias to the normal memory-mapped status register. See section x.x for a description of the status registers. Reading the configuration space **cfgStatus** register returns the same data as if reading from the memory-mapped status register.

15.24 cfgScratch Register

The **cfgScratch** register can be used as scratch pad storage space by software. The values of **cfgScratch** are not used internally to alter functionality, so any value can be stored to and read from **cfgScratch**.

Bit	Description
31:0	Scratchpad register. Default is 0x0.

15.25 New capabilities (AGP and ACPI)

AGP and ACPI Use PCI ‘s new capabilities mechanism. The New Capabilities structure is implemented as a linked list of registers containing information for each function supported by Avenger. The list contains both AGP status and command registers. AGP registers read back ‘0’ if AGP is disabled via the strapping pins.

15.26 Capability Identifier Register

The capability register resides at offset (CAP_OFFSET). This register identifies AGP revision compliance

Bit	Description
7:0	Capability ID. Always == 2 for AGP
15:8	Next Capability ID Pointer. Default is 0x60.
19:16	Minor AGP revision, the interface conforms to
23:20	Major AGP revision, the interface conforms to
31:24	Reserved. Defined as 0.

15.27 AGP Status

AGP status register documents maximum number of requests that Avenger can manage, AGP sideband capable, and transfer rate

Bit	Description
2:0	Data rates that Avenger can deliver/receive. Bit[0] = 1x, bit[1] = 2x, bit[2] = 4x. Default is 3.
4:3	Reserved. Default is 0
5	AGP_4G. AGP supports above 4 Giga bytes of memory. Default is 1.
8:6	Reserved. Default is 0.
9	SBA. Device supports side band addressing
23:10	Reserved. Default is 0
31:24	RQ_DEPTH. Max # of requests that Avenger can manage. Default is 7.

15.28 AGP Command

AGP status register documents maximum number of requests that Avenger can manage, AGP sideband capable, and transfer rate

Bit	Description
2:0	Data Rate bit[0] = 1x, bit[1] = 2x. Only 1 bit must be set (Read/Write)
4:3	Reserved. Default is 0
5	AGP_4G_ENABLE. AGP supports above 4 Giga bytes of memory. Default is 0.
7:6	Reserved. Default is 0
8	AGP enable. Enables AGP function. AGP_RESET sets this bit to 0. (R/W)
9	SBA_ENABLE. Enable side band addressing mechanism. (R/W)
23:10	Reserved. Default is 0
31:24	RQ_DEPTH. Max # of requests System can handle. (R/W)

15.29 ACPI Cap ID

The ACPI Cap ID register identifies what Avenger supports in ACPI.

Bit	Description
7:0	Capability ID. Always == 1 for ACPI
15:8	Next Capability ID Pointer. Default is 0
18:16	Version. Default is 0x1.
19	PME Clock. Default is 0.
20	Aux Power Source. Default is 0.
21	DSI. Default is 1. Indicates additional software initialization must take place.
24:22	Reserved. Default is 0
25	D1 Support. Default is 0.
26	D2 Support. Default is 0.
31:27	PME Support. Default is 0.

15.30 ACPI Ctrl/Status

ACPI status register allows transition from the D3 to D0 state.

Bit	Description
1:0	Power State. Defaults to 0x0. Avenger only accepts writes of 0x0 or 0x3 to these bits. (R/W)
7:2	Reserved. Default is 0
8	Sticky bit. Default is 0.
12:9	Data Select. Default is 0.
14:13	Data Scale. Default is 0.
15	Sticky bit. Default is 0.
21:16	Reserved. Default is 0
22	B2 B3 support. Default is 0.
23	BPCC_En. Default is 0.
31:24	Data for data select. Default is 0.

16. Init Registers

Register Name	I/O Address	Bits	R/W	Description
status	00-03	31:0	R	Avenger status register
pciInit0	04-07	31:0	R/W	PCI initialization register
sipMonitor	08-0b	31:0	R/W	Silicon Process Monitor register.
lfbMemoryConfig	0c-0f	31:0	R/W	Starting Tile page and stride register
miscInit0	10-13	31:0	R/W	Misc. initialization register
miscInit1	14-17	31:0	R/W	Misc. initialization register
dramInit0	18-1b	31:0	R/W	Dram initialization register0
dramInit1	1c-1f	31:0	R/W	Dram initialization register1
agpInit0	20-23	31:0	R/W	AGP initialization register
textureGRXBackend	24-27	31:0	R/W	Read-reordering memory initialization register
vgaInit0	28-2b	31:0	R/W	VGA initialization register
vgaInit1	2c-2f	31:0	R/W	VGA initialization register
2d_command	30-33	31:0	W	2d command register (to be used to write SGRAM mode and special mode registers)
2d_srcBaseAddr	34-37	31:0	W	2d srcBaseAddr register (to be used to write to SGRAM mode and special mode registers)
strapInfo	38-3b	11:0	R	Strap bits after power up.

16.1 status Register (0x0)

The **status** register provides a way for the CPU to interrogate the graphics processor about its current state and FIFO availability. The **status** register is read only, but writing to **status** clears any Avenger generated PCI interrupts.

Bit	Description
4:0	PCI FIFO freespace (0x1f=FIFO empty). Default is 0x1f.
5	PCI FIFO busy. Default is 0x0.
6	Vertical retrace (0=Vertical retrace active, 1=Vertical retrace inactive). Default is 1.
7	FBI graphics engine busy (0=engine idle, 1=engine busy). Default is 0.
8	TREX busy (0=engine idle, 1=engine busy). Default is 0.
9	Avenger busy (0=idle, 1=busy). Default is 0.
10	2D busy (0=idle, 1=busy). Default is 0.
11	Cmd fifo 0 busy. Default is 0x0.
12	Cmd fifo 1 busy. Default is 0x0.
27:13	reserved
30:28	Swap Buffers Pending. Default is 0x0.
31	PCI Interrupt Generated. Default is 0x0.

Bits(4:0) show the number of entries available in the internal host FIFO. The internal host FIFO is 32 entries deep. The FIFO is empty when bits(4:0)=0x1f. Bit(6) is the state of the monitor vertical retrace signal, and is used to determine when the monitor is being refreshed. Bit(7) of **status** is used to determine if the graphics engine of FBI is active. Note that bit(7) only determines if the graphics engine of FBI is busy – it does not include information as to the status of the internal PCI FIFOs. Bit(8) of **status** is used to determine if TREX is busy. Note that bit(8) of **status** is set if any unit in TREX is not idle – this includes the graphics engine and all internal TREX FIFOs. Bit(9) of **status** determines if all units in the Avenger system (including graphics engines, FIFOs, etc.) are idle. Bit(9) is set when any internal unit in

Avenger is active (e.g. graphics is being rendered or any FIFO is not empty). Bit(10) of **status** is used to determine if the 2D graphics engine is active. Bits(11:10) of **status** is used to determine if either command fifo 0 or command fifo 1 are active. When a SWAPBUFFER command is received from the host cpu, bits (30:28) are incremented – when a SWAPBUFFER command completes, bits (30:28) are decremented. Bit(31) of status is used to monitor the status of the PCI interrupt signal. If Avenger generates a vertical retrace interrupt (as defined in **pciInterrupt**), bit(31) is set and the PCI interrupt signal line is activated to generate a hardware interrupt.

16.2 pciInit0 Register (0x4)

pciInit0 register contains the control information on how PCI should behave. Bits 15:0 are the output of the counter clocked by GRX clock. Bits 19:18 control Interrupts. Bits 17:13 allow the retry interval to be increased, while bits 12 and 11 allow retries to be disabled. Bits 9 and 8 determine the bus performance. Bits 6:2 determine the PCI fifo Low water mark. This value should never be 0 (no overflow checking is done) and should be set greater than 2 for any fast device operations. Bits 25:20 control how many non modal LFB accesses are grouped together before being pushed to memory. This register is read write and defaults to 0x01800040.

Bits 28:27 control the adjustable timeout for PCI writes travelling into the frame buffer. After the indicated number of clocks, writes in the PCI fifo will get flushed out to the frame buffer. By increasing this timeout, LFB/Command traffic can be made more efficient because the timeout will force the hardware to wait for more entries in the fifo to accumulate before writing anything to the frame buffer. By emptying the fifo in larger and less-frequent groups, overall Avenger memory efficiency can increase. The downside of increasing this threshold is potentially more latency between the time when a PCI write is received, and the time that it actually enters the frame buffer. In general, this increase in latency does not cause hardware slowdown.

Bit	Description
1:0	Reserved
6:2	PCI FIFO Empty Entries Low Water Mark. Valid values are 0-31. Default is 0x10.
7	Reserved. Default is 0x1.
8	Wait state cycles for PCI read accesses (0=1 ws, 1=2 ws). Default is 0x0.
9	Wait state cycles for PCI write accesses (0=no ws, 1=one ws). Default is 0x0.
10	Reserved. Default is 0x0.
11	Disable PCI IO access retries. . Default is 0x0.
12	Disable PCI Memory access retries. Default is 0x0.
17:13	Retry interval, less 8 clocks. Default is 0.
18	PCI Interrupt Enable Default is 0.
19	PCI Interrupt Time Out Enable. Default is 0.
25:20	PCI Fifo read threshold. Default is 0x18.
26	Force PCI/CMD Frame buffer accesses to high priority. (1= high, 0 = low, except for PCI frame buffer reads). . Default is 0x0.
28:27	PCI fifo to LFB write timeout in MCLKs (0=64, 1=128, 2= 196, 3=256).

16.3 sipMonitor Register (0x8)

sipMonitor register contains the silicon performance counters used to measure silicon performance by clocking a counter by a ring oscillator of NOR's and NAND's and comparing the value to a counter based

on GRX clock. The larger the process counter, the faster the process. Bits 15:0 are the output of the counter clocked by GRX clock. Bits 27:16 is the counter clocked by the ring oscillator. Bit 28 clears the ring oscillator counter to zero. Bit 29 selects either a nand chain or a nor chain so one can measure the P transistor strength or the N transistor strength. Bit 30 enables the monitor. This register is read write and defaults to 0x40000000.

Bit	Description
15:0	Silicon Performance process counter in GRX domain
27:16	Oscillation counter
28	Oscillation counter reset_n. Default is 0x0.
29	Oscillation Nor select (0=nand, 1=nor) . Default is 0x0.
30	Frequency counter enable (1=enabled, 0=disabled) . Default is 0x0.

16.4 lfbMemoryConfig Register (0xC)

memoryConfig defines the tile beginning page in sgram space. Bits 12:0 define the first page of tile addressing. Bits 15:13 define tile stride in bytes for PCI 2 XY translation. Bits 21:16 define the sgram tile stride in X. This register is read write and defaults to 0xa2200.

Bit	Description
12:0	Tile aperture begin page. (0-8k pages). Default is 0x200
15:13	Tile aperture stride in bytes (0=1k, 1=2k, 2=4k, 3=8k, 4 = 16k). Default is 0x1.
22:16	Number of sgram tiles in X. Default is 0xa.

16.5 miscInit0 Register (0x10)

miscInit0 contains resets to all subsystems, pixel swizzling, and Y origin subtraction. Bits [1:0] reset the 3D graphics subsytem. Bits [3:2] enable byte/word swizzling during register accesses to 2D or 3D. Bits [6:4] define resets for video, 2D, and memory subsytems. Bits[29:18] define the Y origin subtraction value used during address calculation when Y flip is enabled in fbzMode. Bits [31:30] enable byte/word swizzling during non modal LFB reads and writes

Bit	Description
	Miscellaneous Control
0	FBI Graphics Reset (0=run, 1=reset). Default is 0.
1	FBI FIFO Reset (0=run, 1=reset). Default is 0. [resets PCI FIFO and the PCI data packer]
2	Byte swizzle incoming register writes (1=enable). [Register byte data is swizzled if **miscInit0**[2]==1 and pci_address[20]==1] for 3D registers, and pci_address[19] ==1 for 2D registers. Default is 0.
3	Word swizzle incoming register writes (1=enable). [Register word data is swizzled if **miscInit0**[2] == 1 and pci_address[20] ==1] for 3D registers, and pci_address[19] ==1 for 2D registers. Default is 0.
4	Video Timing Reset (0=run, 1=reset). Default is 0.
5	2D Graphics Reset (0=run, 1=reset). Default is 0.
6	Memory Timing Reset (0=run, 1=reset). Default is 0.
7	VGA Video Timing Reset (0=run, 1=reset). Default is 0.
10:8	Programmable delay to be added to the blank signal before it outputs to the TV out interface. This is in terms of number of flops clocked by the 2x clock. The objective is to synchronize the blank signal with the data output by matching the CLUT delay. Default is 0x0. 000 = 2 flops; 001 = 3 flops; …… 111 = 9 flops
13:11	Programmable delay to be added to the vsync and hsync signals before they are output to the TV out interface. This is in terms of number of flops clocked by the 2x clock. The objective is to synchronize the sync signals with the data output by matching the CLUT delay. Default is 0x0. 000 = 2 flops; 001 = 3 flops; …… 111 = 9 flops
16:14	Programmable delay to be added to the vsync and hsync signals before they are output to the monitor. This is in terms of number of flops clocked by the 2x clock. The objective is to synchronize the sync signals with the data output by matching the delay through the CLUT and DAC. Default is 0x0. 000 = 2 flops; 001 = 3 flops; …… 111 = 9 flops
17	Reserved
	Y Origin Definition bits
29:18	Y Origin Swap subtraction value (12 bits). Default is 0x0.
30	Byte swizzle incoming non modal LFB writes (1=enable). Default is 0.
31	Word swizzle incoming non modal LFB writes (1=enable). Default is 0.

16.6 miscInit1 Register (0x14)

miscInit1 register controls miscellaneous operations of Avenger available in real mode. Bit 0 is used to correct for CLUT addresses being inverted during host accesses. This bit should be set to 1 for proper operation. Bit 3 enables and disables writes to the PCI **subVendorID** and **subSystemID** registers. Bit 4 enables writes to the ROM through **romBaseAddr**. Bit 5 enables the new triangle address aliasing allowing better address compaction. Bit 6 disables texture mapping.

Power down of H3 is controlled by bits 11:7, where bit 7 powers down the color lookup tables, bit 8 powers down the DAC itself, bits 9, 10, and 11 power down the three PLL's.

Bits 17 and 18 disable stalling on the opposite pipe (either 2D or 3D) when a command is sent down. These bits are used for testing, and should not be set during normal operation.

Bit 19 is used to terminate command fifo activity. Setting this bit to '1' halts the command fifo and resets all of the registers in the command register space to their default values. In order for Avenger to be shut down gracefully, this bit should only be set when Avenger is idle. Be sure to restore this bit to 0 when finished.

Bits 28 through 24 indicate the value of the strapping registers at boot up. Note that altering these bit effect the read back information of PCI and AGP resource reporting. For more information on the strapping registers, see the section on Power on Strapping

Bit	Description
	Miscellaneous Control
0	invert_clut_address. Default = 0.
2:1	tri_mode – triangle iterator mode. Default is 0x0.
3	Enable Sub Vendor/ Subsystem ID writes. (0=disable, 1=enable). Default is 0x0.
4	Enable ROM writes. (0=disable, 1=enable). Default is 0x0.
5	Alternate triangle addressing map (0=disable, 1=enable). Default is 0x0.
6	Disable texture mapping (0=enable, 1=disable). Default is 0x0.
	Power Down Control
7	Power Down CLUT. Default is 0x0.
8	Power Down DAC. Default is 0x0.
9	Power Down Video PLL. Default is 0x0.
10	Power Down Graphics PLL. Default is 0x0.
11	Power Down Memory PLL. Default is 0x0.
	2D Block Write Control
14:12	Block write threshold. Default is 0x0.
15	Disable 2D Block write. Default is 0x0.
16	Disable 2D stall on 3D synchronous dispatch. When set to 1, 2D will not wait on pending 3D operations to complete before being issued. Default is 0x0.
17	Disable 3D stall on 2D synchronous dispatch. When set to 1, 3D will not wait on pending 2D operations to complete before being issued. Default is 0x0.
18	Reserved.
19	Command Stream Reset (1=reset command streams, 0 = normal operation). Default is 0x0.
23:20	Reserved
24	PCI Fast device. Default strapped on VMI_DATA 0
25	PCI BIOS Size. Default strapped on VMI_DATA 1
26	PCI 66Mhz. Default strapped on VMI_DATA 2.
27	AGP Enabled. Default strapped on VMI_DATA 3.
28	PCI Device Type. Default strapped on VMI_DATA 4.

16.7 dramInit0 Register (0x18)

dramInit0 controls the sgram interface timing of specific timing parameters. The default value of this register is 0x00579d29.

Bit	Description
	Sgram access timing
1:0	tRRD – row active to row active (1-4 clks). Default is 0x1 (2 clks)
3:2	tRCD – RAS to CAS delay (1-4 clks). Default is 0x2 (3 clks).
5:4	tRP – row precharge (1-4 clks). Default is 0x2 (3 clks).
9:6	tRAS – minimum active time (1-16 clks). Default is 0x4 (5 clks).
13:10	tRC – minimum row cycle time (1-16 clks). Default is 0x7 (8 clks).
15:14	tCAS latency (1-4 clks). Default is 0x2 (3 clks).
16	tMRS mode and special mode register cycle time (1-2 clks).. Default is 0x1 (2 clks)
17	tDQR Rd to DQM assertion delay (0-1 clks). Default is 0x1 (1 clk)
18	tBWC Block write cycle time (1-2 clks). Default is 0x1 (2 clks)
19	tWL WR to pre (1-2 clks). Default is 0x0 (1 clk)
21:20	tBWL BKWR to Pre (1-4 clks). Default is 0x1 (2 clks)
22	tRL RD to PRE (1-2 clks). Default is 0x1 (2 clks)
23	dont allow WR/BKWR to terminate RD, use BST. Default is 0x0 (allow wr-term)
24	Disable the dead bus cycle btw. RD and WR (0=enable, 1=disable). Default is 0x0
25	SGRAM write per bit enable (0=disable, 1=enable). Default is 0x0
26	Number of Sgram chipsets (0=1, 1=2). Default is 0x0. (power on strap = VMI_DATA_5)
27	Sgram type (0=8Mbit, 1=16Mbit). Default is 0x0. (power on strap = VMI_DATA_6)

16.8 dramInit1 Register (0x1C)

Bit	Description
	SGRAM Refresh Control
0	Refresh Enable (0=disable, 1=enable). Default is 0.
9:1	Refresh_load Value. (Internal 14-bit counter 5 LSBs are 0x0) Default is 0x100.
	Video Refresh Control
10	Video arbitration priority. (0=normal, 1=aggressive) Default is 0
	Miscellaneous video Control
11	Triple buffer enable (0=double buffering, 1=triple buffering). Default = 0.
12	Dither passthrough (0=dithering, 1=bypass dithering). Default = 0.
	SGRAM read data sampling control
13	sg_clk_nodelay – bypass the delay element. Default = 1.
14	sg_use_inv_sample – resample the flopped sgram data with another negative-edge flop before flopping data with mclk. Default = 0.
15	sg_del_clk_invert – invert delayed clock before using it. Default = 0.
19:16	sg_clk_adj – delay value for sgram read data sample clock. Default = 0x0. (2-62 NAND gates of delay, in steps of 4)
	SGRAM frame buffer output delay control (control + data bits)
23:20	sg_oflop_del_adj – delay value for mclk (2-62 nand gates of delay, in steps of 4) to transparent latch that sends fb_* off chip. Default = 0xf.
24	sg_oflop_trans_latch – forces latch for fb_* signals to be transparent. (0=use delayed mclk to latch, 1=make latch always transparent). Default = 0.
	Memory Controller configuration bits
25	mctl_short_power_on. Power on in 128 cycles. Default = 0. VMI_ADDR_1
26	mctl_no_aggressive – turn off mem_ctrl's aggressive row activation. Default = 0.
27	mctl_pagebreak – force a pagebreak for all accesses. Default = 0.
28	mctl_tristate_outputs – force data outputs to be tristate. Default = 0.
29	mctl_no_vin_locking – prevent vin from locking the bus during requesting. 0=allow locking, 1=prevent locking. Default = 0.
30	Rev. B0 and after: mctl_type_sdram – (0=use SGRAMs, 1=use SDRAMs). Default = 0. VMI_ADDR_2

When using SGRAMs, *mctl_type_sdram* should be set to 0. When using SDRAMs, only 16Mbit (16x512K parts) are supported, which result in a 16MB frame buffer. The *sgram_type* and *sgram_chipsets* bits in **dramInit0** are ignored when *mctl_type_sdram*=1.

Rev. B0 and after: Note that the **fastfillCMD** behaves differently when *mctl_type_sdram*=1 (**dramInit1[30]**). When fastfilling with SGRAMs (*mctl_type_sdram*=0), if dithering is enabled and **fastfillCMD[0]**=1, no dithering will happen. But when fastfilling with SDRAMs (*mctl_type_sdram*=1), if dithering is enabled and **fastfillCMD[0]**=1, dithering will still happen, since SDRAMs do not support blockwriting.

16.9 agpInit0 Register (0x20)

The **agpInit0** register is used to control how AGP behaves when making requests. Bit 0 sets the request priority level. Bits [3:1] control the largest size the request can be. Bits [6:4] determine when the agp

request fifo becomes full (requests that have not yet been issued to the AGP target). Bits [10:7] control when to much data has been returned, and AGP needs to begin stalling. Bits 31:27 allow for active outstanding AGP requests to be monitored. the Value 0x1f indicates no outstanding requests, while 0x0 indicates 17 outstanding requests.

Bit	Description
0	Force AGP request to be high priority. (0=Low, 1 = High). Default is 0x0.
3:1	Maximum AGP request length. (0=1 octword, 7 = 8 octwords). Default is 0x7.
6:4	AGP request fifo full threshold. Default is 0x1.
10:7	AGP read fifo full threshold. Default is 0x9.
26:10	Reserved. Defaults is 0x0.
31:27	AGP Requests Outstanding. (Read Only)

16.10 tmuGbeInit Register (0x24)

tmuGbeInit register contains the fifo water marks for both the TMU and FBI sections. The default value of this register is 0x0bfb. Bits [19:15] control the delay (with a NAND chain) of the TV-Out output clock.

Bit	Description
3:0	Texture read request low water mark - fifo freespace level that TMU will empty the read request queue to before stopping a sequence. Default is 0xb.
7:4	Texture read request high water mark - fifo freespace level at which TMU will start a sequence. Default is 0xf.
11:8	Pixel fifo high water mark - fifo freespace level at which FBI will start a sequence. Default is 0xf.
12	txc_disable_rdrq_max - disable the limit of 16 as the max limit of max number of reads in a row by the read-reordering memory interface to the memory controller. Default = 0.
13	txc_use_min_req - sets the minimum number of reads done by the read-reordering memory to 3. Default = 0.
14	txc_force_cam_miss. Default = 0.
	TV Out clock delay adjust
15	tv_out_clk_inv - invert delayed clock. Default = 0.
19:16	tv_out_clk_del_adj - choose between 16 values of delay. Default = 0.

16.11 vgaInit0 Register (0x28)

The **vgaInit0** register is used for hardware initialization and configuration of the VGA controller in Avenger. VGA can be disabled by writing bit 0 to a "1". Bit 1 allows external video timing to drive the VGA core video scan out logic. Bit 2 controls how the VGA DAC control logic views the width of the RAM. For VGA compatibility, this bit should be set to 0 (6 bit DAC).

VGA extensions are enabled by bit 6. These extensions are mention in the VGA portion of this spec, in the CRTC register space. Bit 10 enables the ability to read back the PCI configuration when bit 6 of this register is 0.

Bit 8 determines if the chips should wake up as a VGA motherboard or an add in card. Bit 9 disables the VGA to response to legacy address decoding. This bit should be set if Avenger is not the primary display adapter in the system. By default, this bit is set if Avenger is set as a multimedia device with the strapping bits. Setting this bit also disables write access to 0x46e8 and 0x102.

Bit 12 should be set when in an extended (non-VGA) mode. This disables the VGA from fetching memory data during video raster scan out.

Bit 13 is used when an external DAC is supported. This bit should always be set to 0.

Bit 21:14 determine the start page of VGA in board memory. By default, VGA is placed at the beginning of memory. If need be, it can be moved anywhere on a 64K byte boundary within 16M bytes.

Bit 22 disables VGA refresh control of board memory. When VGA is in scan out mode, it prefers memory refresh to happen at horizontal sync time. When this bit is set to 0, three memory refresh cycles happen after HBLANK occurs, and the memory refresh time out counter is deferred. When this bit is set to 1, the memory refresh time out counter explicitly controls memory refresh events.

Bit	Description	Default
	Miscellaneous Control	
0	VGA disable. (0=Enable, 1 = Disable). Setting this bit to 1 shuts off all access to the VGA core.	0
1	Use external video timing. This bit is used to retrieve SYNC information through the normal VGA mechanism when the VGA CRTC is not providing timing control.	0
2	VGA 6/8 bit CLUT. (0= 6 bit, 1 = 8 bit).	0
5:3	Reserved.	1
6	Enable VGA Extensions	0
7	Reserved	0
8	0x46e8/0x3C3 Wake up select (0=use 0x46e8, 1=use 0x3C3 or IO Base + 0xC3). VGA add in cards that use 0x46e8 while mother board VGA uses 0x3C3. When Avenger is a multimedia device, this bit should be set to '1' and the VGA subsystem should be enabled with IO Base + 0xC3.	0
9	Disable VGA Legacy Memory/IO Decode (0=Enable, 1=Disable)	0
10	Use alternate VGA Config read back (0 = Enable, 1=Disable). Setting this bit to 0 allows the VGA to read back configuration through CRTC index 0x1c.	0
11	Enable Fast Blink (test bit) (1=fast blink, 0 = normal blink).	0x0
12	Use extended video shift out. Set this bit to 1 to disable all VGA memory access when video processor is shifting out data.	0
13	Decode 3c6. (test bit)	0
21:14	Vga base offset in 64k quantities	
22	Disable SGRAM refresh requests on HBLANK. When set to 1, the VGA does not produce memory refreshes during horizontal blanking.	0
31:23	reserved	0

16.12 vgaInit1 Register (0x2C)

The **vgaInit1** register contains the read and write apertures for VBE. VBE uses address 0xA0000 as an aperture into Avenger memory. See the section on VBE apertures in the VGA portion of this document. Bit 20 enables sequential chain mode, a pseudo packed pixel format Bits 28:21 define lock bits that disable writes to specific sections of the VGA core. See the section on register locking in the VGA portion of this document.

Bit	Description	Default
9:0	VBE write Aperture, in 32K granularity	0
19:10	VBE read Aperture, in 32K granularity	0
20	Enable 0xA0000 Sequential Chain 4 mode.	0
21	Lock Horizontal Timing - 3B4/3D4 index 0,1,2,3,4,5,1a	0
22	Lock Vertical Timing -3B4/3D4 index 6,7 (bit 7,5,3,2 and 0), 9 10, 11 (bits[3:0]), 15,16,1b.	0
23	Lock H2 - 0x3B4/0x3D4 index 17, bit 2	0
24	Lock Vsync - 0x3C2, bit 7.	0
25	Lock Hsync - 0x3C2, bit 6.	0
26	Lock Clock Select - 0x3C2, bits 3 and 2.	0
27	Lock Ram Enable - 0x3C2, bit 1	0
28	Lock Character Clock - 0x3C4, index 1 bit 0.	0

16.13 2d_Command_Register (0x30)

Writing to this register is the same as writing to the 2d unit's **command** register. This mapping is intended to provide a way to initialize the SGRAM mode and special mode registers at init time.

16.14 2d_srcBaseAddr Register (0x34)

Writing to this register is the same as writing to the 2d unit's **srcBaseAddr** register. This mapping is intended to provide a way to initialize the SGRAM mode and special mode registers at init time.

17. Frame Buffer Access

17.1 Frame Buffer Organization

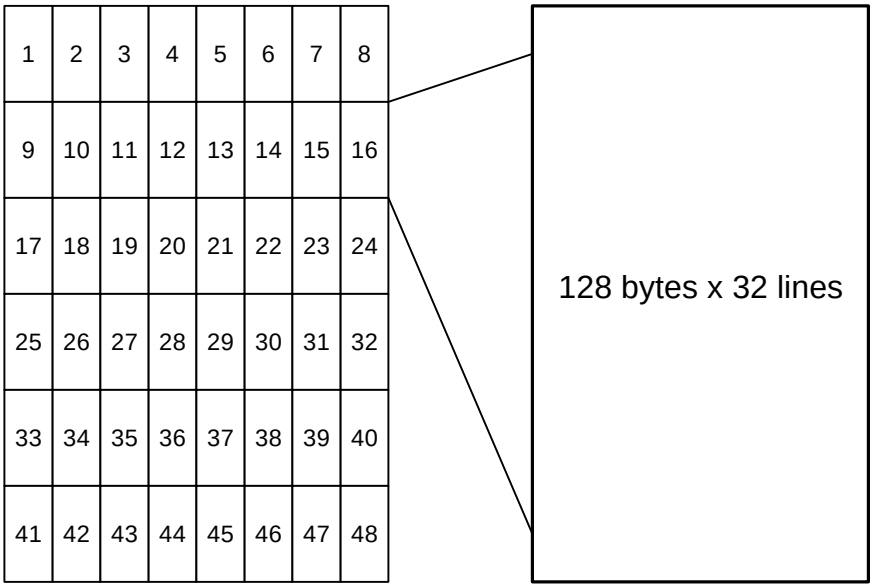
The Avenger linear frame buffer base address is located in a separate memory base address register in PCI config space and occupies 32 megabytes of address space for linear access. Linear memory starts at the beginning of Sgram memory and finishes at the begin of tiled memory specified by the tilebase register in init register space. It is assumed (but not required) that VGA will use the first 256K of linear memory, and the desktop, video, and textures will use the remaining linear memory.

17.2 Linear Frame Buffer Access

Linear frame buffer access is accessed much like system, and can store the desktop, video, 3D front buffer, 3D back buffer, 3D auxillary buffer, and textures. Memory management is done with a true linear memory manager.

17.3 Tiled Frame Buffer Access

Tiled frame buffer access is a rectilinear memory based on 128 byte x 32 line tiles. Tiled memory is suited for 3D performance, where localized access is needed. Tiled frame buffer access is done with a concatenation of Y and X much like the 3D linear frame buffer access, and the frame buffer access of SST1. Tiled frame buffer Access is defined by writing the beginning tile/page into the tiled base address register. When configuring tiled frame buffer access, it is best to set the global tile stride to the largest surface width for best memory management. Memory management for tiled memory must be done by a rectilinear memory manager. Access to tiled memory is shown below. It is recommended that tiled memory be used sparingly since this memory fragments very easily, and utilization will not be 100%.



This is the PCI to XY calculation

PCI Offset to XYbyte conversion

1K byte stride	
Y[14:0], X[9:0]	[24:0]
2K byte stride	
Y[13:0], X[10:0]	[24:0]
4K byte stride	
Y[12:0], X[11:0]	[24:0]
8K byte stride	
Y[11:0], X[12:0]	[24:0]
16K byte stride	
Y[10:0], X[13:0]	[24:0]

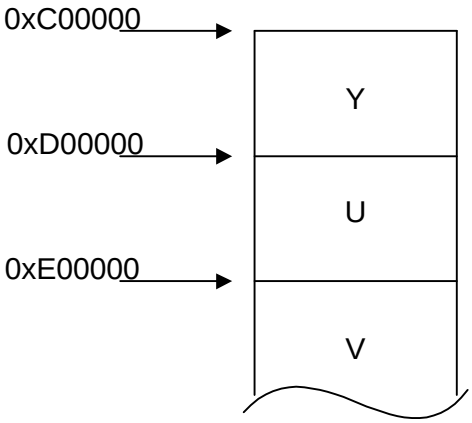
The following is the page calculation that must be used in the base address registers in video, 3D, 2D, and texture units.

Page equation = (X / 128) + basepage + (Y / 32) * tilestride.

: Where X is referenced in bytes, and Y is referenced in lines.

18. YUV Planar Access

YUV planar memory allows the CPU to write Y, U, and V in separate regions of memory space. As Y, U, and V are written, they are converted into YUYV packed form, and stored in the frame buffer at the correct offset from the YUV base address register. The first megabyte region defines Y, where each 32-bit write, generates a 64-bit write on Avenger, with appropriate byte masks. The second megabyte region of YUV planar memory defines U space, where each 32-bit write generates two 64-bit writes with appropriate byte enable bits. The third region of YUV planar memory defines the V space, where each 32-bit write generates two 64-bit writes with appropriate byte enable bits. The conversion between planar and packed is described below. YUV planar space has a fixed 1024 byte stride, and a programmable destination stride.



Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
Y8	Y9	Y10	Y11	Y12	Y13	Y14	Y15
Y16	Y17	Y18	Y19	Y20	Y21	Y22	Y23
Y24	Y25	Y26	Y27	Y28	Y29	Y30	Y31
Y32	Y33	Y34	Y35	Y36	Y37	Y38	Y39
Y40	Y41	Y42	Y43	Y44	Y45	Y46	Y47
Y48	Y49	Y50	Y51	Y52	Y53	Y54	Y55
Y56	Y57	Y58	Y59	Y60	Y61	Y62	Y63

U0	U1	U2	U3
U4	U5	U6	U7
U8	U9	U10	U11
U12	U13	U14	U15

V0	V1	V2	V3
V4	V5	V6	V7
V8	V9	V10	V11
V12	V13	V14	V15

YUV Planar space

Byte #

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Y0	U0	Y1	V0	Y2	U1	Y3	V1	Y4	U2	Y5	V2	Y6	U3	Y7	V3
Y8	U0	Y9	V0	Y10	U1	Y11	V1	Y12	U2	Y13	V2	Y14	U3	Y15	V3
Y16	U4	Y17	V4	Y18	U5	Y19	V5	Y20	U6	Y21	V6	Y22	U7	Y23	V7
Y24	U4	Y25	V4	Y26	U5	Y27	V5	Y28	U6	Y29	V6	Y30	U7	Y31	V7
Y32	U8	Y33	V8	Y34	U9	Y35	V9	Y36	U10	Y37	V10	Y38	U11	Y39	V11
Y40	U8	Y41	V8	Y42	U9	Y43	V9	Y44	U10	Y45	V10	Y46	U11	Y47	V11

Banshee Frame buffer

19. Texture Memory Access

! New for AVENGER: Avenger requires special flushing to be done surrounding texture downloads. Please see section 7.3 for more information.

There are two methods of storing textures: (1) a single base address for all LODs within a texture or (2) multiple base addresses. With method (1), textures are stored as if mipmapped, even for textures containing only one level of detail. The largest texel map (LOD level 0) is stored first, and the others are packed contiguously after; for tiled space, successive LODs after 0 are stored in a manner that groups the LODs into a somewhat rectangular space; more on this later. When only some or one of the LOD levels are used, *lodmin* and *lodmax* are used to restrict texture lookup to the levels that were loaded.

With method (2), multi-base address mode, *texbaseaddr* points to where LOD level 0 starts and *texbaseaddr1*, *texbaseaddr2*, and *texbaseaddr38* point to where LOD levels 1, 2, and 3-8 start, respectively. This mode provides more granularity to texture storage, and can help texture memory allocation. There is only one base address for mipmaps 3-8, so these are stored contiguously.

Texture memory can be defined as either linear or tiled, as defined by the appropriate bit in *texbaseaddr*.

When in single base address mode, *texbaseaddr* points to where the texture would start if it contained LOD level 0 (256x* dimension). As described above, all LODs are stored contiguously after the first.

Addresses are generated by adding *texbaseaddr* and an offset that is a function of LOD, S, T, *tclamps*, *tclampt*, *tformat*, *lod_aspect*, *lod_s_is_wider*, *trexinit0*, *trexinit1*. *texbaseaddr* can be set below zero, such that the offset to the texture wraps to a positive number.

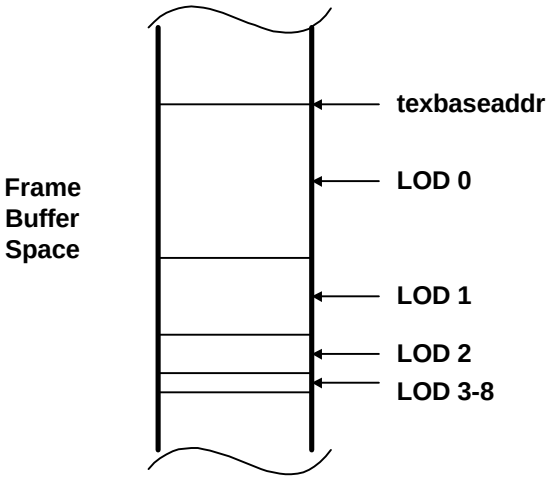
19.1 Writing to texture space

H3 provides a dedicated texture download port which is synchronized with normal rendering. Texture downloads done through this port are guaranteed to be processed in-order with 3d rendering, which alleviates the host from having to idle the chip before downloading. The texture port allows writing to frame buffer memory within a 4 byte aligned region.

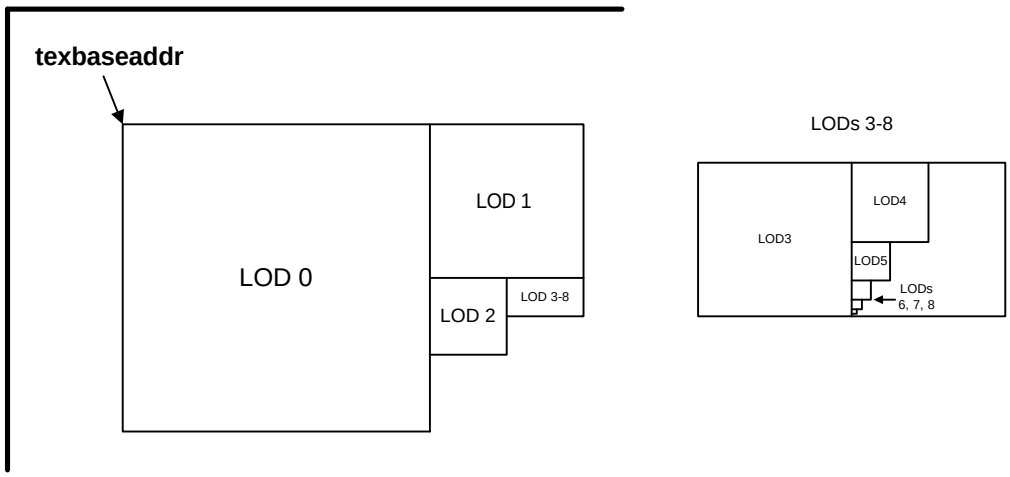
The H3 texture download port occupies 2 Mbytes of PCI address space as shown in section 4.. Note that only writes are supported through the texture download port; reads to this area return undefined data. Texture space can also be accessed (in a non-synchronized fashion) through the LFB port, which provides both read and write accesses.

Texture downloads through the texture download port are always *texbaseaddr*-relative, even in multi-base address mode. *texbaseaddr1*, *texbaseaddr2*, and *texbaseaddr38* are unused during such downloads.

linear texture space



tiled texture space



19.2 Calculating texel addresses

When downloading textures through the texture download port, a different scheme of address translation is applied, depending on whether the texture memory space is in tiled or linear space.

Linear

texture download aperture address definition

20 … 0
offset[20:0]

In linear texture space, texels are packed totally linearly starting from the base address. Any texel can be referenced by using this equation:

lfb_texel_addr[23:0] = ({texbaseaddr[23:4], 4'h0} + lod_offset[17:0] + s + t*lod_width);
(lod_offset[17:0] is the offset of the given mipmap level from the base address)

Example: Calculate the physical address for texel 30,45 in LOD 2, with aspect=1x1, 16bpp texels, and a texbaseaddr = 0x200000; assume that LODs 0,1 exist and that we are non-multi base addr.:

lfb_phys_addr[23:0] = (0x200000 + size_lod_0 + size_lod_1 + (30 + 45 * 64)*2);
= (0x200000 + (256*256*2) + (128*128*2) + (30+45*64)*2);
= 0x2296bc;
offset[20:0] = lfb_phys_addr[20:0] = 0x296bc;

You could download to this texel by setting *texbaseaddr* to 0x200000 and then writing to the lower (s is even) 16 bits of this pci address:

pci_addr[23:0] = (0x600000 + 0x296bc) = 0x6296bc;

Tiled

texture download aperture address definitions

16-bit textures

20 … 17	16 … 9	8 … 2	1 0
lod[3:0]	t[7:0]	s[7:1]	-

8-bit textures

20	19 … 16	15 … 8	7 … 2	1 0
-	lod[3:0]	t[7:0]	s[7:2]	-

In tiled texture space, LOD levels can be viewed as rectangles of texture space, which are packed edge-to-edge as described in the figure above. The aim of the packing is to make the footprint of a full texture rectangular.

Example: Calculate the physical address for texel 30,45 in LOD 2, with aspect=1x1, 16bpp texels, and a texbaseaddr = 0x200000; assume that LODs 0,1 exist and that we are non-multi base addr:

pci_download_addr[20:0] = (0x200000 + {4'h2, 8'h2d, 7'h0f, 2'b00});

= 0x245a3c;

The pci write would have to write to only the lower (s is even) two bytes of the address to insure that only texel 30, 45 is written.

Avenger will allow texture memory to be loaded from two different address spaces, the first being linear frame buffer space, and the second being the SST1 texture download port.

19.3 Maintaining read-reordering coherency in Avenger

Avenger has two TMUs, and thus two read-reordering memorys. Avenger needs explicit software intervention to maintain read-reordering memory coherency.

The rules to maintain read-reordering memory coherency with the frame buffer texture memory are as follows:

1. precede any group of texture downloads with a “pixel” flush of the 3d pipeline; ie. Issue a 2D NOPcmd to flush all pixels from the 3d section.
2. Next, perform texture downloads
3. Then, explicitly flush the read-reordering memory, by changing the value of ***texBaseAddr***. It is recommended that software write the inverse of the ***texBaseAddr*** to the register, and then re-write ***texBaseAddr*** with a correct value.
4. Now, again, do a “pixel” flush with a 2d NOPcmd. This will force all texture downloads to complete.
5. Now, procede with new rendering commands.

20. Accessing the ROM

20.1 ROM Configuration

Avenger supports either 32K or 64K of ROM space. The size of the ROM is determined during at power up by an external strapping pin (see the section of strapping pins for more information).

Directly after reset, PCI subsystem and subvendor information are loaded from the next to last four bytes of ROM memory. The last four bytes are reserved for checksum information.

20.2 ROM Reads

Avenger supports reads to the ROM through the normal PCI mechanism. In order to read the ROM, set the **romBaseAddr** register bit 0 to 1. ROM accesses are then possible at the address indicated by the most significant bits of **romBaseAddr**. ROM reads can have any combination of byte enables asserted. Since the ROM is a byte device however, asserting multilble byte enables at once will cause the transfer of data on the PCI bus to be slow.

It is important to note the ROM shares the bus with VMI and TV out. During ROM accesses, data on these ports will become ROM information providing what may appear to be bad pixels on the display. This is normal; however, if it is know that ROM accesses are to occur, it is recommended that VMI or TV out be disabled prior to ROM access.

20.3 ROM Writes

Avenger also supports a mechanism for programming flash ROMs when they are available. The model that Avenger uses is that of a 32K/64K EEPROM that allows programming by polling the EEPROM.

By default, Avenger will not respond to writes pointed at by **romBaseAddr**. By enabling bit 0 of **romBaseAddr** and also setting bit 4 of **miscInit1**, writes pointed at by **romBaseAddr** will be processed.

Typically, programmable ROMs have a sequence of write events that must occur to be placed in the 'Program Mode'. Then either a single or multiple writes occur (depending of the ROM used) to fill in data. Finally, the ROM is polled via ROM reads, to confirm the write is complete. This process is repeated until the ROM is completely written.

For more information on how to program a specific ROM, see its data sheet or application notes.

21. Power on Strapping Pins

During power up, Avenger gets some of its configuration information from strapping pins. This information is used to control how Avenger will behave.

StrapInfo bit	**PIN**	**Description**
11	VMI_ADDR_3	PLL_BYPASS (0=no bypass, 1=bypass)
10	VMI_ADDR_2	mctl_type_sdram (0=SGRAMs, 1=SDRAMs)
9	VMI_ADDR_1	mctl_short_power_on (0-normal power-on, 1=for RTL simulation onl
8	VMI_ADDR_0	re-map IDSEL (0=IDSEL is IDSEL, 1= PCI_AD_16 is IDSEL)
7	VMI_DATA_7	Disable PCI IRQ register (0=Enable, 1 = Disable).
6	VMI_DATA_6	SGRAM chip size (0=8Mb, 1=16Mb)
5	VMI_DATA_5	SGRAM number of chips (0=4 parts, 1=8 parts)
4	VMI_DATA_4	PCI Device Type (0= VGA, 1= Multimedia)
3	VMI_DATA_3	AGP Enable (0=Disabled, 1 = Enabled).
2	VMI_DATA_2	PCI 66Mhz (0 = 33Mhz, 1 =66Mhz)
1	VMI_DATA_1	BIOS Size (0=32K, 1 = 64K)
0	VMI_DATA_0	PCI Fast Device. (0=DEVSEL Medium,1= DEVSEL Fast)

22. Monitor Sense

Avenger Supports the ability to detect a monitor, as well as determine if the monitor is color or monochrome. This is accomplished with an internal MSENSE signal. MSENSE becomes active when a current is driven through either the RED, GREEN or BLUE DAC outputs. If a monochrome monitor is present, only the GREEN output will cause MSENSE to become active. MSENSE is readable through IO 0x3c2, bit 4.

23. Hardware Initialization

- PCI Configuration
- DRAM Init
- VGA Core Wakeup
- Other Init?

24. Data Formats

Signal	Pixel Sequence for 4 : 1 : 1								Pixel Sequence for 4 : 2 : 2					
P15	Y7	Y7	Y7	Y7	Y7	Y7	Y7	Y7	Y7	Y7	Y7	Y7	Y7	Y7
P14	Y6	Y6	Y6	Y6	Y6	Y6	Y6	Y6	Y6	Y6	Y6	Y6	Y6	Y6
P13	Y5	Y5	Y5	Y5	Y5	Y5	Y5	Y5	Y5	Y5	Y5	Y5	Y5	Y5
P12	Y4	Y4	Y4	Y4	Y4	Y4	Y4	Y4	Y4	Y4	Y4	Y4	Y4	Y4
P11	Y3	Y3	Y3	Y3	Y3	Y3	Y3	Y3	Y3	Y3	Y3	Y3	Y3	Y3
P10	Y2	Y2	Y2	Y2	Y2	Y2	Y2	Y2	Y2	Y2	Y2	Y2	Y2	Y2
P9	Y1	Y1	Y1	Y1	Y1	Y1	Y1	Y1	Y1	Y1	Y1	Y1	Y1	Y1
P8	Y0	Y0	Y0	Y0	Y0	Y0	Y0	Y0	Y0	Y0	Y0	Y0	Y0	Y0
P7	U7	U5	U3	U1	U7	U5	U3	U1	U7	V7	U7	V7	U7	V7
P6	U6	U4	U2	U0	U6	U4	U2	U0	U6	V6	U6	V6	U6	V6
P5	V7	V5	V3	V1	V7	V5	V3	V1	U5	V5	U5	V5	U5	V5
P4	V6	V4	V2	V0	V6	V4	V2	V0	U4	V4	U4	V4	U4	V4
P3	0	0	0	0	0	0	0	0	U3	V3	U3	V3	U3	V3
P2	0	0	0	0	0	0	0	0	U2	V2	U2	V2	U2	V2
P1	0	0	0	0	0	0	0	0	U1	V1	U1	V1	U1	V1
P0	0	0	0	0	0	0	0	0	U0	V0	U0	V0	U0	V0

25. Test Requirements

26. Issues/Requirements

26.1 PCI/AGP requirements

- Devsel needs to decoded for VGA
- Add 8bit xfers
- Add AGP bus master.
- Add interrupt logic
- Incorporate VGA information into PCI CFG space
- No wait state palette writes

26.2 2D requirements (SST-G)

- Binary / Ternary Rasterops
- 8x8x24 patterns
- Blits, src stride and dst stride. Stretch blits
- 1:n color expansion (text)

- lfb byte writes
- YUV - RGB color space conversion
- SGRAM fast fill
- Lines / Tri's and rects use FBI fastfill / triangles.
- support 8bit, 16bit, and 24bit color formats

26.3 Video / Monitor requirements

- Anti Dither logic?
- Extra Gamma logic for 8bit desktop.
- 135 Mhz DAC (1280x1024 resolution)
- Triple 256x8 lookup tables
- Hardware Cursor
  64x64x2 windows compatible
  Cursor image data stored in offscreen memory
  single 128 bit internal cache stores current scanline cursor information
  Cursor scanline is read during active hsync
  registers required curXpos, curYpos, curCtrl, CurC0, CurC1
  Cursor registers live in the pci domain
  X compatibility?
- Window ID information for entire screen is RLE encoded and stored in offscreen memory
- Each window can be single or double buffered
- Each window can be YUV or 16 bit RGB format color
- Bilinear filtering in both X and Y support for window magnification
- Decimation (point sampling) for window minification
- Windows desktop is a special case (wid 0) and is single buffered, palettized or 16/24 RGB only
- 4 or 8 or 16 unique window Ids supported
- wid requires widRowStart(buffer0), widRowStart(buffer1), widRowStart(buffer2), widCtrl, widXStart
  widYStart, widXSize, widYSize, wid_dudx, wid_dvdy
- Single 8 bit overlay with transparency?
- Pixel replication / line duplication for lower rez support (320x240)
- Software control of vsync/hsync
- DDC compliance
- vsync interrupts
- Genlock to external video source (tristate hsync/vsync controls)
- invert hsync/vsync
- LCD shutter glasses support
- Interlaced video output support
- Filtered interlaced video output support?
- Video in? S3 scenic Highway? VESA video In Port
- Support for VBE2.X LFB modes / and functions?

26.4 VGA Controller requirements

- Palette snooping
- Palette control in PCI space.
- Relocatable VGA extension rom
- Needs to be disabled.
- Supports all VGA lfb modes including 4 bit planar.

26.5 Memory Controller requirements

- Combined Texture, 2D, 3D(color & auxillary), Video
- 128 bit wide
- Support SGRAM write per bit, block write. Continued support for EDO
- 3D in a window
- Bank / port swizzling for 2D and texture performance
- Read-reordering memory.
- Arbitrate between VGA, 2D, FBI, TMU
- Packed 24 bit mode in addition to 8 bit and 16 bit pixel formats
- Tiled, interleaved bank memory organization for optimal performance.

26.6 Configuration Eeprom

- Serial Eeprom for storing video configuration
- Support for VGA rom expansion.

26.7 Dac requirements

- Must support pixel format switching per pixel
- Must support triple CLUT
- Must support >= 135Mhz pixel frequency.
- Must support 8 bit psuedo color lookup.
- Must reset to 8 bit psuedo color lookup
- VGA wants 18 bit clut writes.
- Support monitor sensing for VGA
- SAR at clut output for diagnostic/testability.
- Dac CLUT Addr (24bits) DAC CLUT data (18-24bits)

26.8 PLL requirements

- Must support video frequencies >= 135Mhz
- Graphics clock must reset >= 50Mhz < 75Mhz
- Video clock must reset = = 25.175 (VGA)
- PLL must support the following 2 frequencies in HW 25.175, and 28.322 Mhz for VGA
- Requires M / N register pair per PLL. M is 8 bits, N is 8 bits. (14.318Mhz source)
- PLL output test port.

26.9 Overall requirements

- Avenger must reset to VGA mode with no software help.
- Power down support? VBE 2.0 APM document?

26.10 PC97 requirements

- Primary graphics adapter does not use legacy bus
- Support for NTSC/PAL TV
- Support for multiple adapters / monitors
- Minimum resolution 1024x768x16
- Graphics operations use relocatable registers only
- Graphics adapter operates normally with default VGA mode driver (4 bit planar)
- ergonomic timing rates per current VESA specification: 75Hz
- Color ordering rgb most significant is red least significant is blue, bpp 15, 16, 24, 32

- Downloadable RAMDAC entries to support image color matching (gamma correction)
- Support of DDC 2.0 monitor detection
- VGA must be able to configure its bios base address to c000
- Direct frame buffer access can be performed at any time
- If supported, low resolution modes are 320x200, 320x240, 400x300, 512x384, and 640x400 all in 8 or 16 bit depths
- Hardware arithmetic stretching
- Transparent blter
- Perform double buffering with no tearing
- Current scan line of refresh
- Programmable blter stride (better memory management)
- YUV off-screen surfaces for color space conversion

26.11 Testability requirements

- Ram disable, Pll disable, blank for IDDQ.
- External access to pixel data at DAC input to verify DAC and pixel logic.
- Clock / PLL bypass for the tester. (2 pins, 1 for video, 1 for grx)
- Partial scan for coverage?

27. Revision History

0.10 Initial Draft

0.11 Split spec into multiple documents