JOEL SUDER
#1002

JAGUAR

Technical Reference Manual

12 June, 1992

Flare II Limited

The Business Centre
Station Road
Histon
Cambridge CB4 4LF

(0223) 236296
(0223) 236659 fax

1 Table of Contents

JAGUAR

2 Introduction

This document is the Jaguar Technical Reference Manual - it is intended to be a definitive reference work to the programmer's view of the Jaguar ASIC. It is neither a hardware reference work nor a guide to a particular implementation of the Jaguar design.

This document is unfinished. Further explanation and examples will be added at a later date, and many corrections will no doubt be necessary.

2.1 What is Jaguar?

Jaguar is a custom chip primarily intended to be the heart of a very high-performance games / leisure computer. It may also be used as a graphics accelerator in more complex systems, and applied to work-station and business uses.

Jaguar contains three processing units. These are:

- **Object Processor**

  The object processor is responsible for generating the display. For each display line it processes a set of commands - the object list - and generates the display for that line in an internal line buffer.

  Objects may be bit maps in a range of display resolutions, they may be scaled, conditional actions may be performed within the object list, and interrupts to the Graphics Processor may be generated.

- **Graphics Processor**

  The Graphics Processor is a very fast micro-processor which is optimised for performing graphics generation. It has its own local RAM, and a powerful ALU including fast multiply and divide operations.

- **Blitter**

  The Blitter is closely coupled to the GPU, and is able to rapidly move and fill graphical objects in memory. It includes hardware support for Z-buffering and shading at very high speed.

Jaguar provides these blocks with a 64-bit data path to external memory devices, and is capable of a peak transfer rate of 160 Mbytes / second into external dynamic RAM.

2.2 Jaguar Specification

Key Features

* 40 MIPs Graphics Processor
* Blitter capable of drawing 1 Billion Pixels per second
* Pixel Clocks up to 40 MHz
* 24-bit true colour
* Interlaced or non-interlaced displays
* Object list processor for display generation
* Windows with 1,2,4,8 & 16 bits per pixel on the same screen
* Hardware Gouraud shading and Z-buffering
* Ultra-fast shaded polygon drawing
* 16 or 32 bit microprocessors
* 8, 16, 32 or 64 bit memory interface
* Two banks of ROM, each up to 16 Mbyte
* Two banks of (page mode) DRAM, each up to 32 Mbyte
* Support for low cost highly integrated peripherals
* System capable of de-compressing images in real time
* 720 pixel/line suited to broadcast quality digital TV

3 Jaguar video and object processor

3.1 Overview

The Jaguar video section has been designed to drive a PAL/NTSC TV. The display has a horizontal resolution of 720 pixels and a vertical resolution of about 220 lines non-interlaced or 440 lines interlaced. However by adopting a flexible approach to the design the chip can be used with a range of display standards through VGA to Workstation. This will allow the chip to become the backbone of many (possibly unforseen) products.

Two colour resolutions are supported, 24 bit RGB and our own standard 16 bit CRY (Cyan, Red, Intensity). The 24 bit mode is useful for applications requiring true colour. The 16 bit mode is designed for animation. It consumes less memory, fits better into 64 bit memory, is simpler to shade and is almost indistinguishable from 24 bit mode.

Jaguar decouples the pixel frequency from the system clock by using a line buffer. This means that the system clock does not have to be related to the colour carrier frequency and may be unaffected by gen-locking. There are actually two line buffers one is displayed while the other is prepared by the object processor. Each line buffer is a 360 x 32 bit RAM which is cycled at 40 MHz. The line buffer contains physical pixels these may be either 16 bit CRY pixels or 24 bit RGB pixels. The line buffers may be swapped over at the start and in the middle of display lines.

The 16 bit CRY pixels at the output of the line buffer are converted to 24 bit RGB pixels using a combination of look-up tables and small multipliers. Eight bit resistor ladders with emitter followers will be used as video DACs.

The video timing is completely programable in units of the pixel clock. The pixel clock can be up to 40 MHz although there is provision for use with an external multiplexer. For TV applications the pixel clock will be in the range 12 to 15 MHz. The pixel clock will be synthesised from the chroma carrier or from an external video source using a device like the MC1378. Eight bits per pixel at up to 160 MHz can be supported by using an external multiplexer, colour-look-up and DAC.

Jaguar uses an object processor similar to PANTHER. Object processors combine the advantages of frame store and sprite based architectures. Jaguar's object processor is simpler yet more sophisticated than PANTHER. It shares some of the object types found on PANTHER (scaled-bit-map, branch and interrupt) but it does not have the others (run_length, move immediate, add immediate, move indirect and palette loads). Instead it can interrupt the graphics processor to perform more complex operations on its behalf. The graphics processor will support perspective, rotation, branches, palette loads, etc.

The object processor can write into the line buffer at up to 80 million pixels/sec. The source data can be 1,2,4,8,16 or 24 bits per pixel. Except for 24 bits, objects of different colour resolutions can be mixed. The low resolution objects, one to eight bits, use a palette to obtain a 16 bit physical colour.

A sophistication in the object processor is that it can modify the existing contents of the line buffer with another image. This could be used to produce shadows, mist or smoke, coloured glass or say the effect of a room illuminated by flashlamp.

The object processor can also ignore data which is stored alongside pixel data. If, for instance, a Z buffer is needed then this can be situated next to the pixels. This helps because DRAM RAS pre-charges are needed less frequently.

3.2 Object Processor Performance

Each object is described by an object header which is two phrases for an unscaled object and three phrases for a scaled object. When an image has been processed the modified header is written back to memory.

The object processor fetches one phrase (64 bits) of video data at a time. This phrase is expanded into pixels (and written into the line buffer) while the next phrase is fetched.

Image data consists of a whole number of phrases. The image data may need to be padded with transparent pixels (colour zero in 1,2,4,8 & 16 bit modes).

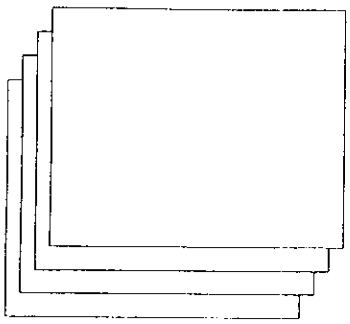
The object processor writes into the line buffer at 40 MHz. In 24-bits-per-pixel mode and for scaled objects one pixel is written per cycle. For unscaled objects with 16 or fewer bits-per-pixel two pixels are written per cycle. Most objects will therefore be expanded at 80 million pixels per second.

If the read-modify-write flag is set in the object header the object data is added to the previous contents of the line buffer. In this case the data rate into the line buffer is halved.

This peak rate may be reduced if the memory bandwidth is not high enough. However if 64-bit wide DRAM is installed then these data rates will be sustained for all modes.

When accessing successive locations in 64-bit wide DRAM the memory cycle time is 50 ns. These are page mode cycles. When the DRAM row address must change there is an overhead of between 125-225 ns (depending on DRAM speed). These RAS cycles will occur infrequently during object data fetches but will typically occur during the first data read after reading the object header (because the header and image data will not normally be near each other in memory). RAS cycles will also occur after refresh cycles or if a bus master with a higher priority steals some memory cycles in an area of memory with a different row address. Refresh cycles will normally be postponed until object processing has completed.

To get an idea of how this could be used to support a window based operating system. I will calculate roughly how many overlapping windows can exist on the same screen line. Assume a screen with N identical overlapping 8 bit windows as shown:-



As a rough approximation assume that only the first phrase of each underlying window is shown and that the top window displays 720 pixels.
Each window has a two phrase object header which must be read and written back and will probably incur a RAS cycles. Each of the underlying windows will take about 22 cycles to process (2 read, 2 read, 7 RAS, 2 read, 7 RAS, 2 write). The uppermost window will take an additional 360 cycles to generate 720 pixels.

If the object processor is active for the whole line (64 Os or 2560 cycles) then the number of windows which can be displayed is:-

N = (2560 - 360) / 22
N = 116

3.3 Memory controller

Jaguar's memory controller is designed to be very fast and flexible. It is designed to hide the memory width, speed and type from the other parts of the system.

Memory is grouped into banks which may be of different widths, speeds and types (although both ROM banks have the same width and speed). Each bank is enabled by a chip select. In the case of DRAM there are two chip selects RAS & CAS. Memory widths can be 8,16,32 or 64 bits wide but the memory controller makes it all look 64 bits wide.
There are eight write strobes one for each eight bits. There are three output enables corresponding to d[0-15],d[16-31] and d[32-63]. Three memory types are supported DRAM, SRAM and ROM.

ROM/EPROM will be used for bootstrap and for cartridges. The ROM speed is programmable. The memory controller allows the system to view ROM as 64 bits wide. Pull-up / pull-down resistors determine the ROM width during reset.

DRAM is the principal memory type. It is cheap and fast (when used in fast page mode). In fast page mode the DRAM is cycled at 20 MHz: this requires an 85 ns or 100 ns DRAM depending on the manufacturer. The row time access is programmable. The column access time is not programmable and can only be adjusted by changing the system clock. (A page mode cycle takes two clock ticks). The memory controller decides on a cycle by cycle basis whether the next cycle can be a fast page mode cycle. Data and algorithms should be organised to minimise the number of page changes.

Static RAM (SRAM) with speeds between 50 ns and 200 ns is also supported. SRAM might be used in cartridges or as a way of accelerating the system.

There are five memory banks; two ROM, two DRAM and one SRAM.

3.4 Microprocessor Interface

JAGUAR has been designed to work with any 16 or 32 bit microprocessor with (up to) 32 address lines. The interface is based on the 68030 but most microprocessors can be attached by using a PAL to synthesize those control signals which differ. All peripherals are memory mapped; there is no separate IO space.

The width of the microprocessor is determined during reset by a pull-up / pull-down resistor. Variations in the address of the cold boot code/vector is accommodated by making the bootstrap ROM appear everywhere until the memory configuration is set up by the microprocessor.

The microprocessor interface is generally asynchronous so the clock speeds of the microprocessor and co-processors may be independent.

The DSP uses the same microprocessor interface.

The CPU normally has the lowest bus priority but under interrupt its priority is increased above that of the graphics processor.

The following list gives the priorities of all bus masters.

Highest priority	Higher priority daisy-chained bus master
	Refresh
	GPU / High priority DMA (un-buffered peripherals)
	Object processor
	DSP
	CPU under interrupt
	GPU
	Blitter
Lowest priority	CPU

The graphics processor (GPU) services a DMA request as an interrupt. The bus priority is determined by the interrupt service routine.

3.5 Digital Sound Processor (DSP)

The DSP is a separate chip to the Jaguar video/system chip. This allows the DSP to be used in other products (possibly in a standalone synthesizer).

The DSP design uses the same design and instructions as the core of the graphics processor. The graphics specific parts of the graphics processor (the blitter and the systolic logic) are discarded and replaced by sound specific logic (DACs and programmable interrupt timers). The DSP has a bus interface which resembles the 68030 while the GPU uses an internal synchronous 64 bit bus.

The DSP runs at up to 40 MIPs. This gives it the capability of synthesizing over 80 FM operators or 64 amplitude & pitch modulated sampled voices.

Because ROM is at least seven times more area efficient than RAM on standard cell cost can be saved by replacing RAM with ROM containing standard synthesis & maths subroutines. If the DSP has a reasonable amount of ROM it should be possible for naive programmers to treat the DSP as a 'black box' rather than a processor which has to be coded. This should do a lot to overcome programmers reluctance to use the DSP.
Until these subroutines are developed the first version of the DSP will contain only RAM.

3.6 Falcon mode

In Falcon mode Jaguar has no external memory of its own. All external memory accesses are controlled by an external memory controller and Jaguar must become bus master before it can access the memory. The external bus is either 68030 or 68040 but Jaguar has access to the full 64 bit memory data bus. The graphics processor can execute programs out of its local memory without becoming bus master. Jaguar's internal memory is available to the rest of the system between addresses FFFE0000 and FFFEFFFF.

In Falcon mode the Jaguar graphics processor is used to enhance the graphics capability of the system. The Jaguar video generator and object processor are unused. In Falcon mode the bus priorities are as follows:-

Highest priority	Higher priority daisy-chained bus masters
	CPU under interrupt
	Graphics processor
	Blitter
	Lower priority daisy-chained bus masters
Lowest priority	CPU

Jaguar's mode is determined by input pins.

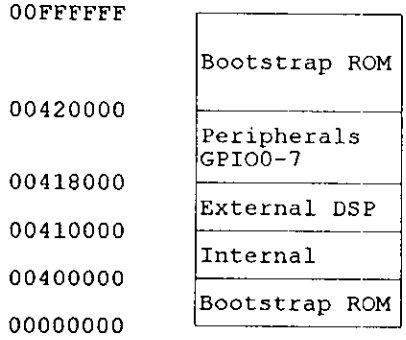
3.7 Memory Map

Jaguar's memory map depends on how it is being used.

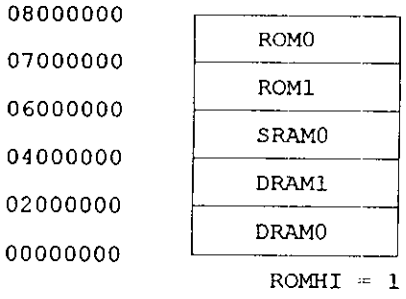
In Falcon mode the Jaguar responds to addresses between FFFE0000 and FFFEFFFF.

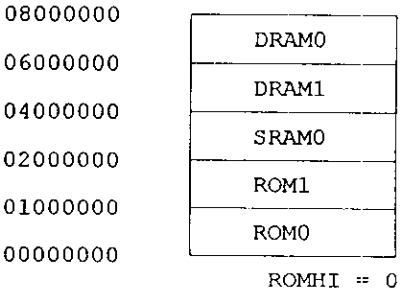
In Jaguar mode the 128 Mbyte map is repeated 32 times throughout the 4 Gbyte microprocessor address space.

Following reset the following 16 Mbyte window is repeated throughout the 128 Mbyte window until memory is configured by the microprocessor. (This allows the system to boot whether the microprocessor is a 680X0, an 80X86 or a Transputer.) After configuration, this map corresponds to the area defined as ROM0 on the maps below.



When the memory configuration is set one of two memory maps is selected depending on bit ROMHI of the memory configuration register.





ROM0 is the bootstrap ROM but internal (ASIC) memory and peripherals occupy 128 Kbytes of this space. ROM1 is the cartridge ROM. RAM0 is the cartridge static RAM. DRAM0 and DRAM1 are the two banks of DRAM.

Internal Memory Map

Internal Memory is mostly 16 bits wide to allow operation with 16 bit microprocessors. The addresses shown are offsets relative to the 64K window occupied by internal memory. In FALCON mode the addresses are relative to the 64K window FFFE0000 to FFFEFFFF.

32 bit write cycles are allowed to some areas of internal memory notably the line buffer and the graphics processor memory. The line buffer support 32 bit writes primarily in order to accelerate blitter writes to the line buffer. The graphics processor supports 32 bit writes to accelerate program and data loads.

MEMCON1 First Memory Configuration Register 0h RW

Bit 0	ROMHI	When set the two ROM decodes address the top 16M within the 64M window. When clear the ROM decodes address the bottom 16M.
Bits 1,2	ROMWIDTH	Specifies the width of ROM 0 -> 8 bits, 1 -> 16 bits, 2-> 32 bits, 3 -> 64 bits
Bits 3,4	ROMSPEED	Specifies the ROM cycle time 0 -> 250 ns, 1 -> 200 ns, 2 -> 150 ns, 3 -> 120 ns
Bits 5,6	DRAMSPEED	Specifies the DRAM Speed. The page mode cycle time is always two clock cycles. These bits determine RAS related timing as follows:

Bits 5,6	Precharge	RAS to CAS	Refresh
0	4	3	5
1	4	3	4
2	3	2	4
3	2	1	3

The times are clock cycles (nominal 40 MHz).

Bits 7,8	SRAMWIDTH	Specifies the static RAM Width 0 -> 8 bits, 1 -> 16 bits, 2-> 32 bits, 3 -> 64 bits
Bits 9,10	SRAMSPEED	Specifies the static RAM speed 0 -> 200 ns, 1 -> 150 ns, 2 -> 100 ns, 3 -> 50 ns
Bits 11,12	IOSPEED	Specifies the speed of external peripherals 0 -> 450 ns, 1 -> 250 ns, 2 -> 100 ns, 3 -> 50 ns
Bit 13	NOCPU	Indicates that there is no microprocessor and that the graphics processor should boot from ROM at address 0.
Bit 14	CPU32	Indicates that the microprocessor is 32 bits
Bit 15	FALCON	Falcon mode.

All the ROMSPEED bits are set to zero on reset. ROMHI, ROMWIDTH, CPU32, NOCPU & FALCON are determined by external pull-up / pull-down resistors. All the other bits are undefined. ROM0 repeats every 16 Mbytes until this register is written to.

MEMCON2 Second Memory Configuration Register 2h RW

Bits 0,1	COLS0	Specifies number of columns in DRAM0 0 -> 256, 1 -> 512, 2-> 1024, 3-> 2048
Bits 2,3	DWIDTH0	Specifies the width of DRAM0 0 -> 8 bits, 1 -> 16 bits, 2-> 32 bits, 3 -> 64
Bits 4,5	COLS1	Specifies number of columns in DRAM1 0 -> 256, 1 -> 512, 2-> 1024, 3-> 2048
Bits 6,7	DWIDTH1	Specifies the width of DRAM1 0 -> 8 bits, 1 -> 16 bits, 2-> 32 bits, 3 -> 64
Bits 8-11	REFRATE	Specifies the refresh rate. DRAM rows are refreshed at a frequency of CLK / (64 x (REFRATE+1)). Many DRAM chips require a refresh frequency of 64 KHz. Refresh cycles occur at the end of object processing. If REFRATE is zero refresh is disabled.
Bit 12	BIGEND	Specifies that big-endian addressing should be used. This determines the address of a byte within a phrase and allows JAGUAR to be used comfortably with Big-endian (Motorola) processors or with Little-endian (Intel) processors.
Bit 13	HILO	Specifies that image data should be displayed from high order bits to low order.

All the above bits are undefined on reset except BIGEND which is determined by external pull-up / pull-down resistors.

HC Horizontal Count 4h RW

This register comprises of a ten bit counter which counts from zero up to the value in the horizontal period register twice per video line. An eleventh bit determines which half of the display is being generated. The counter is incremented by the pixel clock. The vertical counter is incremented every half line in order to support interlaced displays. This register is only for ASIC test purposes.

VC Vertical Count 6h RW

This register comprises of an eleven bit counter which counts from zero up to the value in the vertical period register once per field. A twelfth bit determines which field (odd/even) is being generated. The counter is incremented every half line. This register can be read to do beam synchronous operations. It is only writable for ASIC test purposes.

LPH Horizontal Light-pen 8h RO

This read only eleven bit register gives the horizontal position in pixels of the light-pen.

LPV Vertical Light-pen 0Ah RO

The low eleven bits of this register gives the vertical position of the light-pen in half lines. The most significant bit is cleared at the end of vertical sync and set when a pulse is received on the light-pen input.

CLK1 System Clock Frequency 0Ch WO

This ten bit register programmes a divider which may be used to synthesize the system clock. An external phase comparator is connected to the output of the divider and also to 1/64 of the chroma crystal frequency. The output of the phase comparator drives a VCO which generates the system clock. The above phase-locked-loop (PLL) synthesizes a system clock with a frequency given by (N+1)*CHROMA/64 where N is the value written into CLK1 and CHROMA is the crystal frequency. The value of this register is forced to one on reset.

CLK2 Video Clock Frequency 0Eh RW

This register is similar to CLK1 except that it defines the pixel clock and that it is not defined on reset.

OB[0-3] Object Code 10h-16h RO

These four registers allow the graphics processor to read the current object. This allows the graphics processor object to pass parameters to the GPU interrupt service routine.

OLP1 Object List Pointer (less significant word) 20h WO
OLP2 Object List Pointer (more significant word) 22h WO

This points to the start of the object list. All objects must be on a phrase boundary so the bottom three bits are always zero. When one object links to another bits 3 to 22 of this address are replaced by the LINK data in the object.

ODP Object processor data pointer 24h WO

The top nine bits of this register form the top nine bits of object data addresses ie A[23-31]. Within an object image data is specified by the 20 bit DATA field. This specifies bits 3 to 22 of the address. Image data must lie on a phrase boundary so the bottom three bits are zero.

OBF Object processor flag 26h WO

Bit zero of this register can be tested by the object processor branch instruction. If set the branch is taken, if clear execution continues with the next object. This flag is intended as a mechanism for letting the graphics processor control the object processor program flow.

VMODE Video Mode 28h WO

Bit 0	VIDEN	When set enables time-base generator.
Bits 1,2	MODE	Determines how the line buffer contents are translated into physical pixels.
	0	16 bit CRY. Each 32 bit entry in the line buffer is treated as two 16 bit CRY pixels on successive clock cycles. Each is converted into eight bits of red, green & blue using a combination of lookup tables and multipliers.

	1	24 bit RGB. Each 32 bit entry in the line buffer is treated as one physical pixel with eight bits of red, eight bits of blue, eight bits of green and eight bits unused.
	2	16 bit direct. Each 32 bit entry in the line buffer is divided into two 16 bit words which are output directly onto the red and green outputs on alternate phases of the video clock. This mode is for applications requiring a dot clock in excess of 40 MHz. It is assumed that further multiplexing and colour lookup will occur outside the chip. In this mode blanking and video_active are output on the two least significant bits of blue.
	3	16 bit RGB. Each 32 bit entry in the line buffer is treated as two 16 bit RGB pixels. Bits [0-5] are green, bits [6-10] are blue and bits [11-15] are red.
Bit 3	GENLOCK	When set this bit enables digital genlocking. This means that external syncs will reset the internal time-base generators. On its own this mechanism does not give satisfactory genlocking because there is a one pixel jitter. However this mechanism is used to quickly lock onto a new video source. An external Phase Locked Loop is required for true genlocking.
Bit 4	INCEN	Enables encrustation. When set the least significant bit of the CRY intensity is used to switch between local and external video sources using an external video multiplexer. This allows the video source to be switched on a pixel by pixel basis.
Bit 5	BINC	Selects the local border colour if encrustation is enabled.
Bit 6	CSYNC	Enables composite sync on the vertical sync output.
Bit 7	BGEN	Clears the line buffer to the colour in the background register after displaying the contents. This only has effect in CRY and RGB16 modes.
Bit 8	VARMOD	Enables variable colour resolution mode. When this bit is set the least significant bit of each word in the line buffer is used to determine the colour coding scheme of the other 15 bits. If the bit is clear the bits the word is treated as a CRY pixel. If the bit is set then bits [1-5] are green, bits [6-10] are blue and bits [11-15] are red. This mechanism allows JAGUAR to support an RGB window against a CRY background for instance.
Bits 9-15	Unused	Write zeroes.

BORD1 Border Colour (Red & Green) 2Ah WO
BORD2 Border Colour (Blue) 2Ch WO

These registers determine the physical border colour. There are eight bits per primary colour. Red is the less significant byte of BORD1. This colour is displayed between the active portions of the screen and blanking. It is not necessary to display a border. The border area is defined by the video time-base registers.

HP Horizontal Period 2Eh WO

This ten bit register determines the period of half a display line in pixels. The period is one tick longer than the value written into this register.

HBB Horizontal Blanking Begin 30h WO

This eleven bit register determines the start position of horizontal blanking. The most significant bit is usually set because blanking starts in the second half of the line.

HBE Horizontal Blanking End 32h WO

This eleven bit register determines the end position of horizontal blanking. The most significant bit is usually clear because blanking ends in the first half of the line.

HS Horizontal Sync 34h WO

This eleven bit register determines the width of the horizontal sync and equalization pulses. The pulses start when the horizontal count equals the value in the register. The pulses end when the horizontal count equals the horizontal period. The most significant bit is usually set because horizontal sync happens at the end of the line. The most significant bit is ignored in the generation of equalization pulses which are the same width as horizontal sync but which appear twice per line (for 10 half lines during field blanking).

HVS Horizontal Vertical Sync 36h WO

This ten bit register determines the end position of the vertical sync pulses. Vertical Sync consists of long sync pulses for several half lines. These pulses are generated twice per line. Vertical sync starts at the same time as the horizontal sync or equalization pulses but end when the least significant ten bits of the horizontal count match the HVS register.

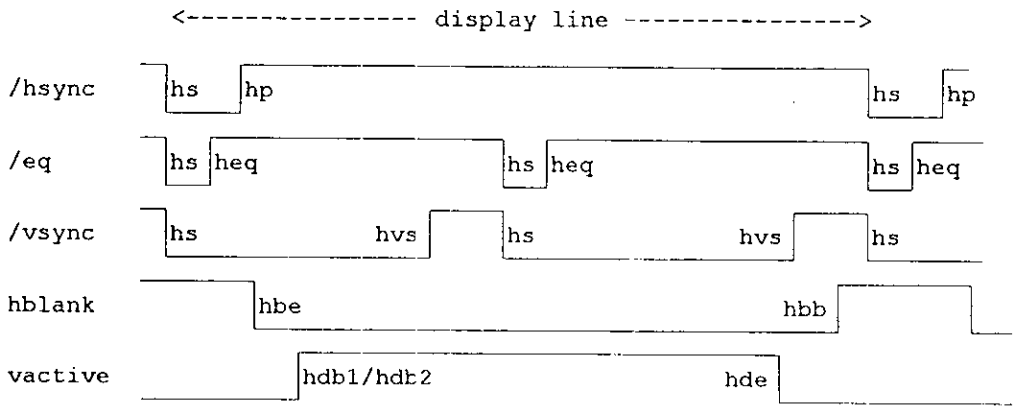
HDB1 Horizontal Display Begin 1 38h WO
HDB2 Horizontal Display Begin 2 3Ah WO

These eleven bit registers control where on the display line the object processor starts. When the horizontal count matches either of the above registers the object processor starts execution at the address in OLP, the line buffers swap over and pixels are shifted out of the line buffer. The object processor can run twice per line in order to support display modes where the amount of data on a display line is greater than can be contained in one line buffer. The line buffers are each 360 words x 32 bits. If the display mode was 720 x 24 bits per pixel then line buffer A might be displayed at the start of the line while buffer B was being written. Then during the second half of the display line buffer B would be displayed while line buffer A was prepared for the next line. In this case HDB1 would contain a value corresponding to the left hand edge of the display and HDB2 would contain a value corresponding to the middle of the display. If the object processor needs to run only once per line then either the registers take the same value or one register is given a value greater than the line length.

HDE Horizontal Display End 3Ch WO

This eleven bit register specifies when the display ends. Either border colour or black (if HBB < HDE) is displayed after the horizontal count matches this register.

The relative positions of some of the above signals and the registers which define them are shown on the following diagram.



!!! example setup required

VP **Vertical Period** **3Eh** **WO**

This eleven bit register determines the number of half lines per field. The number is one more than the value written into this register. If the number of half lines is odd then the display is interlaced.

VBB **Vertical Blanking Begin** **40h** **WO**

This eleven bit register specifies the half line on which vertical blanking begins.

VBE **Vertical Blanking End** **42h** **WO**

This eleven bit register specifies the half line on which vertical blanking ends.

VS **Vertical Sync** **44h** **WO**

This eleven bit register specifies the half line on which vertical sync begins. Vertical sync pulses are generated from this line to the line specified by the vertical period.

VDB **Vertical Display Begin** **46h** **WO**

This eleven bit register specifies the half line on which object processing begins. Object processing restarts on every line until the half line specified by the VDE register. The border colour (or black) is displayed outside these active lines.

VDE **Vertical Display End** **48h** **WO**

This eleven bit register specifies the half line at which object processing ends.

VEB **Vertical Equalization Begin** **4Ah** **WO**

This eleven bit register specifies the half line on which equalization pulses start.

VEE Vertical Equalization End 4Ch WO

This eleven bit register specifies the half line on which equalization pulses end.

VI Vertical Interrupt 4Eh WO

This eleven bit register specifies a half line on which the microprocessor is interrupted.

PIT[0-1] Programmable Interrupt Timer 50-52h WO

These two 16 bit registers control the frequency of interrupts to the CPU and to the GPU. PIT[0] & PIT[1] operate as a pair controlling the interrupts.

The system clock (nominally 40MHz) is divided by (one plus the value in the first register). If the first register contains zero the timer is disabled. The resulting frequency is divided by (one plus the value in the second register) and the output of this divider generates the interrupt.

HEQ Horizontal equalization end 54h WO

This ten bit register determines the end position of the equalization pulses. Equalization consists of short sync pulses for several half lines on either side of vertical sync. These pulses are generated twice per line.

BG Background Colour 58h WO

This register specifies the CRY colour to which the line buffer is cleared.

INT1 Interrupt Control Register 0E0h RW

This register enables, identifies and acknowledges interrupts from the six different interrupt sources. The interrupts sources are as follows:

0	Video	This interrupt is generated by the video time-base at field rate on the display half-line specified in register VI. The interrupt is generated at the right hand edge of the picture.
1	GPU	This interrupt is generated by the graphics processor writing to an internal register.
2	Object	This interrupt is generated by stop objects.
3	Timer	This interrupt is generated by the programmable timer.
4	DSP	This interrupt is generated by an input to the JAGUAR chip and is intended for use by the DSP. This is an active high edge-triggered interrupt - the first interrupt will occur on the first rising edge after it has been enabled.
5	External	This interrupt is generated by an input to the JAGUAR chip and is intended for future peripherals. This is an active high interrupt, as the DSP interrupt.

Bits [0-5] enable the individual interrupt sources i.e. if bit 1 is set the graphics processor interrupt is enabled. When read bits [0-5] indicate which interrupts are pending. i.e. if bit 5 is set there is an external interrupt pending. Bits [8-13] clear pending interrupts from the corresponding interrupt source.

INT2 Interrupt resume register 0E2h WO

When an interrupt is applied to the CPU the bus priorities of the graphics processor and blitter are reduced so that the CPU can service real time interrupts promptly. The bus priorities are restored by writing to this register.

CLUT Colour Look-Up Table 400h-7FEh

The colour look-up table translates an eight bit colour index into a 16 bit CRY colour. The eight bit index comes from the object data, which may be 1,2,4 or 8 bits. In order to achieve a high throughput there are two tables allowing two pixels at a time to be written into the line buffer. There are 256 16 bit entries in each table. Addresses in the range 400-5FE read from table A. Addresses in the range 600-7FE read from table B. Writing to either address range writes to both tables. Each 16 bit entry should be a CRY value with the intensity in the less significant byte.

LBUF Line Buffer 800h-0D9Eh
1000h-159Eh
1800h-1D9Eh

There are two line buffers each of which consists of a 360 x 32 bit RAM. Each 32 bit long-word can be read/written as two 16 bit words. In 16 bit CRY mode each word is a CRY pixel; the less significant byte is the intensity. The word with the lowest address corresponds to the left-most pixel. In 24 bit RGB mode each 32 bit long-word is a pixel. The less significant byte of the word at the lower address is the red value. The more significant byte is the green value and the less significant byte of the word at the high address is the blue value. The fourth byte is unused.

The first address range addresses line buffer A. The second addresses line buffer B. The third addresses the line buffer currently selected for writing. The first two address ranges are for test purposes the third is for the graphics processor to assist the object processor in preparing the line buffer.

By adding 8000h to the above address ranges 32 bit writes can be made to the line buffer. This is mainly to accelerate the blitter.

Peripheral Memory Map

Peripherals occupy the 64k above the internal memory. All Peripheral Memory is 16 bits wide although it is likely that many devices will have eight bit busses. Jaguar generates IORD or IOWR for accesses to locations within this window These are required for PC peripherals. The addresses shown are offsets relative to the 64k window.

DSP Digital Sound Processor 0h-7FFEh

The breakdown of the DSP's memory has yet to be decided. A lot of space has been reserved because the DSP RAM may be replaced by a lot of ROM.

GPIO[0-7] General purpose IO decodes 8000h - 0FFFEh

Jaguar has eight outputs which decode 4k byte memory spaces in this range. These will be used to control peripheral devices including joysticks, serial and parallel IO, floppy and hard disks, SCSI etc.

3.8 Object definitions

There are five basic object types

Bit Mapped Object

This object displays an unscaled bit mapped object. The object should be on a 16 byte boundary in 64 bit RAM.

First Phrase

Bits	Field	Description
[0-2]	TYPE	Bit mapped object is type zero
[3-13]	YPOS	This field gives the value in the vertical counter (in half lines) for the first (top) line of the object. The vertical counter is latched when the object processor starts so it has the same value across the whole line. If the display is interlaced the number is even for even lines and odd for odd lines. If the display is non-interlaced the number is always even. The object will be active while the vertical counter >= YPOS and HEIGHT > 0.
[14-23]	HEIGHT	This field gives the number of data lines in the object. As each line is displayed the height is reduced by one for non-interlaced displays or by two for interlaced displays. (The height becomes zero if this would result in a negative value.) The new value is written back to the object.
[24-43]	LINK	This defines the address of the next object. These 20 bits replace bits 3 to 22 in the register OLP. This allows an object to link to another object within the same 8Mbytes.
[44-63]	DATA	This defines where the pixel data can be found. Like LINK this is a phrase address. These twenty bits define bits 3 to 22 of the data address. Bits 23 to 31 are defined by register ODP. This allows object data to be positioned anywhere within an 8 Mbyte window. After a line is displayed the new data address is written back to the object.

Second Phrase

Bits	Field	Description
[0-11]	XPOS	This defines the X position of the first pixel to be plotted. This 12 bit field defines start positions in the range -2048 to +2047. Address 0 refers to the left-most pixel in the line buffer.
[12-14]	DEPTH	This defines the number of bits per pixel as follows: 0 -> 1 bit/pixel 1 -> 2 bits/pixel 2 -> 4 bits/pixel 3 -> 8 bits/pixel 4 -> 16 bits/pixel

		5 -> 24 bits/pixel
[15-17]	PITCH	This value defines how much data, embedded in the image data, must be skipped. For instance two screens and their common Z buffer could be arranged in memory in successive phrases (in order that access to the Z buffer does not cause a page fault). The value 8 * SKIP is added to the data address when a new phrase must be fetched. A pitch value of one is used when the pixel data is contiguous - a value of zero will cause the same phrase to be repeated.
[18-27]	DWIDTH	This is the data width in phrases. ie. Data for the next line of pixels can be found at 8 * (DATA+DWIDTH)
[28-37]	IWIDTH	This is the image width in phrases (must be non zero).
[38-44]	INDEX	For images with 1 to 4 bits/pixel the top 7 to 4 bits of the index provide the most significant bits of the palette address.
[45]	REFLECT	Flag to draw object from right to left.
[46]	RMW	Flag to add object to data in line buffer
[47]	TRANS	Flag to make logical colour zero and reserved physical colours transparent.
[48]	RELEASE	This bit forces the object processor to release the bus between data fetches. This should typically be set for low colour resolution objects because there is time for another bus master to use the bus between data fetches. For high colour resolution objects the bus should be held by the object processor because there is very little time between data fetches and other bus masters would probably cause DRAM page faults thereby slowing the system. External bus masters, the refresh mechanism and graphics processor DMA mechanism all have higher bus priorities and are unaffected by this bit.
[49-54]	FIRSTPIX	This field identifies the first pixel to be displayed. This can be used to clip an image. The significance of the bits depends on the colour resolution of the object and whether the object is scaled. The least significant bit is only significant for scaled objects where the pixels are written into the line buffer one at a time. The remaining bits define the first pair of pixels to be displayed. In 1 bit per pixel mode all five bits are significant, In 2 bits per pixel mode only the top four bits are significant. Writing zeroes to this field displays the whole phrase.
[55-63]		Unused write zeroes.

Scaled Bit Mapped Object

This object displays a scaled bit mapped object. The object should be on a 32 byte boundary in 64 bit RAM. The first 128 bits are identical to the bit mapped object except that TYPE is one. An extra phrase is appended to the object.

Bits	Field	Description
[0-7]	HSCALE	This eight bit field contains a three bit integer part and a five bit fractional part. The number determines how many pixels are written into the line buffer for each source pixel.

[8-15]	VSCALE	This eight bit field contains a three bit integer part and a five bit fractional part. The number determines how many display lines are drawn for each source line. This value equals HSCALE for an object to maintain its aspect ratio.
[16-23]	REMAINDER	This eight bit field contains a three bit integer part and a five bit fractional part. The number determines how many display lines are left to be drawn from the current source line. After each display line is drawn this value is decremented by one. If it becomes negative then VSCALE is added to the remainder until it becomes positive. HEIGHT is decremented every time VSCALE is added to the remainder. The new REMAINDER is written back to the object.
[24-63]		Unused write zeroes.

Graphics Processor Object

This object interrupts the graphics processor, which may act on behalf of the object processor. The object processor resumes when the graphics processor writes to the object flag register.

Bits	Field	Description
[0-2]	TYPE	GPU object is type two
[3-13]	YPOS	This object is active when the vertical count matches YPOS unless YPOS = 03FF in which case it is active for all values of vertical count.
[14-63]	DATA	These bits may be used by the GPU interrupt service routine. They are memory mapped so the GPU can use them as data or as a pointer to additional parameters.

Execution continues with the object in the next phrase. The GPU may set or clear the (memory mapped) object processor flag and this can be used to redirect the object processor using the following object.

Branch Object

This object directs object processing either to the LINK address or to the object in the following phrase.

Bits	Field	Description
[0-2]	TYPE	Branch object is type three
[3-13]	YPOS	This value may be used to determine whether the LINK address is used.
[14-15]	CC	These bits specify what condition is used to determine whether to branch as follows: 0 -> Branch if YPOS == VC or YPOS == 7FF 1 -> Branch if YPOS > VC 2 -> Branch if YPOS < VC 3 -> Branch if object processor flag is set
[16-23]	unused	

[24-43]	LINK	This defines the address of the next object if the branch is taken. The address is defined as described for the bit mapped object.
[44-63]	unused	

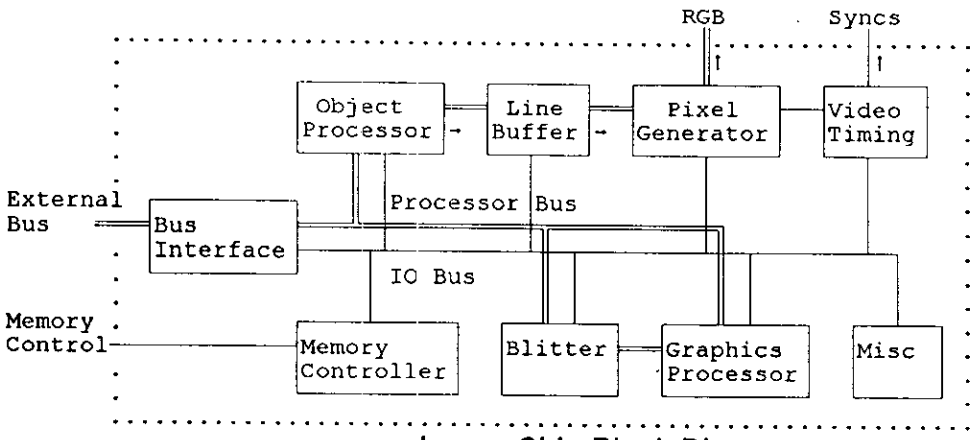
Stop Object

This object stops object processing and interrupts the host.

Bits	Field	Description
[0-2]	TYPE	Stop object is type four
[3-63]	DATA	These bits may be used by the CPU interrupt service routine. They are memory mapped so the CPU can use them as data or as a pointer to additional parameters.

3.9 Description of Object Processor/Pixel path

The following two diagrams show where the object data path fits into the Jaguar Chip. All diagrams that follow are drastically simplified for clarity.

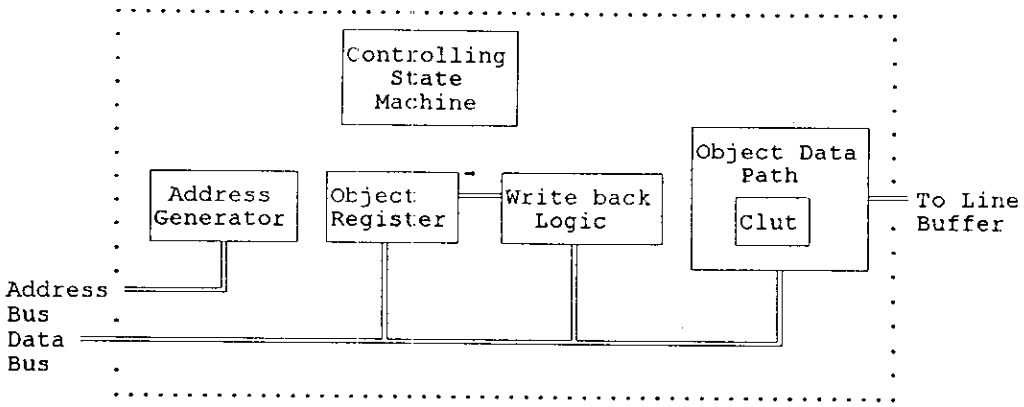


Jaguar Chip Block Diagram

The processor bus is a 64 data, 32 address multi-master bus. The bus master can change on a cycle by cycle basis with no overhead. The external CPU controls this bus when it is the bus master. The IO bus is a 16 data 16 address bus used for reading and writing to internal memory and registers. The bus interface logic and memory controller allows transfers of any width (one to eight bytes) to be made to any width of external memory. The bus interface accommodates 16 and 32 bit microprocessors. The bus interface also generates a multiplexed address for dynamic RAMs. The multiplexed address is a function of memory width and number of columns. The memory controller only performs RAS cycles when the row address changes. This allows contiguous regions of memory to be accessed three times faster.

The line buffer is a bridge between two asynchronous parts of the chip. On one side are the processors and memory. On the other side are the video timing and pixel generators. In fact there are two line buffers. While one is written into by the object processor the other is read by the pixel logic. Each line buffer is a small 360x32 RAM with independent write strobes for the high and low words.

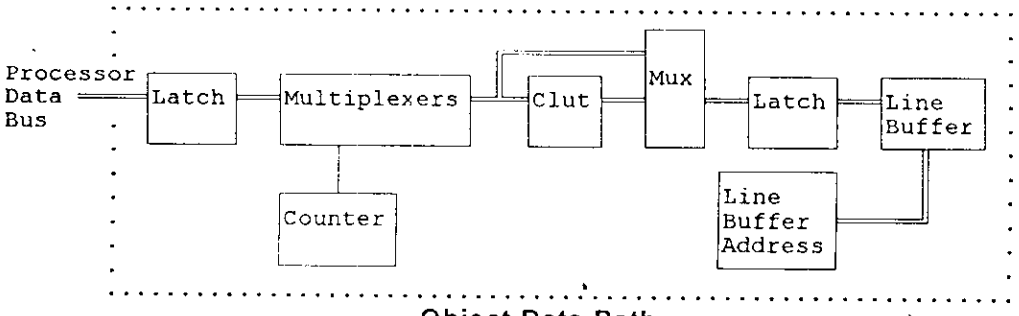
Each location in the line buffer may contain one 24 bit pixel or two 16 bit pixels.



Object Processor Block Diagram

The object processor reads object headers and image data and writes back modified headers. The write back logic normally increases the data address by the data width. If the object is scaled then the data address is increased by a multiple of the data width and the vertical remainder is modified.

The object data contains either physical colours in the case of 16 and 24 bits-per-pixel objects or logical colours in the case of 1,2,4 and 8 bits-per-pixel objects. Logical colours are translated into physical colours by the colour look up table or CLUT.



Object Data Path

The object processor fetches data one phrase at a time until the image data, for that header, is exhausted or until the line buffer address (X coordinate) has become invalid. The behaviour of the object data path depends on the colour resolution of the object (bits-per-pixel) and on whether the object is scaled.

In 24 bits-per-pixel mode each phrase contains two pixels (16 bits unused per phrase). The multiplexers select each in turn and one 24 bit pixel is written into the line buffer per clock cycle. The Clut is bypassed for 24 bits-per-pixel objects.

In 16 bits-per-pixel mode each phrase contains four pixels. The multiplexers select two pixels at a time and two pixels are written into the line buffer each clock cycle. The Clut is bypassed for 16 bits-per-pixel objects.

In 1, 2, 4 and 8 bits-per-pixel modes each phrase contains 64, 32, 16 and 8 pixels respectively. The multiplexers select two pixels at a time. In 1, 2 and 4 bit modes the pixel is made up to eight bits by taking the top bits from the top bits of the palette offset (a field in the object header). The two eight bit values are used as addresses to a pair of identical Cluts yielding two sixteen bit physical pixels which are written into the line buffer every cycle.

If an object is scaled the object processor deals with one pixel at a time not pairs. Scaling is achieved by incrementing the line buffer address independently of the counter controlling the multiplexer. For instance if the line buffer address is incremented twice as often as the counter then the image will be twice as wide.

There are two line buffers A & B. While A is written by the object processor B is being read by the pixel logic. At the start of the next display line the buffers swap over so A is displayed and B is written. This swap is effectively achieved by multiplexers on all the signals attached to the line buffers.

The above description is complicated by the following:

* If a pair of pixels must be written to an odd location in the line buffer they must be swapped and one pixel delayed.

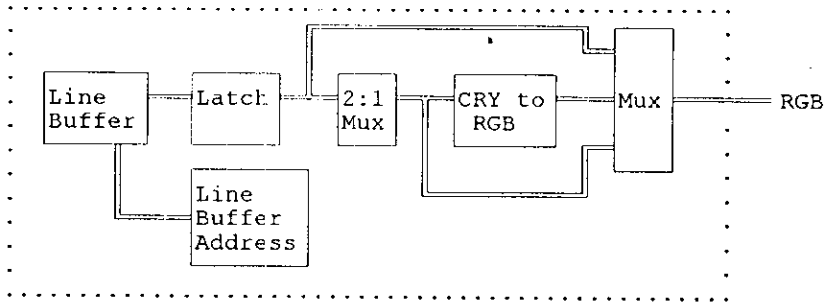
* The line buffer address decrements if the object is reflected.

* The colour to be written into the line buffer can be added to the previous value instead.

* One colour may be used as transparent and is not written into the line buffer.

* The line buffers also appear as memory to the rest of the system.

The pixel data path is shown in the following diagram. All the logic in this box runs from a different clock to the previous logic.



Pixel Data Path

The operation of the pixel data path depends on the video mode.

In 24 bits-per-pixel mode the line buffer is read at the video clock frequency. The line buffer data is simply latched and presented at the pins as red, green and blue data bits.

In CRY mode the line buffer is read at half the video clock frequency. Each read yields two 16 bit CRY values. These are multiplexed into the CRY to RGB conversion logic during succeeding video clock cycles. In this logic the more significant eight bits specify the colour and the less significant bits specify the intensity or brightness. The colour value is used as an index to three ROMs. These ROMs contain the relative amounts of red, green and blue for each colour. The outputs of the ROMs are multiplied by the brightness to get a final eight bits of red, green and blue.

In RGB16 mode the line buffer is read at half the video clock frequency. Each read yields two 16 bit RGB values. Bits 0-5 form the six most significant bits of green, bits 6-10 form the five most significant bits of blue and bits 11-15 form the five most significant bits of red. All other bits are set to zero.

In all these modes a small amount of additional logic sets the output colour to black during blanking and to the border colour where appropriate.

A fourth mode exists to allow the system to support very high pixel rates using external multiplexers and DACs. This is called direct mode. In this mode the line buffer is read at the video clock frequency and the 2:1 multiplexer is driven by the video clock directly. The output of the 2:1 mux is connected directly to the red and green outputs of the chip. This allows 16 bit values to be output at twice the maximum video clock frequency. This provides a video bandwidth of up to 160Mbytes per second. These values should be re-synchronised, de-multiplexed and converted to analogue outside the chip. In this mode the blanking and border signals are output on the blue pins.

The above picture is slightly complicated by the following:

* The least significant bit in CRY and RGB16 modes can be sacrificed (treated as zero) and used to control an external video switch through the incrust output pin.

* In CRY and RGB16 modes a background colour may be written into the line buffer after it has been read.

* In CRY and RGB16 modes the least significant bit may be used to determine whether the mode is CRY or RGB16. This could be used to drop a decompressed RGB picture into a CRY picture without having to do a RGB to CRY conversion.

3.10 Refresh Mechanism

The average refresh frequency is defined by the REFRATE bits in the MEMCON2 register. Refresh cycles are grouped together in order to lessen the impact on system performance. However they cannot performed in very large numbers or they would create "dead spots" in which no processing was possible. This could disrupt the display or sound production.

Jaguar uses a counter to accumulate a count of refresh cycles. When this counter reaches eight then eight refresh cycles are done and the counter is set to zero.

Refresh cycles are also invoked when the object processor reaches the end of the object list. After the object processor executes a STOP object JAGUAR performs as many refresh cycles as are necessary to decrement the refresh counter to zero.

This mechanism guarantees that the minimum refresh rate is maintained without interrupting the object processor and without creating "dead spots" of more than a few microseconds.

4 Colour Mapping

4.1 Introduction

Jaguar produces a video output using eight digital bits each for red, green and blue. This allows each output to have two hundred and fifty-six intensity levels, and is enough to allow smooth shading from one colour to another. This twenty-four bit scheme is known as *true-colour.*

Jaguar can produce a display based on true colour pixels stored in memory in long words, with eight bits unused, and this is known as true colour mode. However, these thirty-two bit pixels are large and so consume a lot of memory; and they also consume a lot of memory bandwidth to fetch from RAM for display.

True-colour mode is therefore unattractive for general use, as most images do not need its range of colours, and it is desirable to avoid the detrimental effects it has on performance. True colour mode is therefore a special case, and when it is used only true-colour images may be displayed.

In normal operation, the Jaguar display system is based on sixteen-bit pixels. Images in memory may be stored either as sixteen bit pixels, or may be stored as one, two, four or eight bit *logical* colours. These logical colours are used as indices into a Palette or Colour-Look-Up-Table (CLUT), which contains their corresponding sixteen-bit physical colours.

Sixteen-bit pixels may stored as six bits of green, and five bits each for red and blue, but this no longer allows smooth shading. There is therefore an additional scheme, known as the CRY scheme (cyan, red and intensity, see below) which still allows smooth intensity shading. This CRY scheme is now discussed in greater detail.

4.2 The CRY Colour Scheme

Gouraud Shading Requirements

The CRY scheme was derived principally to meet the requirements of *Gouraud Shading.* This is a technique which models the appearance of a lit curved surface from a set of polygons. If the intensity due to a light source is calculated for each polygon and the polygon is painted in that colour, then the polygons that make up that surface are each clearly visible.

The technique of Gouraud shading helps avoid this by calculating the intensity at each vertex, and then linearly interpolating along each polygon edge, and hence along each scan line that makes up the display. If only white light sources are considered, then the only variation is one of luminous intensity, and not one of colour. It is therefore attractive to have a colour scheme which contains an intensity vector, as the Gouraud shading calculations have then only to be performed for one value, rather than the three values that would have to be calculated in a true colour scheme.

As there is general agreement that eight bits is enough to give smooth intensity shading (and it is a round number), it was therefore necessary to come up with a scheme which allowed the colour to be expressed in eight bits.

Colour Space

The colour space to be modelled may be considered as the RGB cube shown, where the lowest vertex represents black, and the highest white. The three edges running out from black are the three orthogonal vectors red, green and blue. The sum of these three vectors can describe any point in the cube. The three lower vertices therefore represent fully saturated red, green and blue, and the three higher ones yellow, cyan and magenta.

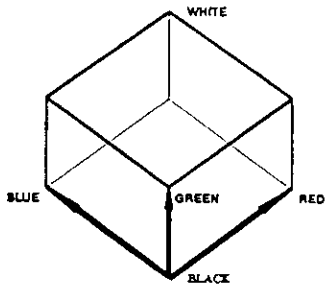


Figure 1 - RGB Cube

This colour space model is only one of many ways of considering what the human brain 'sees', but it has the advantage of modelling the display system used by colour monitors, and of being mathematically simple.

Physical requirements

The intensity vector can be considered as that component of the sum of the red, green and blue vectors which lies along the diagonal of the RGB cube from black to white. This is not the 'true' intensity, which is a weighted sum of red, green, and blue; but it bears a linear relationship to it when the colour is not changed.

It is necessary to come up with a scheme to encode the colour value in the remaining eight bits of the CRY pixel. The following requirements were made on this scheme:

1. All two hundred and fifty-six values should represent valid, and different, colours.
2. The colours should be well spread out across the colour space.
3. Colours should be able to be mixed by linearly averaging their colour values.
4. An intensity value of zero must be black.

As the remaining colour space without intensity is two-dimensional, two vectors are required to represent a point in it. An *r,* □ scheme was discarded as it would not meet requirement two, and so a scheme based on two *x, y* vectors was decided on.

To meet requirement one, the two vectors must describe a point on a square area. As no existing colour space model is square when viewed along the intensity axis, it was necessary to come up with a new one.

The approach decided on, after considerable experimentation, was to take the view along the intensity axis of the RGB cube, which is a hexagon, and distort it into a square. This does not quite meet requirement 3, but is close to it.

CRY Colour Scheme

The colour mapping scheme chosen is based on defining 256 points on the upper surface of the RGB cube.

In the figure shown, the hexagon corresponds to a view looking down onto the RGB cube. This hexagon is distorted onto a square, whose X and Y co-ordinates are four-bit values. This defines 256 colour levels. The choice of green as the primary colour which lies on the middle of one face was made after observing the effects of the three possible mappings, and corresponds with the expected result, as the human eye is least able to distinguish shades of green.

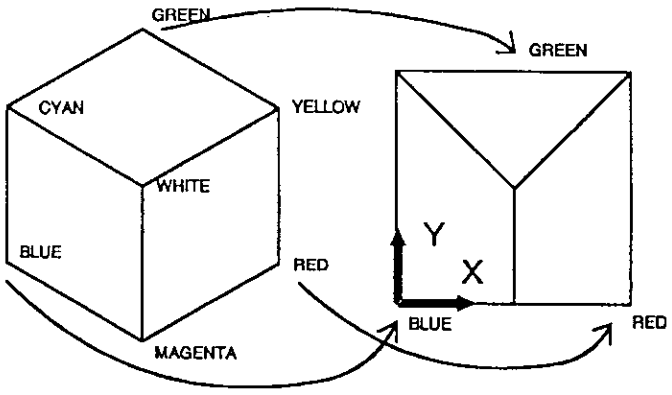


Figure 2 - RGB Mapping Scheme

Note that in each of the three areas defined on the hexagon and square, one of red, green or blue is at full intensity, and the others vary. At the centre (white) they are all at full intensity. The intensity scale for any given colour lies along the line between black, and the point on the top surface of the cube defined in the colour table.

Colours may be averaged by taking the average of their eight-bit intensity value, and each of the four-bit X and Y components of the colour value. This will not produce exactly the same colour as the point midway between them in the RGB cube, but will be close to it.

This is a summary of the pros and cons of the CRY scheme:
Advantages of CRY

- Smooth intensity shading from 16-bit pixels
- Better matched to the capabilities of the human eye than 5:6:5 bit RGB schemes
- Suitable for efficient Gouraud shading

Disadvantages

- Steps are visible in smooth changes of saturation or hue
- Translation from RGB to CRY is not straightforward
- Non-standard

Physical Implementation

The eight-bit colour value is used to index a look-up table of modifier values for each of red green and blue; which is multiplied by the intensity value to give the output level for each drive to the display. The look-up tables are:

RED	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	34	34	34	34	34	34	34	34	34	34	34	34	34	34	19	0
	68	68	68	68	68	68	68	68	68	68	68	68	64	43	21	0
	102	102	102	102	102	102	102	102	102	102	102	95	71	47	23	0
	135	135	135	135	135	135	135	135	135	135	130	104	78	52	26	0
	169	169	169	169	169	169	169	169	169	170	141	113	85	56	28	0
	203	203	203	203	203	203	203	203	203	183	153	122	91	61	30	0
	237	237	237	237	237	237	237	237	230	197	164	131	98	65	32	0
	255	255	255	255	255	255	255	255	247	214	181	148	115	82	49	17
	255	255	255	255	255	255	255	255	255	235	204	173	143	112	81	51
	255	255	255	255	255	255	255	255	255	255	227	198	170	141	113	85
	255	255	255	255	255	255	255	255	255	255	249	223	197	171	145	119
	255	255	255	255	255	255	255	255	255	255	255	248	224	200	177	153
	255	255	255	255	255	255	255	255	255	255	255	255	252	230	208	187
	255	255	255	255	255	255	255	255	255	255	255	255	255	255	240	221
	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
GREEN	0	17	34	51	68	85	102	119	136	153	170	187	204	221	238	255
	0	19	38	57	77	96	115	134	154	173	192	211	231	250	255	255
	0	21	43	64	86	107	129	150	172	193	215	236	255	255	255	255
	0	23	47	71	95	119	142	166	190	214	238	255	255	255	255	255
	0	26	52	78	104	130	156	182	208	234	255	255	255	255	255	255
	0	28	56	85	113	141	170	198	226	255	255	255	255	255	255	255
	0	30	61	91	122	153	183	214	244	255	255	255	255	255	255	255
	0	32	65	98	131	164	197	230	255	255	255	255	255	255	255	255
	0	32	65	98	131	164	197	230	255	255	255	255	255	255	255	255
	0	30	61	91	122	153	183	214	244	255	255	255	255	255	255	255
	0	28	56	85	113	141	170	198	226	255	255	255	255	255	255	255
	0	26	52	78	104	130	156	182	208	234	255	255	255	255	255	255
	0	23	47	71	95	119	142	166	190	214	238	255	255	255	255	255
	0	21	43	64	86	107	129	150	172	193	215	236	255	255	255	255
	0	19	38	57	77	96	115	134	154	173	192	211	231	250	255	255
	0	17	34	51	68	85	102	119	136	153	170	187	204	221	238	255
BLUE	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
	255	255	255	255	255	255	255	255	255	255	255	255	255	255	240	221
	255	255	255	255	255	255	255	255	255	255	255	255	252	230	208	187
	255	255	255	255	255	255	255	255	255	255	255	248	224	200	177	153
	255	255	255	255	255	255	255	255	255	255	249	223	197	171	145	119
	255	255	255	255	255	255	255	255	255	255	227	198	170	141	113	85
	255	255	255	255	255	255	255	255	255	235	204	173	143	112	81	51
	255	255	255	255	255	255	255	255	247	214	181	148	115	82	49	17
	237	237	237	237	237	237	237	237	230	197	164	131	98	65	32	0
	203	203	203	203	203	203	203	203	203	183	153	122	91	61	30	0
	169	169	169	169	169	169	169	169	169	170	141	113	85	56	28	0
	135	135	135	135	135	135	135	135	135	135	130	104	78	52	26	0
	102	102	102	102	102	102	102	102	102	102	102	95	71	47	23	0
	68	68	68	68	68	68	68	68	68	68	68	68	64	43	21	0
	34	34	34	34	34	34	34	34	34	34	34	34	34	34	19	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

JAGUAR

The tables are generated by this C procedure:

```
void init_rgb_map ()

{
int n,x,y;
float xflt,yflt,sc,radius,alpha,beta;

for (n=0; n<256; n++)
    {
    x = n/16;
    y = n%16;

    xflt = (float) (x-7.5)*(1.0/7.5);
    yflt = (float) (y-7.5)*(1.0/7.5);

    yflt = yflt * (1.0 - fabs(xflt/2.0));
    xflt = xflt * COS30;

    /* determine which triant its in */

    if ((xflt<0.0)&&((yflt<0.0)||(yflt<-xflt*TAN30)))       /* triant A */
        {
        blue_lookup[n]     = 255;
        green_lookup[n]    = (float) 255.0 * ((1.0+yflt) + xflt*TAN30);
        red_lookup[n] = (float) 255.0 * (1.0+(xflt/COS30));
        }
    else
        {
        if ((xflt>0.0)&&((yflt<0.0)||(yflt<xflt*TAN30)))    /* triant C */
            {
            red_lookup[n]   = 255;
            green_lookup[n] = (float) 255.0 * ((1.0+yflt) - xflt*TAN30);
            blue_lookup[n]  = (float) 255.0 * (1.0-(xflt/COS30));
            }
        else                                                /* triant B */
            {
            radius = sqrt (xflt*xflt + yflt*yflt);

            green_lookup[n]  = 255;

            beta = atan (yflt/xflt);
            if (beta < 0.0) beta += PI;

            alpha = beta - PI/6.0;

            if (alpha < PI/2.0)
                blue_lookup[n] = (float) 255.5*((float) 1.0 - ((float)
                        radius*sin(alpha)*tan(PI/6.0) + radius*cos(alpha)));
            else
                blue_lookup[n] = (float) 255.5*((float) 1.0 -
                                 ((float)
                                  radius*sin(PI-alpha)*tan(PI/6.0) -
                                  radius*cos(PI-alpha)));
            alpha = 4.0*PI/6.0 - alpha;

            if (alpha < PI/2.0)
                red_lookup[n] = (float) 255.5*((float) 1.0 - ((float)
                        radius*sin(alpha)*tan(PI/6.0) + radius*cos(alpha)));
            else
                red_lookup[n] = (float) 255.5*((float) 1.0 -
                                ((float)
                                 radius*sin(PI-alpha)*tan(PI/6.0) -
                                 radius*cos(PI-alpha)));
        }   }

    }
}
```

This is provided for information only, and is not considered the best way to generate RGB to CRY mapping, due to the amount of trigonometry. The best technique is to calculate the intensity value, and from this the ideal ROM entries for that colour. This can then be matched to the actual ROM tables to find the nearest match.

Note that the intensity value used here is the largest of red, green and blue. The C procedure below shows this:

```
void disp_lookup (r, g, b)
unsigned char *r;
unsigned char *g;
unsigned char *b;

{
int x,y,n,nearest,ndist,tdist;
unsigned int intens,rt,gt,bt;
long m;

for (y=0; y<240; y++)
    {
    for (x=0; x<256; x++)
        {
        ndist = 1024;
        if (*g > *r) intens = *g;
        else         intens = *r;
        if (*b > intens) intens = *b;

        if (intens > 0)
            {
            rt = (*r << 8) / intens;
            gt = (*g << 8) / intens;
            bt = (*b << 8) / intens;
            }
        else rt=gt=bt=0;

        for (n=0; n<255; n++)
            {
            tdist = abs (red_lookup[n]  -rt)
                  + abs (green_lookup[n]-gt)
                  + abs (blue_lookup[n] -bt);
            if (tdist<ndist)
                {
                ndist=tdist;
                nearest=n;
                if (ndist==0) break;
                }
            }
        rt = (intens *  red_lookup[nearest])>>8;
        bt = (intens * blue_lookup[nearest])>>8;
        gt = (intens *green_lookup[nearest])>>8;
        if (rt<0) rt=0; if (rt>255) rt=255;
        if (gt<0) gt=0; if (gt>255) gt=255;
        if (bt<0) bt=0; if (bt>255) bt=255;
        m = (65536L * bt)
            + (256L * rt)
            +         gt;
        plot (x,y,m);
        r++; g++; b++;
}   }   }
```

5 Graphics Processor Subsystem

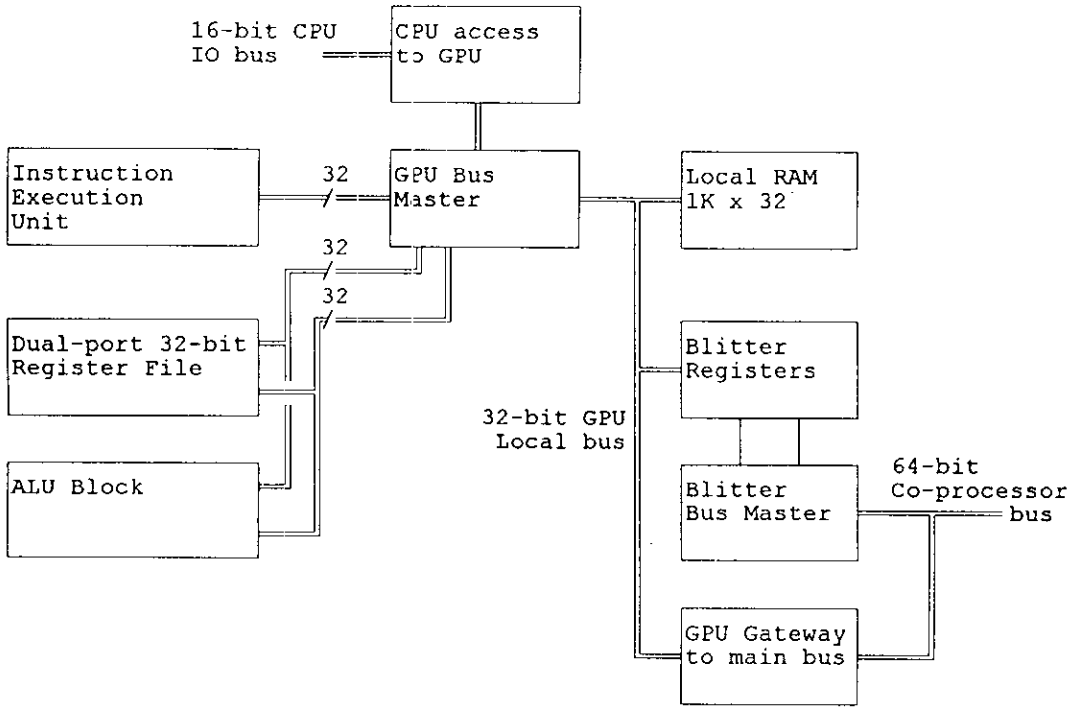
The Graphics Subsystem of Jaguar is a self-contained processing unit, whose view of the external system processor and memory are controlled by a separate memory controller, which is not part the graphics system.

The graphics subsystem transfers data to or from external memory by becoming the master of the co-processor bus. This bus has a 64-bit (phrase) data path, and a 32-bit address, with byte resolution. This bus has multiple masters, and ownership of it is gained by a bus request/acknowledge system, which is prioritised, i.e. ownership can be lost during a request (but not during a memory cycle). The graphics subsystem actually contains two bus masters, the Graphics Processor and the Blitter.

The graphics subsystem also acts as a slave on the IO bus. This bus normally has a 16-bit data path, and allows external processors to access memory and registers within the graphics subsystem. As the data path within the graphics subsystem is 32-bit, all reads and writes must be in pairs.

The memory within the Graphics Subsystem appears to be part of the general machine address space, both to the GPU / Blitter and to external processors. The advantage to the GPU of having local memory is both that it is faster, and that it does not require ownership of the system bus to be accessed.

This diagram shows the architecture and data paths of the graphics subsystem:



JAGUAR

5.1 Memory Map

The Graphics sub-system address space contains the following locations:

```
00402100  GPU_FLAGS RW    GPU flags
00402104  GPU_MTXC  W     GPU matrix control
00402108  GPU_MTXA  W     GPU matrix address
0040210C  GPU_BIGEND      W     GPU big / little endian control
00402110  GPU_PC          RW    GPU program counter
00402114  GPU_CTRL  RW    GPU operation control / status
00402118  GPU_HIDATA      RW    GPU bus interface high long-word
0040211C  GPU_REMAIN      R     GPU division remainder
00402200  BLIT_A1BASE     W     Blitter A1 base
00402204  BLIT_A1FLAGS    W     Blitter A1 flags
00402208  BLIT_A1WIN      W     Blitter A1 window size
0040220C  BLIT_A1PTR      RW    Blitter A1 pointer
00402210  BLIT_A1STEP     W     Blitter A1 step
00402214  BLIT_A1STEPF    W     Blitter A1 step fraction
00402218  BLIT_A1FRAC     RW    Blitter A1 pointer fraction
0040221C  BLIT_A1INC      W     Blitter A1 pointer increment
00402220  BLIT_A1INCF     W     Blitter A1 pointer increment fraction
00402224  BLIT_A2BASE     W     Blitter A2 base
00402228  BLIT_A2FLAGS    W     Blitter A2 flags
0040222C  BLIT_A2MASK     W     Blitter A2 mask
00402230  BLIT_A2PTR      RW    Blitter A2 pointer
00402234  BLIT_A2STEP     W     Blitter A2 step
00402238  BLIT_CMD  W     Blitter command
0040223C  BLIT_COUNT      W     Blitter loop counters
00402240  BLIT_SRCD W     Blitter source data
00402248  BLIT_DSTD W     Blitter destination data
00402250  BLIT_DSTZ W     Blitter destination Z data
00402258  BLIT_SRCZ1      W     Blitter source Z data 1
00402260  BLIT_SRCZ2      W     Blitter source Z data 2
00402268  BLIT_PATD W     Blitter pattern data
00402270  BLIT_IINC W     Blitter intensity increment
00402274  BLIT_ZINC W     Blitter Z increment
00403000  GPU_RAMBASE     W     Local RAM base
```

All these locations may be accessed for read or write as appropriate at the above addresses as 16-bit locations. In addition, for high-speed write operations, they may be written to as 32-bit locations at an offset of 8000 hex from the addresses above. They are not readable at these addresses.

6 Graphics Processor

This section describes the Jaguar Graphics Processor (GPU).

6.1 What is the Graphics Processor?

The Graphics processor (called here the GPU - Graphics Processor Unit) is a simple, very fast, micro-processor; intended for performing the functions associated with generating graphics, such as three-dimensional modelling, shading, fast animation, and unpacking compressed images.

The graphics processor corresponds to the accepted notion of a R.I.S.C. processor (Reduced Instruction Set Computing). This means that:

- most instructions execute in one tick
- all computational instructions involve registers
- memory transfers are performed by load/store instructions
- instructions are of a simple fixed format, with few addressing modes
- there is a wealth of registers, and local high-speed memory

It has several features to give high computational powers, including:

- highly pipe-lined architecture
- 40 MIPs peak throughput
- internal program/data RAM
- register score-boarding
- sixty-four thirty-two bit registers
- ALU includes barrel shifter and parallel multiplier
- systolic matrix multiplication
- fast hardware divide unit
- high-speed interrupt response, including video object interrupts
- close coupling with the blitter

6.2 Programming the Graphics Processor

The GPU is programmed in the same way as any other micro-processor. It has a full instruction set with a broad range of arithmetic instructions, including add, subtract, multiply and divide; boolean instructions, and bit-oriented instructions. It has a range of instructions for loading and storing values in memory, with either register indirect or register indirect plus offset addressing modes. It has jump relative and absolute instructions, both of which may be made dependant on combinations of the zero, carry and negative flags. There are also some more specialist instructions suited to computing matrix multiplies, and some useful aids to floating-point calculations.

The GPU is a full 32-bit processor in that all internal data paths are 32-bits wide, and all arithmetic instructions (except multiply) perform 32-bit computations. The instructions are 16-bits wide.

The GPU has sixty-four internal 32-bit general purpose registers, of which thirty-two are visible at one time. It also has 1K of local high-speed 32-bit RAM, which is where its instructions and working data are normally stored. It also has access to external memory via the 64-bit co-processor bus, and can perform byte, word, long-word and phrase data transfers on this bus. It can also execute its instructions from external RAM.

6.3 Design Philosophy

The GPU is a RISC processor, normally executing one instruction per tick, and therefore capable of very high instruction throughput. The RISC versus CISC debate is a complex one, and will not be discussed here. The RISC approach was chosen for the GPU principally because it occupies less silicon.

The RISC approach leads to a processor design without micro-code, effectively the instruction set is the micro-code, and most instructions execute in one tick. The advantage is that instructions are executed quicker, but the disadvantage is that some operations require more instructions to execute.

The GPU is also intended to perform rapid floating-point arithmetic. However it has no floating-point instructions as such, but has some specific simple instructions that allow a limited precision floating-point library to be capable of in excess of 1 MegaFlop.

The GPU is intended to be programmed in assembly language, and not in a compiled language, as the tasks it is intended to perform are simple repetitive operations, best written in assembly language.

6.4 Pipe-Lining

The GPU design makes extensive use of pipe-lining to improve its throughput. This means that although the GPU can achieve a peak rate of one instruction per tick, each instruction is actually executed over several ticks, but only spends one tick at each pipe-line stage. It is important to understand this as it does have some significant consequences on GPU behaviour.

For a typical instruction, such as ADD, the pipe-line stages are:

1 decode instruction
2 read operands from registers
3 add operands
4 write result back to register

In addition to these stages, a pre-fetch unit attempts to maintain a small queue of unexecuted instructions, to keep the instruction execution unit busy.

Register Score-Boarding

The main side effect of the pipe-lined nature of GPU operation is to do with the interaction of instructions at different stages of the pipe-line, where they affect the same operand, or the same piece of the hardware.

For instance, if the instruction after an ADD was a second ADD of another value to the same register; then if the two instructions were just to follow each other through the pipe-line, then the second ADD would use the old value (the value from before the first ADD). Fortunately, the GPU hardware detects this erroneous condition and suspends execution until the correct value is ready. Clock cycles which occur during these hold-ups are referred to as *wait states*.

Figure 3 shows the data flow associated with the operands of an arithmetic instruction. The thick lines correspond to a pipe-line stage, so that when an instruction is at the **Read operands** stage, the previous instruction is at the **Compute result** stage, and the one before that at the **Write back result** stage.

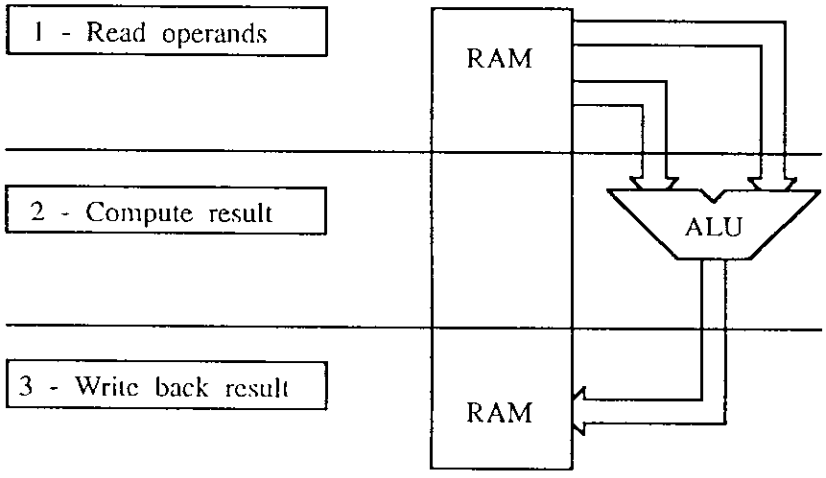


Figure 3 - Instruction data pipe-line

Two problems arise from this architecture:

1. The RAM used within the GPU for its registers has only two data ports, so if the instruction at stage three has to write back to a different register from the two registers being read by the instruction at stage one, then a clash occurs.

2. The instruction at stage one of the pipe-line may need to read a value being computed by the instruction at stage two, but this value will not be available until the instruction at stage two reaches stage three.

To help the programmer avoid a whole class of these problems, the GPU operates what is known as a *score-board*. This tags registers which will alter once some operation has been completed, and will force program flow to wait if an instruction reads a tagged register. This mechanism also applies to the flags, and will wait if:

- an instruction would read a register which is still in the process of being computed by the ALU.
- an instruction would perform a conditional jump, or add/subtract with carry, before the flags have been set as the result of some arithmetic operation.
- an instruction would read a register which is being read from internal memory.
- an instruction would read a register which is the target of a divide operation - as the divide unit is relatively slow, this can cause a significant delay.
- an instruction would read from a register which is waiting to be loaded from slow external memory (which takes a variable amount of time).

Register Write-Back

The score-board unit also controls the writing back of computed values. The registers are a bank of dual-port RAM, so it is not possible to read two register values simultaneously while writing to a third.

If the register to be written back to is being read by the instruction currently at stage 2 of the pipe-line, or if one of the operands of that instruction does not involve a register read, then the write-back will be concealed. Otherwise, the instruction will be held up one cycle while the computed value is written back.

The score-board unit controls all operations which involve writing to registers, and will also generate a wait state if the instruction that would have executed reads two registers, neither of which is the target of the write. Write-back data sources are:

- the result of an ALU computation
- the result of a divide operation (this occurs in parallel with the ALU)
- the data from an internal load operation
- the data from an external load operation

If two of these are to be written back simultaneously, execution is always held up for a tick.

Note that the concealed register write-back mechanism will fail if an instruction reads the same register as both of its operands, therefore programmers should not use the same register as both operands of an instruction if there is any possibility of a concealed write-back occurring

One technique which can be used to help avoid wait states from the score-board unit is to *interleave* two sets of calculations, i.e. ensure that consecutive instructions do not use the same registers, but that instructions two apart generally do.

Jump Instructions

Pipe-lining also affects the execution of jump instructions. The transfer of control does not occur until the instruction *after* the jump instruction has been executed. This can be confusing, but helps to increase the overall instruction throughput. The safest technique is to follow all instruction with a NOP (null operation), but it is quite reasonable to place any other instruction here - but see the notes below on program control flow.

6.5 Memory Interface

The Graphics Processor is intended to operate in parallel with the other processing elements in the Jaguar system. In order to do this, a well-behaved GPU program should only make occasional use of the main memory bus. The GPU therefore has a large amount of local memory, currently four Kilobytes organised as 1K locations of thirty-two bits.

This memory is intended to be used for both program and data. It can be cycled at the graphics processor clock rate, and so is extremely fast. It may be viewed as a simple cache RAM, with software cache control - this technique is known as *visible caching*. When the graphics processor is executing code out of internal RAM, program fetch cycles will occupy less than half the RAM bandwidth.

To the GPU programmer the local RAM, local hardware registers, and external memory all appear in the same address space. The GPU memory controller determines whether a transfer is local or external, and generates the appropriate cycle. The only difference is that only 32-bit transfers are possible within the GPU local address space, whereas 8, 16, 32 or 64-bit transfers are permitted externally.

The local RAM sits on an internal GPU 32-bit bus. Also present on this bus are various GPU control registers, and the blitter control registers. When a GPU transfer occurs outside the local address space, a gateway connects the local bus to the main bus. If a sixty-four bit transfer has been requested, a special register is used for the other half of the data.

The address space is organised as follows (all addresses in bytes):

00000000 - 00401FFF	*external memory*
00402000 - 004021FF	graphics processor registers
00402200 - 004022FF	blitter registers
00402300 - 00402FFF	reserved
00403000 - 00403FFF	local RAM
00404000 - 0040FFFF	reserved
00410000 - FFFFFFFF	*external memory*

This local address space is also available to external devices via the 16-bit I/O bus mechanism (described elsewhere).

The GPU local bus can therefore perform transfers for three quite separate mechanisms. These are, in decreasing order of priority:

- CPU I/O access

- Operand data transfer
- Instruction fetch

The GPU and Data Ordering Conventions

The GPU can operate in both a big-endian and little-endian environment, and as long as the memory interface is programmed to the correct endian mode, and the transfer requested is the width of the operand required, then this operation is largely invisible to the programmer.

The GPU is itself little-endian - this means that the first instruction of the pair in a long-word is the one aligned to bit 0 of the long-word.

6.6 Load and Store Operations

The GPU has a set of load and store instructions, each of which take two register operands. One register is used to provide the address, the other is read to supply data to be stored, or is written with load data.

Load and stores may be performed at byte, word, long-word and phrase width. Bytes and words are aligned with bit 0, and when loaded the rest of the register is set to zero. When phrases are read or written, a register within the GPU local address space should already contain the other long-word for store operations, or is loaded with the other long-word for load operations. Performing phrase loads and stores is the fastest way of transferring blocks.

Load and store operations may also be performed using a simple indexed addressing scheme. This allows either R14 or R15 to be used as a base register, with a five bit unsigned offset encoded into one of the register fields. There is a two tick overhead involved in using these instructions, as the address has to computed.

In local memory, only long-word reads and writes are permitted.

Load and store operations will normally complete in one tick, or two ticks for indexed addresses. The transfer may not be complete at this point, and if another load or store operation occurs before the previous one has completed it will be held up. Load data is written under the control of the score-board unit, which is described elsewhere.

The gateway between the GPU local bus and the external co-processor bus contains a control block for generating external memory transfers. When this block is idle, load and store operations complete as quickly as they would in local memory. For load operations, the data is not loaded into the target register, however, until the external transfer has taken place. The score-board mechanism prevents use of this data before it has been loaded, but other computation may take place. If there is another load or store instruction in the program before the gateway has completed its transfer, then it will be held up until the gateway is idle.

Operand data transfers may occur at two bus priorities in external memory, either at the normal GPU priority, or at the higher DMA priority level. This is controlled by the DMAEN flag. This does not affect program reads, which are always at GPU priority. Bus priority is discussed elsewhere.

6.7 Arithmetic Functions

The GPU contains a powerful ALU section, which as well as the normal arithmetic and boolean functions, all with 32-bit word size, contains a 16 x 16 fast parallel multiplier, and a 32-bit barrel shifter, both of which perform their respective functions in one tick.

The GPU also contains a divide unit. This performs serial division at the rate of two bits per tick, on 32-bit unsigned operands, producing a 32-bit quotient. The operation of this runs in parallel with normal GPU operation.

The ALU has the following set of flags:

Z	zero	set appropriately by all arithmetic operations, normally being set if the result of the operation was zero.
N	negative	set appropriately by all arithmetic operations, normally being set if the result of the operation was negative (bit 31 is a one).
C	carry	set according to carry or borrow out of all add and subtract operations; set with the bit that is shifted out of shift and rotate operations for shift by one; left undefined by other arithmetic operations.

6.8 Interrupts

The GPU can be interrupted by five sources. Interrupts force a call to an address in local RAM, given by sixteen times the interrupt number (in bytes), from the base of RAM. It is the responsibility of the programmer to preserve the registers and flags of the underlying code. Primary register 31 is the interrupt stack pointer. Primary register 30 is corrupted when instruction flow is transferred to the interrupt service routine. Neither register should be used for any other purpose when interrupts are enabled.

Interrupts are allocated as follows:

4 blitter
3 object processor
2 timing generator
1 external, used to allow the GPU to act as a DMA controller
0 CPU interrupt

The flags register contains individual interrupt enables for each of these sources, as well as a master interrupt mask for all interrupts. When the master interrupt mask is set, the primary register bank is selected (see below).

When an interrupt occurs, the master interrupt mask bit is set. The individual enables are not affected, but no other interrupts will be serviced until the mask bit is cleared. The interrupt service routine should clear the master interrupt mask, and the appropriate interrupt latch, and enable higher priority interrupts immediately.

The value pushed onto the R31 stack is the address of the last instruction to be executed before the interrupt occurred. The interrupt service routine should therefore add two to this value before using it to return from the interrupt.

The interrupt latches may be read in the status port, and are cleared by writing a one to their clear bits, writing a zero leaves them unchanged.

The cause of the interrupt may be determined by the location jumped to, but not from the flags register, as more than one interrupt latch bit may be set.

There is a certain degree of interrupt prioritization, in that if two interrupts arrive within a few ticks of each other, the higher numbered will be serviced first. Beyond this, interrupt prioritization is under software control, as described above.

The only operations that are atomic are single instructions, or certain instruction combinations (see below). Interrupts may be disabled by clearing all the enable bits. It is therefore not practical for the interrupt stack to be shared with the underlying code, unless all interrupts are masked across stack operations.

An example interrupt service routine, which does no more than clear the interrupt, is shown below. The interrupt source was interrupt 2.

```
int_serv:
        movei   GPU_FLAGS,r30       ; point R30 at flags register
        load    (r30),r29           ; get flags
        bclr    3,r29               ; clear IMASK
        bset    11,r29              ; and interrupt 2 latch
        load    (r31),r28           ; get last instruction address
        addq    2,r28               ; point at next to be executed
        addq    4,r31               ; updating the stack pointer
        jump    (r28)               ; and return
        store   r29,(r30)           ; restore flags
```

Similar interrupt service routines can handle all the interrupts. Note the following points about this code:

- Registers r28 and r29 may not be used by the under-lying code as they are corrupted, in addition to r30 and r31 which are always for interrupts only.
- Interrupts are re-enabled on the instruction after the jump. If they were enabled any sooner then no other interrupt service routine would be able to use r28 and r29, as they could potentially corrupt them before this service routine had completed,

Atomic Operations

It is necessary for certain operations to be atomic. Three GPU instruction types temporarily lock out interrupts while they complete their operation. These are:

- Immediate data moves, using the MOVEI instruction. Interrupts are locked out while the two words of immediate data are fetched.
- Matrix multiply operations, using the MMULT instruction. Interrupts are locked out until the operation has completed.

- Multiply and accumulate operations, using the IMULTN and IMACN instructions. The result register is not preserved by interrupts, and therefore any multiply/accumulate operation must consist of a sequence of IMULTN and IMACN instructions followed by a RESMAC instruction, with no intervening instructions. The IMULTN and IMACN instructions are always atomic with the succeeding instruction. See the section below on multiply / accumulate instructions.
- Jump instructions are always atomic with the instruction which succeeds them.

6.9 Program Control Flow

Program control normally runs upwards through memory executing instructions sequentially. The GPU can also transfer program flow by performing jump instructions.

Two types of jump are supported, relative and absolute. Jump relative takes a signed five-bit offset, which is treated as an offset in words, and added to the program counter. Jump absolute transfers the contents of a register into the program counter.

Both types of jump may be conditional on the contents of the ALU flags. If the appropriate condition is not met, then the jump instruction is ignored and program flow continues with the next instruction after the jump.

The instruction after a jump is always executed. This is a side-effect of the pre-fetch queue. Programmers may choose either to place a NOP after every jump instruction, or may take advantage of this to place a useful instruction after the jump which will be executed whichever branch is followed.

- Do not place a MOVEI instruction after a jump, as the jump will take effect before the data is fetched, and so will change where the immediate data is fetched from.

- Do not place two jump instructions sequentially, the results are not predictable, and may not be relied on.

The program counter may also be copied into a register.

The GPU can cease operation by clearing the GPUGO bit in the GPU control register (described below). It may then only be restarted by an external write to this register, or by a reset.

Single Step Operation

As an aid to the debugging of GPU programs, the GPU can be set to single step through programs, pausing between instructions until restarted. This operation is controlled by and external CPU as follows:

1- Set up the program counter, then set the GPUGO and SINGLE_STEP control bits in the control register.
2- Poll for the SINGLE_STOP flag in the status register - at this point the first instruction has been executed.

3- Set the SINGLE_GO bit in the control register (keeping GPUGO and SINGLE_STEP set).
4- Poll for the SINGLE_STOP flag, which indicates that the next instruction has been executed.
5- Repeat from step 3.

If the GPU register file is to be read from or written to, then single-stepping will have to be suspended and an appropriate transfer routine run, which will require that the GPUGO bit must be cleared first and the program counter modified. Unfortunately, clearing the GPUGO bit has the effect of altering the value in the program counter, as the pre-fetch queue is discarded. Therefore, after step 4 above, the following operations should be performed:

- read the program counter value
- clear the GPUGO control bit
- read or write to the register file as required
- add two to the program counter value read
- restart from step 1 above

It is necessary to add two to the program counter, as the value read reflects the last instruction executed (or last word of immediate data if it was MOVEI).

6.10 Multiply and Accumulate Instructions

The GPU supports multiply and accumulate (MAC) operations. These involve multiplying two values together, and adding their product to the sum of the products of some previous multiply operations. These are typically used for matrix multiply and digital filtering type applications.

Due to the pipe-lined nature of the design, the multiply and its associated add do not take place in the same cycle. MAC instructions are not therefore like other instructions, in that a special instruction is needed to write back their result.

Take as an example multiplying R8 times R9, R10 times R11, R12 time R13, and placing the sum of their products in R2. All values are signed. The instructions are as follows:

```
imultn      r8,r9       ; compute the first product, into the result
imacn       r10,r11     ; second product, added to first
imacn       r12,r13     ; third product, accumulated in result
resmac      r2          ; sum of products is written to r2
```

MAC instructions may only be followed by further MAC instructions or by the RESMAC instruction. No other combinations are permitted.

6.11 Systolic Matrix Multiplies

The GPU contains a mechanism for performing integer matrix multiplies at a burst rate of the maximum obtainable from the hardware multiplier, which is one multiply per tick. This is generally useful, but has been designed in particular for the matrix multiplies required by the Discrete Cosine Transform algorithm. One technique for this involves performing two 8x8 integer matrix multiplies in succession on a matrix, using the same fixed coefficients, but rotated for the second multiply.

The GPU therefore has a MMULT instruction, which initiates a sequence of between three and fifteen multiply / accumulate instructions, as described above, corresponding to one product term of the result matrix. One of the source matrices is held in the secondary register bank, the other in local RAM. The matrix held in registers is packed, i.e. two elements per register. This allows all of an eight-by-eight matrix to be stored in the secondary register bank.

A matrix multiply is initiated by the MMULT instruction. This takes as its source parameter the register, which is always in the secondary register bank, containing the first two elements of the matrix row. Its destination parameter is the register, in the currently selected register bank, in which to write the result.

The matrix held in RAM may be accessed in either increasing row or increasing column order, in other words the data for each successive multiply operation are either one location or the matrix width apart.

Like interrupts, the systolic operation is performed by forcing internally generated instructions into the instruction stream. The first instruction is IMULTN, the middle ones IMACN, and the last RESMAC. These have their operands modified in the manner described above.

The MMULT instruction should not be preceded by a LOAD or STORE instruction.

6.12 Register File

The GPU contains a register file of sixty-four thirty-two bit registers. All of them may be used as general purpose registers, although some are also assigned special functions.

All instructions contain two five-bit register operand fields, although they are not always used as such. Where an instruction references a register, this five-bit field is turned into the register address. There are two banks of these 32-bit registers, primary and secondary. The primary register bank, bank 0, is always used for interrupt service.

Bank select bits are provided in the flags register, and special MOVE instructions allow data to be moved between banks.

6.13 External CPU Access

The GPU internal address space is accessible to an external bus master at any time - external access having the highest priority on the GPU local bus. This means that the blitter may be used to load data into the local RAM.

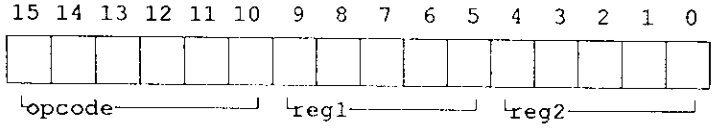
The local address space is accessible for read or write at the addresses given elsewhere in this document, and these locations are presented as sixteen bit memory, which must always be accessed in the order low address then high address.

To allow faster transfers into the GPU space, all the registers are also available as thirty-two bit memory, at an offset of 8000 hex from their normal addresses. At this address, the internal memory is write only.

If the blitter is being used to write into the GPU space, then phrase wide transfers may be performed, as the bus control mechanism will automatically divide these up to suit the width of the memory being addressed.

6.14 Instruction Set

The GPU instructions are all sixteen bits, made up as follows:



- opcode defines the instruction to be executed
- reg2 is the destination register, or the only register of single register instructions
- reg1 is the source register

The reg2 and reg1 fields can have other meanings with some instructions. When both operands are registers, the general rule is that they should not be the same register. This is discussed above under register write-back.

The instruction set is as follows, where the syntax is

<Opcode name> <source>,<destination>

Code	Syntax	Description
0	ADD Rn,Rn	32-bit two's complement integer add, result is destination register contents added to the source register contents, and is written to the destination register. Z set if the result is zero N set if the result is negative C represents carry out of the adder

1	ADDC Rn,Rn	32-bit two's complement integer add with carry in according to the previous state of the carry flag, otherwise like ADD. Z set if the result is zero N set if the result is negative C represents carry out of the adder
2	ADDQ n,Rn	32-bit two's complement integer add, where the source field is immediate data in the range 1-32, otherwise like ADD. Z set if the result is zero N set if the result is negative C represents carry out of the adder
3	ADDQT n,Rn	32-bit two's complement integer add, like ADDQ except that it is transparent to the flags, which retain their previous values. ZNC unaffected
4	SUB Rn,Rn	32-bit two's complement integer subtract, result is the source register contents subtracted from the destination register contents, and is written to the destination register. The carry flag represents borrow out of the subtracter, and the zero flag is set if the result is zero. Z set if the result is zero N set if the result is negative C represents borrow out of the subtracter
5	SUBC Rn,Rn	32-bit two's complement integer subtract with borrow in according to the carry flag, otherwise like SUB. Z set if the result is zero N set if the result is negative C represents borrow out of the subtracter
6	SUBQ n,Rn	32-bit two's complement integer subtract, where the source field is immediate data in the range 1-32, otherwise like SUB. Z set if the result is zero N set if the result is negative C represents borrow out of the subtracter
7	SUBQT n,Rn	32-bit two's complement integer subtract, like SUBQ except that it is transparent to the flags, which retain their previous values. ZNC unaffected
8	NEG Rn	32-bit two's complement negate, the result is the destination register contents subtracted from zero, and is written back to the destination register. Z set if the result is zero N set if the result is negative C represents borrow out of the subtracter

9	AND Rn,Rn	32-bit logical AND, the result is the boolean AND of the source register contents and the destination register contents, and is written back to the destination register. Z set if the result is zero N set if the result is negative C undefined
10	OR Rn,Rn	32-bit logical OR, the result is the boolean OR of the source register contents and the destination register contents, and is written back to the destination register. Z set if the result is zero N set if the result is negative C undefined
11	XOR Rn,Rn	32-bit logical XOR, the result is the boolean XOR of the source register contents and the destination register contents, and is written back to the destination register. Z set if the result is zero N set if the result is negative C undefined
12	NOT Rn	32-bit logical invert, the result is the boolean XOR of FFFFFFFF hex and the destination register contents, and is written back to the destination register. Z set if the result is zero N set if the result is negative C undefined
13	BTST n,Rn	Test the bit in the destination register selected by the immediate data in the source field, which is in the range 0-31. Z set if the selected bit is zero N set if bit 31 is one C undefined
14	BSET n,Rn	Set the bit in the destination register selected by the immediate data in the source field, which is in the range 0-31. The other bits of the destination register are unaffected. Z set if the result is zero N set if the result is negative C not defined
15	BCLR n,Rn	Clear the bit in the destination register selected by the immediate data in the source field, which is in the range 0-31. The other bits of the destination register are unaffected. Z set if destination register is now all zero N set from bit 31 of the result C not defined

16	MULT Rn,Rn	16-bit unsigned integer multiply, the 32-bit result is the unsigned integer product of the bottom 16-bits of each of the source and destination registers, and is written back to the destination register. Z set if the result is zero N set if bit 31 of the result is one C undefined
17	IMULT Rn,Rn	16-bit signed integer multiply, the 32-bit result is the signed integer product of the bottom 16-bits of each of the source and destination registers, and is written back to the destination register. Z set if the result is zero N set if the result is negative C undefined
18	IMULTN Rn,Rn	Like IMULT, but result is not written back to destination register. Intended to be used as the first of a multiply/accumulate group, as there are potential speed advantages in not writing back the result. Z set if the result is zero N set if the result is negative C undefined
19	RESMAC Rn	Takes the current contents of the result register and writes them to the register indicated. Intended to be used as the final instruction of a multiply/accumulate group. * refer to the section on Multiply and Accumulate instructions ZNC unaffected
20	IMACN Rn,Rn	16-bit signed integer multiply and accumulate, like IMULT, except that the 32-bit product is added to the result of the previous arithmetic operation, and the result is not written back to the destination register. Intended to be used after IMULTN to give a multiply/accumulate group. * refer to the section on Multiply and Accumulate instructions ZNC unaffected
21	DIV Rn,Rn	32-bit dividend, 32-bit divisor, unsigned division, yielding 32-bit quotient, like normal microprocessor division. Refer to the section on arithmetic functions. ZNC unaffected
22	ABS Rn	32-bit integer absolute value. Has the same effect as NEG if the operand is negative, otherwise does nothing. Z set if the result is zero N cleared C set if the operand was negative

23	SH Rn,Rn	32-bit shift left or right given by the value in the source register. A positive value causes a shift to the right. Values of plus or minus thirty-two or greater give zero. Zero is shifted in. Z set if the result is zero N set if the result is negative C represents bit 0 of the un-shifted data for right shift, or bit 31 for left shift
24	SHLQ n,Rn	32-bit shift left by n positions, in the range 1-32. Otherwise like SH. (The shift value is actually encoded as 32-n, this is handled by the assembler). Z set if the result is zero N set if the result is negative C represents bit 31 of the un-shifted data
25	SHRQ n,Rn	As SHLQ but shift right, zero shifted in. Z set if the result is zero N set if the result is negative C represents bit 0 of the un-shifted data
26	SHA Rn,Rn	As SH but right shift is arithmetic, i.e. sign shifted in. Z set if the result is zero N set if the result is negative C represents bit 0 of the un-shifted data for right shift, or bit 31 for left shift
27	SHARQ n,Rn	As SHRQ but arithmetic shift right, i.e. sign shifted in. Best mnemonic. Z set if the result is zero N set if the result is negative C represents bit 0 of the un-shifted data
28	ROR Rn,Rn	32-bit rotate right by the bottom 5 bits of the source register. Can be used for ROL functions by complementing the value. Z set if the result is zero N set if the result is negative C represents bit 31 of the un-shifted data
29	RORQ n,Rn	Immediate data version of ROR. Z set if the result is zero N set if the result is negative C represents bit 31 of the un-shifted data
30	CMP Rn,Rn	32-bit compare, this is the same as SUB without the result being stored, but the flags reflect the result of the comparison, which may therefore be used for equality testing and magnitude comparison. Z set if the result is zero (operands equal) N set if the result is negative (source greater than destination operand) C represents borrow out of the subtracter

31	CMPQ n,Rn	32-bit compare with immediate data in the range -16 to +15. Z set if the result is zero (operands equal) N set if the result is negative (immediate data greater than destination operand) C represents borrow out of the subtracter
32	SAT8 Rn	Saturate the 32-bit signed integer operand value to an 8-bit unsigned integer. If it is negative it is set to zero, if it is greater than 255 it is set to 255. This is useful for computed intensities and so on, to counteract the effect of rounding errors. Z set if the result is zero N cleared C undefined
33	SAT16 Rn	Saturate the 32-bit signed integer operand value to an 16-bit unsigned integer. If it is negative it is set to zero, if it is greater than 65535 it is set to 65535. This is useful for computed Z, audio values, and so on, to counteract the effect of rounding errors. Z set if the result is zero N cleared C undefined
34	MOVE Rn,Rn	32-bit register to register transfer. ZNC unaffected
35	MOVEQ n,Rn	32-bit register load with immediate value in the range 0-31. ZNC unaffected
36	MOVETA Rn,Rn	32-bit register to alternate register transfer, the destination register lying in the other bank of 32 registers. ZNC unaffected
37	MOVEFA Rn,Rn	32-bit alternate register to register transfer, the source register lying in the other bank of 32 registers. ZNC unaffected
38	MOVEI n,Rn	32-bit register load with next 32-bits of instruction stream. The first word in the instruction stream is the low word, the second the high word. ZNC unaffected
39	LOADB (Rn),Rn	8-bit memory read. The source register contains a 32-bit byte address. The destination register will have the byte loaded into bits 0-7, the remainder of the register is set to zero. This applies to external memory only, internal memory will perform a 32-bit read. ZNC unaffected

40	LOADW (Rn),Rn	16-bit memory read. The source register contains a 32-bit byte address, which must be word aligned. The destination register will have the word loaded into bits 0-15, the remainder of the register is set to zero. This applies to external memory only, internal memory will perform a 32-bit read. ZNC unaffected
41	LOAD (Rn),Rn	32-bit memory read. The source register contains a 32-bit byte address, which must be long-word aligned. The destination register will have the data loaded into it. ZNC unaffected
42	LOADP (Rn),Rn	64-bit memory read. The source register contains a 32-bit byte address, which must be phrase aligned. The destination register will have the low long-word loaded into it, the high long-word is available in the high-half register. This applies to external memory only, internal memory will perform a 32-bit read. ZNC unaffected
43 44	LOAD (R14+n),Rn LOAD (R15+n),Rn	32-bit memory read, as LOAD, except that the address is given by the sum of either R14 or R15 and the immediate data in the source register field, in the range 1-32. The offset is in long words, not in bytes, therefore a divide by four should be used on any label arithmetic to give the offset. This is slower than normal LOAD operations due to the two-tick overhead of computing the address. ZNC unaffected
45	STOREB Rn,(Rn)	8-bit memory write. The source register contains a 32-bit byte address. The destination register has the byte to be written in bits 0-7. This applies to external memory only, internal memory will perform a 32-bit write. ZNC unaffected
46	STOREW Rn,(Rn)	16-bit memory write. The source register contains a 32-bit byte address, which must be word aligned. The destination register has the word to be written in bits 0-15. This applies to external memory only, internal memory will perform a 32-bit write. ZNC unaffected
47	STORE Rn,(Rn)	32-bit memory write. The source register contains a 32-bit byte address, which must be long-word aligned. The destination register contains the data to be written. ZNC unaffected
48	STOREP Rn,(Rn)	64-bit memory write. The source register contains a 32-bit byte address, which must be phrase aligned. The destination register contains the low long-word of the data to be written, the high long-word is obtained from the high-half register. This applies to external memory only, internal memory will perform a 32-bit write. ZNC unaffected

49 50	STORE Rn,(R14+n) STORE Rn,(R15+n)	32-bit memory write, write as STORE, with address generation in the same manner as the equivalent LOAD instructions. ZNC unaffected
51	MOVE PC,Rn	Load the destination register with the address of the current instruction. The actual value read from the PC is modified to take into account the effects of pipe-lining and prefetch, to give the correct address. This is the only way for the GPU to read its own PC. ZNC unaffected
52	JUMP cc,(Rn)	Jump to location pointed to by the source register, destination field is the condition code, where the bits encode as follows: Bit Condition 0 zero flag must be clear for jump to occur 1 zero flag must be set for jump to occur 2 flag selected by bit 4 must be clear for jump to occur 3 flag selected by bit 4 must be set for jump to occur 4 if set select negative flag, if clear select carry. If more than one condition is set, then they must all be true for the jump to occur (the conditions are ANDed). ZNC unaffected
53	JR cc,n	Relative jump to the location given by the sum of the current address and the immediate data in the source field, which is signed and therefore in the range +15 or -16 words. The condition codes encode in the same way as JUMP. ZNC unaffected
54	MMULT Rn,Rn	Start systolic matrix element multiply, the source register is the location of the register source matrix, the product is written into the destination register. Refer to the section on matrix multiplies. The flags reflect the final multiply/accumulate operation: Z set if the result is zero N set if the result is negative C represents carry out of the adder
55	MTOI Rn,Rn	Extract the mantissa and sign from the IEEE 32-bit floating-point number in the source register, and create a signed integer in the destination. The most significant bit is bit 23, but the value is sign extended. Z set if the result is zero N set if the result is negative C undefined

JAGUAR

56	NORMI Rn,Rn	Gives the 'normalisation integer' for the value in the source register, which should be an unsigned integer. The normalisation integer is the amount by which the source should be shifted right to normalise it (the value can be negative), and is also the amount to be added to the exponent to account for the normalisation. Z set if the result is zero N set if the result is negative C undefined
57	NOP	Do nothing. ZNC unaffected

6.15 Internal Registers

This section describes the internal registers of the Graphics processor. Note that some of these are read or write only.

All GPU registers are 32-bit, and will require all 32 bits to be written.

Flags Register

00402100 GPUFLAGS Read/Write

This register provides status and control bit for several important GPU functions. Control bits are:

0	ZERO_FLAG	The ALU zero flag, set if the result of the last arithmetic operation was zero. Certain arithmetic instructions do not affect the flags, see above.
1	CARRY_FLAG	The ALU carry flag, set or cleared by carry/borrow out of the adder/subtracter, but undefined after other arithmetic operations.
2	NEGA_FLAG	The ALU negative flag, set if the result of the last arithmetic operation was negative.
3	IMASK	Interrupt mask, set by the interrupt control logic at the start of the service routine, and is cleared by the interrupt service routine writing a 0. Writing a 1 to this location has no effect.
4-8	INT_ENA0-4	Interrupt enable bits for interrupts 0-4. The status of these bits is overridden by IMASK.
9-13	INT_CLR0-4	Interrupt latch clear bits. These bits are used to clear the interrupt latches, which may be read from the status register. Writing a zero to any of these bits leaves it unchanged, and the read value is always zero.
14	REGPAGE	Switches from register bank 0 to register bank 1. This function is overridden by the IMASK flag, which forces register bank 0 to be used.
15	DMAEN	When DMAEN is set, GPU LOAD and STORE instructions perform external memory transfers at DMA priority, rather than GPU priority. This has no effect on program data fetches, which continue at GPU priority.

WARNING - writing a value to the flag bits and making use of those flag bits in the following instruction will not work properly due to pipe-lining effects. If it is necessary to use flags set by a STORE instruction, then ensure that at least one other instruction lies between the STORE and the flags dependent instruction.

Matrix Control Register

00402104 MTXCTRL Write only

This register controls the function of the MMULT instruction. Control bits are:

0-3	MWIDTH	Matrix width, in the range 3 to 15
4	MADDW	When set, this control bit make the matrix held in memory be accessed down one column, as opposed to along one row.

Matrix Address Register

00402108 MTXADDR Write only

This register determines where, in local RAM, the matrix held in memory is.

2-11	MTXADDR	Matrix address.

Data Organisation Register

0040210C DATAORG Write only

This register controls the physical layout of pixel data and CPU I/O registers. If its current contents are unknown, the same data should be written to both the low and high 16-bits.

0	BIG_IO	When this bit is set, 32-bit registers in the CPU I/O space are big-endian, i.e. the more significant 16-bits appear at the lower address.
1	BIG_PIX	When this bit is set the pixel organisation is big-endian. See the discussion elsewhere in this document.

GPU Program Counter

00402110 GPU_PC Read/Write

The GPU program counter may be written whenever the GPU is idle (GPUGO is clear). This is normally used by the CPU to govern where program execution will start when the GPUGO bit is set.

The GPU program counter may be read at any time, and will give the address of the instruction currently being executed. If the GPU reads it, this must be performed by the MOVE PC,Rn instruction, and not by performing a load from it.

The GPU program counter must always be written to before setting the GPUGO control bit. When the GPUGO bit is cleared, the program counter value will be corrupted, as at this point the pre-fetch queue is discarded.

GPU Control/Status Register

00402114 GPUCTRL Read/Write

This register governs the interface between the CPU and the GPU.

0	GPUGO	This bit stops and starts the GPU. The CPU or GPU may write to this register at any time. The status of this bit after a system reset may be externally configured.
1	CPUINT	Writing a 1 to this bit allows the GPU to interrupt the CPU. There is no need for any acknowledge, and no need to clear the bit to zero. Writing a zero has no effect. A value of zero is always read.
2	GPUINT1	Writing a 1 to this bit causes a GPU interrupt type 1. There is no need for any acknowledge, and no need to clear the bit to zero. Writing a zero has no effect. A value of zero is always read.
3	SINGLE_STEP	When this bit is set GPU single-stepping is enabled. This means that program execution will pause after each instruction, until a SINGLE_GO command is issued. The read status of this flag indicates whether the GPU has actually stopped, and should be polled before issuing a further single step command.
4	SINGLE_GO	Writing a one to this bit advances program execution by one instruction when execution is paused in single-step mode. Neither writing to this bit at any other time, nor writing a zero, will have any effect. Zero is always read.
5	DMA_ACK	Setting this bit sets the external DMA acknowledge line. The hardware does not generate a pulse, so software should write a 1 then a 0, or *vice versa*, as required. The actual pin of the Jaguar device is the inverse of this bit.
6-10	INT_LAT0-4	Interrupt latches. The status of these bits indicate which interrupt request latch is currently active, and the appropriate bit should be cleared by the interrupt service routine, using the INT_CLR bits in the flags register. Writing to these bits has no effect.
11	BUS_HOG	When the GPU is executing code out of external RAM it will normally give up the bus between program fetches. This behaviour should allow the CPU to continue to run at the same time. Setting this bit causes the GPU to attempt to hold on to the bus between program fetches, which improves its execution speed, at the expense of any lower priority device using the bus.
12-15	VERSION	These bits allow the GPU version code to be read. Currently this has the value 1. Future variants of the GPU may contain additional features or enhancements, and this value allows software to remain compatible with all versions. It is intended that future versions will be a superset of this GPU.

High Data Register

00402118 HIGHDATA Read/Write

This 32-bit register provides the high part of GPU phrase reads and writes. It is physically a single register, and therefore a phrase read followed by a phrase write will write back the same high data unless this register is modified.

7 Blitter

This section describes the Jaguar Blitter.

7.1 What is the Blitter?

Blitter is an abbreviation for *bit block processor*. It purpose is to process, by filling or copying, blocks of bits or pixels. These blocks may be one contiguous piece, or they may be sub-blocks (such as rectangles) within a larger pixel array.

The blitter may also be seen as a hardware engine designed for painting and moving pixels as quickly as possible - it performs a variety of graphics operations at a rate limited largely by the memory access speed. It is used as an aid to the GPU, allowing a GPU program to process high-level graphics operations, whilst the blitter, in parallel, performs the low-level repetitive pixel-by-pixel operations.

For example, the GPU might calculate the co-ordinates and gradients associated with a polygon, while the blitter draws the strips of pixels. Alternatively, the GPU might be processing text with attributes, and computing font addresses and window positions, while the blitter paints the characters.

The blitter can perform a variety of operations on blocks of memory, including:

- simple memory copies
- copies and fills of rectangles within windows
- line-drawing
- image rotation and scaling
- single-scans of polygons fills
- Gouraud shading
- Z-buffering.

The blitter can operate on 1, 2, 4, 8, 16 or 32 bit packed pixels, with considerable flexibility with regard to the memory layout.

The *tour de force* of the blitter is its ability to generate Gouraud shaded polygons, using Z-buffering, in sixteen bit pixel mode. A lot of the logic in the blitter is devoted to its ability to create these pixels four at a time, and to write them at a rate limited only by the bus bandwidth, using the GPU to calculate the Z and intensity gradients and start and stop pixels on a line-by-line basis. This will give the system the ability to generate realistic animated 3D graphics.

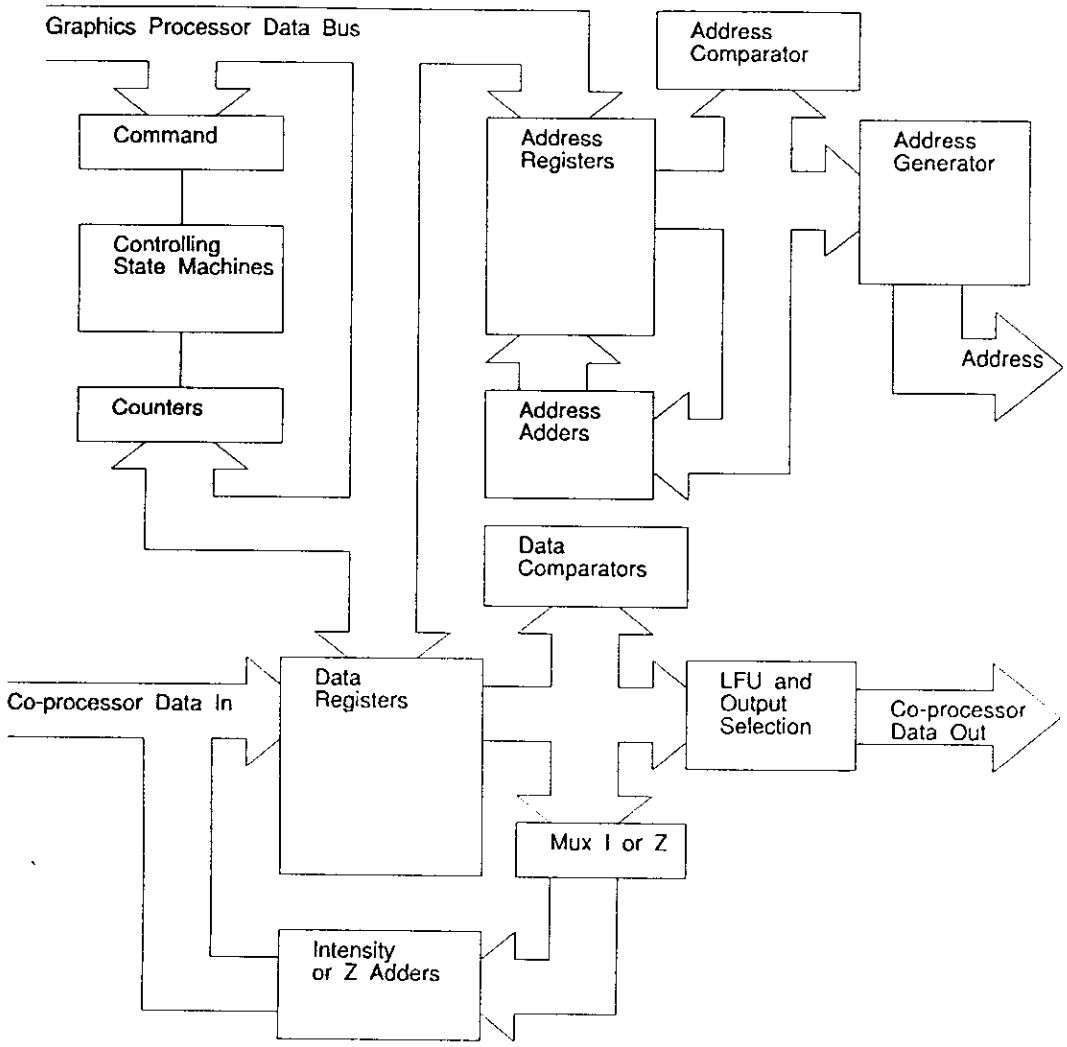


Figure 4 - *Blitter Architecture*

7.2 Programming the Blitter

The blitter is programmed by setting up a description of the required operation in its registers. These are accessible in the system memory map, and so may be set by the GPU or by an external processor.

The registers control the three functional blocks that make up the blitter, the address generator, data path, and control logic. Each of these is described in the sections that follow.

The descriptions that follow give a fairly dry account of how the blitter works. These are useful for reference, but for an introduction to how to use the blitter use the examples further on.

The blitter architecture is summarised in Figure 4.

7.3 Address Generation

The address generator generates an address within a window of pixels. A window is a packed array of pixels in memory, and may well be the data associated with an object processor object. A window is described by its base address and width. A pointer into this window is set up for the blitter start position, and is programmed in terms of its X and Y address. The ability to program the address generator in pixel address terms considerably simplifies the task of preparing blitter commands.

In addition to these registers, various other registers contain specific values to allow considerable flexibility in how the pointers are modified during blitter operations.

The blitter has two address generation units, used for the *source* and *destination* addresses of copy operations, etc. The two address generators are called A1 and A2. A1 is normally the destination address register and A2 the source, although these roles may be reversed. A1 is more sophisticated in its address generation capabilities than A2.

The address register block looks like this:

00402200	A1 base address
00402204	A1 control flags
00402208	A1 window size
0040220C	A1 pointer
00402210	A1 step integer part
00402214	A1 step fractional part
00402218	A1 pointer fractional part
0040221C	A1 increment integer part
00402220	A1 increment fractional part
00402224	A2 base address
00402228	A2 control flags
0040222C	A2 window mask
00402230	A2 pointer
00402234	A2 step integer part

Windows

All notions of address within the blitter correspond with the concept of a window. A window is a rectangle of pixels, stored in memory as a linear array of packed phrases. A window is described by a base register, and has a width and height, both in pixels. A set of flags describe the size of those pixels, their physical layout in memory, and various aspects of how the pointer is updated.

The address itself is generated from a window pointer. This has an X and Y value, and again is in pixels. The pointer may point to areas outside the window, and A1 supports hardware clipping of addresses outside the window.

Address Generation

The X and Y pointers are sixteen bit values. However, the address generation mechanism will only generate valid addresses for Y values in the range 0-4095, i.e. it treats Y values as 12-bit unsigned values. The higher order bits of Y are ignored. X is treated as an unsigned 16-bit value.

The address generator derives the window width from a very simple six-bit floating-point format. The width value has a four bit unsigned exponent, and a three bit mantissa, whose top bit is implicit, and which has the point after the implicit top bit. This is similar to a cut

down version of the IEEE single precision format without the sign bit. It must give a whole number of phrases in the current pixel size. Valid exponent values are in the range 0-11.

For example, a window width of 640 is 1010000000 binary, i.e. 1.01 x 2^9. Therefore the mantissa takes the value 01 (implicit top bit), and the exponent 1001. The width is therefore 1001 01 in binary.

Note that there is a window bounds clipping mechanism for the A1 pointer, which treats the X and Y as signed sixteen bit values. This is described elsewhere.

Pointer Updating

Both blitter address generators can update their pointers so that they describe a raster scan over a rectangle. Along a scan line, the pointer may be updated either by one pixel or to the next phrase boundary, depending on how the blitter is currently operating. Refer to the Data Path section for further details.

At the end of a scan line, the pointer is updated by a step value, which is the distance in X and Y to the start of the next scan line. This action of scan across the block, then step to the next start, is controlled by the blitter's inner and outer control loops, the inner loop traversing a scan line, and the outer loop adding the step value. Thus the inner loop length is the block width, and the outer loop length the block height.

In addition to these modes, both address registers have certain special modes.

A2 may have a boolean mask applied to its pointer. This is logically ANDed with the pointer, so that the pointers may not exceed the bounds of a rectangle, whose sides are a power of two pixels long. This is intended to repeat a source texture or pattern over a larger destination area, e.g. filling a wall with a repeated brick pattern

A1 supports address updates based on a Digital Differential Analyzer. This technique produces successive address by adding an increment to the pointers, both of which have integer and fractional parts, and is used in particular for line-drawing and rotating images.

The pointer and increment of A1, in both X and Y, have sixteen bit integer parts and sixteen bit fractional parts. The step value used on the outer loop address update also has integer and fractional parts.

7.4 Data Path

The blitter has a sixty-four bit data path, with a variety of registers. It can be used to process entire phrases at once, or one pixel at a time. Pixels may the one, two, four, eight, sixteen or thirty-two bits wide, and are always stored in a packed manner.

Data registers are:

00402240	Source data, or computed intensity fractional parts
00402248	Destination data
00402250	Destination Z
00402258	Source Z1, or computed Z integer parts
00402260	Source Z2, or computed Z fractional parts
00402268	Pattern data, or computed intensity integer parts
00402270	Intensity increment
00402274	Z increment

When writing or copying pixels, arbitrary alignment of the source and destination data is allowed, and the blitter aligns the source to match the destination data when required. When transferring phrases the source and destination address pointers do not need to be aligned to the same point in a phrase, the blitter will automatically align the source to the destination, but only for pixels of eight bits or larger.

There are therefore two source data registers, to provide current source and previous source for alignment. There is also a destination data register, which can be logically combined with the source, and is also used to restore the destination data area when only parts of it are updated.

There is a parallel mechanism for Z data, used for Z-buffering. This allows the depth of the data about to be written to be compared with the depth of the data already present on the screen, and the write of the new data inhibited if the data already present has a higher priority. This applies to sixteen bit pixel mode only.

There are therefore two source Z registers and a destination Z register.

- source Z
- computed Z

The GOURZ flag selects computed Z data.

Overriding both these selections is a mechanism to write back unchanged destination data. If a mode is enabled where data may be inhibited, e.g. bit-to-byte expansion, or Z buffering, then a pre-read of the destination data should be performed. This also applies to pixel sizes of less than eight bits.

Data Comparators

There are three data comparators available within the blitter. These are:

- The bit comparator. This is used for bit to pixel expansion, and selects a bit or group of bits from the source data register, using a counter which is cleared every time the inner loop is entered. The bit is then used to control whether a pixel is written at the current location.

- The Z comparator. This is used in 16-bit pixel mode to compare the 16-bit un-signed integer Z attribute of a pixel on the screen with that about to be written, and to prevent the write operation if the pixel on the screen has a higher priority.

- The data comparator. This is used to provide a means to make block copies with transparent colours, and to help with flood fill by performing searches. It compares pixel values in either 8 or 16-bit pixel modes.

The comparators may be used to achieve three effects:

- When painting pixels one at a time a comparator output can be used to inhibit the write of a pixel, leaving the previous value unchanged.

- When painting pixels a phrase at a time, the comparator outputs can force destination data to be written back. If this has been previously read then the data will be left unchanged, if not then a background colour can be used, stored in the destination data register

- The action of the blitter can be stopped altogether. This may be used for collision detection, searching, etc.

Note that the bit comparator can only produce a mask to operate over an entire phrase in 8-bit pixel mode.

7.5 Bus Interface

The blitter accesses memory through the 64-bit co-processor bus, and takes full advantage of the width and high-speed of this bus. The blitter will normally cycle this bus at a rate limited only by the speed of the external memory, although there is a one-tick overhead when turning round from a read to a write transfer.

All external memory is viewed by the blitter as being phrase wide - if the physical layout is narrower then the memory controller expands the transfer into the appropriate number of transfers.

The blitter requests the bus at the start of an operation, and will not stop requesting it until the entire operation is complete. As described elsewhere, higher priority bus masters can request and be granted the bus during a blitter operation, and this will suspend blitter operation until the higher priority operation has released the bus.

7.6 Register Description

The following is a list of all the externally accessible locations within the blitter. The data registers may only be written to while the blitter is idle.

Address Registers

All address registers are 32-bits unless otherwise indicated. The addresses given are byte offsets from the base of the GPU area.

A1 Base Register

00402200 Write only

32-bit register containing a pointer to the base of the window pointer to by A1. This address must be phrase aligned.

A1 Flags Register

00402204 Write only

A set of flags controlling various aspects of the A1 window and how addresses are updated.

Bits	Name	Description
0-1	Pitch	The distance between successive phrases of pixel data in the window data structure. Gaps may be used to provide alternate pixel maps for double-buffering, for Z data, and for other control information. The distance between two successive phrases of pixels is given by two to the power of this value; i.e. a pitch of 0 means pixel data phrases are contiguous, 1 means 1 phrase gaps, 2 means 3 phrase gaps, and 3 means 7 phrase gaps.
2	unused	
3-5	Pixel size	The pixel size, where the actual pixel size is 2^n, n is the value stored here. Values 0-5 are allowed.
6-8	Z offset	This value gives the offset from a phrase of pixel data of its corresponding Z data in phrases. Values of 0 and 7 are not used.
9-14	Width	This width is distinct from the width in pixels stored in the window register, and is the width used for address generation. The width is a six-bit floating point value in pixels, with a four bit unsigned exponent, and a three bit mantissa, whose top bit is implicit, and which has the point after the implicit top bit. This is similar to the IEEE single precision format without the sign bit. It must give a whole number of phrases in the current pixel size.

16-17	X add ctrl.	These control the update of the X pointer on each pass round the inner loop. Values are: 00 Add phrase width and truncate to phrase boundary (sets phrase mode). 01 Add pixel size, effectively add one, 10 Add zero 11 Add the increment
18	Y add ctrl.	This bit controls how the Y pointer is updated within the inner loop. It is overridden by the X control bits if they are in add increment mode. 0 Add zero 1 Add one
19	X sign	This bit may be set in conjunction with the X add pixel size mode to make the operation subtract pixel size. It should not be set with other modes.
20	Y sign	Makes the Y add one mode into Y subtract one.

A1 Window Size

00402208 Write only

This register contains the size in pixels, and is used for optionally clipping writes into the A1 window, so that if the pointer leaves the window bounds no write is performed. The width is an unsigned fifteen bit value in the low word, the height an unsigned fifteen bit value in the high word. The top bits of each word are ignored.

The window origin (0,0) is always at the top left hand corner of the screen, and so clipping is performed when the pointer values are negative, or when the pointer values are greater than or equal to these values.

A1 Window Pointer

0040220C Read/Write

This register contains the X (low word) and Y (high word) pointers onto the window, and are the location where the next pixel will be written. They are sixteen-bit signed values.

A1 Step Value

00402210 Write only

The step register contains two signed sixteen bit values, which are the X step (low word) and Y step (high word). These may be added to the X and Y pointer on each pass round the outer loop, between passes through the inner loop.

When calculating the step value for phrase-mode blits, note that the X pointer will be left pointing at the start of the first phrase not written by the blit.

A1 Step Fraction Value

00402214 Write only

The step fraction register may be added to the fractional parts of the A1 pointer in the same manner as the step value. This is used when A1 is being used to scan over the source of a rotated image.

A1 Window Pointer Fraction

00402218 Read/Write

This register contains the fractional parts of the pointer when A1 is being used to implement a D.D.A. based address generator, for line-drawing, etc. The X part is in the low word, and the Y part in the high word.

A1 Pointer Increment

0040221C Write only

The increment is added to the pointer value within the inner loop when the address update is in add increment mode. This register contains the integer parts of the increment, and the X part is in the low word, and the Y part in the high word.

A1 Pointer Increment Fraction

00402220 Write only

This is the fractional parts of the increment described above.

A2 Base Register

00402224 Write only

32-bit register containing a pointer to the base of the window pointer to by A2. This address must be phrase aligned.

A2 Flags Register

00402228 Write only

A set of flags controlling various aspects of the A2 window and how addresses are updated.

Bits	Name	Description
0-1	Pitch	As A1.
2	unused	

3-5	Pixel size	As A1.
6-8	Z offset	As A1.
9-14	Width	As A1.
15	Mask	Enables boolean AND masking of the A2 pointer by its window register.
16-17	X add ctrl.	These control the update of the X pointer on each pass round the inner loop. Values are: 00 Add phrase width (truncate to phrase boundary) 01 Add pixel size (effectively add one) 10 Add zero
18	Y add ctrl.	This bit controls how the Y pointer is updated within the inner loop. 0 Add zero 1 Add one
19	X sign	This bit may be set in conjunction with the X add pixel size mode to make the operation subtract pixel size. It should not be set with other modes.
20	Y sign	Makes the Y add one mode into Y subtract one.

A2 Window Mask

0040222C Write only

This register is used as the window size only in the sense that it may be used to AND mask the pointer register when the Mask flag is set.

A2 Window Pointer

00402230 Read/Write

This register contains the X (low word) and Y (high word) pointers onto the window, and are the location where the next pixel will be written. They are sixteen-bit signed values.

A2 Step Value

00402234 Write only

The step register contains two signed sixteen bit values, which are the X step (low word) and Y step (high word). These may be added to the X and Y pointer on each pass round the outer loop, between passes through the inner loop.

When calculating the step value for phrase-mode blits, note that the X pointer will be left pointing at the start of the first phrase not written by the blit.

Control Registers

Command Register

00402238 Write only

This register describes the operation of the blitter. A write to this register initiates blitter operation, so it should be written to last when setting up a blitter command. Control bits are:

Bit	Name	Description
Bits 0-5 enable corresponding memory cycles within the inner loop. Destination write cycles are always performed (subject to comparator control), but all other cycle types are optional.		
0	SRCEN	Enables a source data read as part of the inner loop operation.
1	SRCENZ	Enables a source Z read as part of the inner loop operation. This bit is ignored unless SRCEN is set.
2	SRCENX	Enables an "extra" source data read at the start of an inner loop operation. This is necessary where data has to be re-aligned, and may also sometimes be of use in bit-to-pixel expansion. If SRCENZ is set an extra Z read is also performed.
3	DSTEN	Enables a destination data read as part of inner loop operation. This must always be performed for pixels smaller than 8 bits, where part of the destination data write will need to restore the data that was previously there.
4	DSTENZ	Enables a destination Z read as part of inner loop operation.
5	DSTWRZ	Enables a destination Z write as part of inner loop operation.
6	DISO_A1	Enables clipping when the A1 pointer lies outside its window boundaries. This has the effect of inhibiting destination writes within the inner loop, but blitter operation will continue.
7	NOGO	Diagnostic use only, prevents write to the command register starting the blitter. Set to zero.
Bits 8-10 enable address updates within the outer loop. These should only be enabled when required as there is a one-tick overhead per update.		
8	UPDA1F	Add the fractional part of the A1 step value to the fractional part of the A1 pointer between inner loop operations in the outer loop.
9	UPDA1	Add the A1 step value to the A1 pointer between inner loop operations in the outer loop.
10	UPDA2	Add the A2 step value to the A2 pointer between inner loop operations in the outer loop.
11	DSTA2	Reverses the normal roles of the address registers from A1 as destination and A2 as source to A2 as destination and A1 as source.
12	GOURD	Enable Gouraud shaded data updates within inner loop, i.e. the intensity gradient fractional part, repeated four times, is added to the computed intensity fraction register (a.k.a. destination data), then the intensity gradient integer part is

		added with the carry from the previous add to the computed intensity value register (a.k.a. pattern data).
13	GOURZ	Enable polygon Z data updates within the inner loop, i.e. add Z fractions to the Z fraction register (source Z 2), then add with carry the Z integer part to the Z integers (source Z 1).
14	TOPBEN	Enable carry into the top byte of the intensity integers in Gouraud data updates (leave clear for CRY mode).
15	TOPNEN	Enable carry into the top nibble of the intensity integers in Gouraud data updates (leave clear for CRY mode).
Bits 16-17 select alternative write data - the default source is the Logic Function Unit, whose output is controlled by the LFUFUNC bits.		
16	PATDSEL	Select pattern data as the write data.
17	ADDDSEL	Diagnostic purposes only.
18-20	ZMODE	These bits give the conditions under which the Z comparator generates an inhibit. Setting them all to zero disables the Z comparator. bit 0 source less than destination bit 1 source equal to destination bit 2 source greater than destination
21-24	LFUFUNC	The bits control the data produced by the logic function unit. The output is the boolean OR of the following minterms: bit 0 NOT source AND NOT destination bit 1 NOT source AND destination bit 2 source AND NOT destination bit 3 source AND destination
25	CMPDST	Make the pixel value comparator compare destination data with pattern data rather than source data with pattern data.
26	BCOMPEN	Enable write inhibit on the output from the bit comparator. This works pixel by pixel in any size, but over whole phrases only on 8 bit pixels.
27	DCOMPEN	Enable write inhibit on the output from the data comparator.
28	BKGWREN	When a write inhibit occurs, this flag enables the blitter to still perform the write, but to write back destination data. This only applies to pixel mode, in phrase mode destination data is always written.

Status Register

00402238 Read only

This register is present largely for diagnostic purposes when chip testing. The outer loop IDLE bit may be polled to determine whether the blitter has completed.

0	NOWRITE
1	STOPPED
2	inner IDLE
3	inner SREADX
4	inner SZREADX

5	inner SREAD
6	inner SZREAD
7	inner DREAD
8	inner DZREAD
9	inner DWRITE
10	inner DZWRITE
11	outer IDLE
12	outer INNER
13	outer A1FUPDATE
14	outer A1UPDATE
15	outer A2UPDATE
16-31	inner count

Counters Register

0040223C Write only

The low word is the number of iterations of the inner loop operation. This is a sixteen bit value which reloads the inner loop counter on each entry to the inner loop.

The high word is the number of iterations of the outer loop. This is a sixteen bit value which is loaded directly into the outer loop counter.

Data Registers

All data registers are sixty-four bits, unless otherwise noted.

Source Data Register

00402240 Write only

The source data may be pre-loaded with data for bit-to-byte expansion. The source data register also serves to hold the four sixteen bit fractional parts of intensity when computing Gouraud shaded intensity.

Destination Data Register

00402248 Write only

Destination Z Register

00402250 Write only

Source Z Register 1

00402258 Write only

The source Z register 1 is also used to hold the four integer parts of computed Z.

Source Z Register 2

00402260 Write only

The source Z register 2 is also used to hold the four fraction parts of computed Z.

Pattern Data Register

00402268 Write only

The pattern data register also serves to hold the computed intensity integer parts and their associated colours.

Intensity Increment

00402270 Write only

This thirty-two bit register holds the integer and fractional parts of the intensity increment used for Gouraud shading.

Z Increment

00402274 Write only

This thirty-two bit register holds the integer and fractional parts of the Z increment used for computed Z polygon drawing.

7.7 Modes of Operation

This section discusses some of the typical modes of operation of the Blitter. It is by no means a complete guide to all possible modes, but will show how to do certain common operations. This is the best way to learn how to use the blitter.

Throughout this section, flags in flags registers that are not mentioned should always be set to zero. Registers that are not mentioned need not be set up.

Block Moves

The simplest of all blitter operations is a block move, copying one area of memory onto another. The blitter will perform this operation one phrase at a time, and it is therefore a very rapid way of transferring data.

The source address of the data should be stored in the A2 base register, and the destination address in the A1 base register. If these are not phrase aligned addresses then they should be rounded down to a phrase boundary, and the offset (in the pixel size set) from the phrase boundary written into the X pointer. The Y pointer should be set to zero.

The length of the block should be stored in the inner counter - the number represents the number of pixels, so the largest block that can be copied is 65536 pixels, where 32-bit pixels are set this is 256K. For smaller blocks it is usually easier to work in bytes. The outer counter should be set to one.

The blitter needs to be told how to update the pointers after each read and write cycle, so the add control bits are set to zero to indicate phrase mode in both address flags registers.

Having set these, a command is stored in the command register, with the SRCEN bit set to enable source reads, and the LFUFUNC bits set to 1100 to select source data.

Example

Consider copying 66 bytes from address 02003457 hex to address 0200789A. The following values, in hex, would be written into the blitter command registers.

```
BLIT_A1BASE     02003450    Rounded down to phrase boundary
BLIT_A1FLAGS    00000018    Pixel size is 3, phrase mode
BLIT_A1PTR      00000007    Offset from phrase base
BLIT_A2BASE     02007898    Rounded down to phrase boundary
BLIT_A1FLAGS    00000018    Pixel size is 3, phrase mode
BLIT_A1PTR      00000002    Offset from phrase base
BLIT_COUNT      00010042    Length in X, 1 in Y
BLIT_CMD        01800001    SRCEN, and LFUFUNC is 1100
```

Rectangle Moves

Rectangle moves are very like block moves, but use a two-dimensional data set rather than the one-dimension of a block operation. This brings in various new concepts.

A two-dimensional array of pixels is stored in memory as a linear array of phrases. This will usually be the data field of a bit-mapped object. The blitter has to know the width of this *window* of pixels. As an address in the window, in pixel terms, is given by the X pointer plus the width times the Y pointer; a multiply operation is necessary to compute the address. To avoid the need for a hardware multiplier in the blitter address generator, the width is rather strangely encoded.

Blitter window width is expressed as a floating-point number. The actual value has a four-bit exponent and a three-bit mantissa, whose top bit is implicit. This allows blitter window widths to be any value whose binary form has no more than three significant digits followed by some number of zeroes.

As an example, here are how various window widths encode:

Value	Binary	Floating -point	Encoded
20	10100	1.01 x 2^4	0100 01
80	1010000	1.01 x 2^6	0110 01
128	10000000	1.00 x 2^7	0111 00
640	1010000000	1.01 x 2^9	1001 01
3584	111000000000	1.11 x 2^11	1011 11

The largest width value allowed is the last value one in this table - the smallest width is one phrase in the current pixel size. The width must always be a whole number of phrases in the current pixel size.

Rectangles are blitted like a raster scan, i.e. a line of pixels is transferred, then the pointer advances one line and transfers the next scan line of the rectangle. This jump from the end of one line to the start of the next is given by the *step* value. If pixels are being transferred one at a time, then the step value for X is the window width minus the rectangle width. If pixels are being transferred one phrase at a time, then the X pointer is left pointing at the start of the next phrase **after** the end of the block, and so the step value should be reduced accordingly.

Example

Copy a block of pixels from a linear array at 0603AA68 to a 30 x 30 pixel rectangle at offset (315, 17) in a window 320 pixels wide and 200 high at 070C0000 of 16-bit pixels.

This will also introduce the idea of clipping. Clipping may be performed by the A1 address generator, and simply prevents writes occurring at addresses outside the window boundaries, i.e. X or Y either negative or grater than the window size. The window size is programmed in the A1 window size registers. This is not much faster than writing the clipped pixels, so if

a large number of pixels are to be clipped then it is worth performing the clipping at a higher level.

Set up A1 to point at the target window, thus:

```
BLIT_A1BASE     070C0000        Window base address
BLIT_A1FLAGS    00004220        Pixel size = 4
                                Width = 320 = 1.01 x 2^8 = 1000 01
                                X add control = 0 for phrases
BLIT_A1WIN      00C80140        Window size for clipping
BLIT_A1PTR      0011013B        Start position
BLIT_A1STEP     0001FFDF        Y step is 1, X step is given by 315 - end position.  X pointer
                                ends at 315 + 30, rounded up to the next phrase boundary, i.e. 345
                                rounded to 348, so X step is -33.
```

Set up A2 to point at the source data - as this is a linear array only X is incremented:

```
BLIT_A2BASE     0603AA68
BLIT_A2FLAGS    00000020        Pixel size = 4
                                X add control = 0 for phrases
BLIT_A2PTR      00000000
```

Then the command and counter

```
BLIT_COUNT      001E001E        Rectangle size
BLIT_CMD        01800241        Set SRCEN
                                Set DISO_A1 to enable clipping
                                Set UPDA1 to add step value
                                Set LFUFUNC to 1100
```

Note that this makes two assumptions about the linear array being used for source data; firstly that it is phrase aligned, and secondly that it is smaller than 64K pixels. If it was larger, then due to X pointer size limitations, it would have to be set up as a window, and would be subject to the window restrictions.

Character Painting

Character painting is a particular example of a class of operations requiring *bit to pixel expansion.* As well as character painting, this may include such things as background patterns, simple texture fills, etc.

When bit to pixel expansion is being performed, the source data is used as a bit mask. Bits are extracted from the source data and if they are set then the corresponding pixel is painted in the currently selected output data form, if the bit is clear then either the pixel is left unchanged, or a background colour is written.

This allows character painting to paint the characters only, leaving the background unchanged (if the destination data is read), or with another colour written to the 'paper' areas (pre-loaded into the destination data register which is not read in the inner loop).

Character painting can be performed one pixel at a time in all screen modes, and can also be performed one phrase at a time in eight and sixteen bit per pixel modes.

The bit selection counter is reset every time the inner loop is left, so bit packed data patterns may be up to 64 pixels wide.

Example

Consider painting an 8x8 character on an 8-bit per pixel window.

Image Rotation

The blitter can rotate and scale images as a single operation.

Consider taking a rectangular image and rotating it into a window.

- The bounding rectangle of the rotated image is calculated in the destination window.
- This rectangle is then transformed into the source image co-ordinate system.
- A2 is used as the destination address register and performs a raster scan over the bounding rectangle, pixel-by-pixel. The width and height of the blit are given by the size of this bounding rectangle.
- A1 performs a scan over the source image, with the increment integer and fraction set up to describe a scan over the first line of the translated bounding rectangle. The step and fraction parts then translate it to the start of the next scan.
- Clipping is generated when A1 is outside the bounds of the source image, so that writes at A2 will only be enables when A1 lies within the bounds of the source image, clipping the rotated form correctly.

!! picture here

Consider as an example, a 12 pixel square image starting at (10,10) in a window. We would like to rotate this image clockwise by 30 degrees, make it larger by a factor of 1.3, and move it across by 30 pixels.

The image is stored in some RAM at 02000000h, in a window 112 pixels wide, and the object mode is 8 bits per pixel.

First it is necessary to transpose the square's co-ordinates into the target co-ordinate system. The basic program below shows how to do this:

```
100 deg30 = .523598775
110 PRINT "Co-ordinates? "
120 INPUT xi, yi
130 x = xi - 16
140 y = yi - 16
150 xs = (x * COS(deg30)) - (y * SIN(deg30))
160 ys = (x * SIN(deg30)) + (y * COS(deg30))
170 x = xs * 1.3
180 y = ys * 1.3
190 x = x + 46
200 y = y + 16
210 PRINT "Translated: ", INT(x + .5), INT(y + .5)
```

This translates the vertices of the square as follows:

```
(10,10) -> (43,5)
(21,10) -> (56,12)
(21,21) -> (48,25)
(10,21) -> (36,18)
```

The bounding box is therefore from X = 36 to 56, and Y = 5 to 25. The vertices of this are then translated back to the source co-ordinate system, as shown by another basic program:

```
100 degm30 = -.523598775
110 PRINT "Co-ordinates? "
120 INPUT xi, yi
130 x = xi - 46
140 y = yi - 16
150 x = x / 1.3
160 y = y / 1.3
170 xs = (x * COS(degm30)) - (y * SIN(degm30))
180 ys = (x * SIN(degm30)) + (y * COS(degm30))
190 x = xs + 16
200 y = ys + 16
210 PRINT "Reverse translated: ", INT(x + .5), INT(y + .5)
```

This translates the vertices of the bounding box as follows:

```
(36,5)  -> (5,13)
(56,5)  -> ((18,5)
(56,25) -> (26,18)
(36,25) -> (13,26)
```

We then set up A1 as the *source* address register, making its window base the top left hand corner of the source image, and its window size the image size. The A1 pointer will traverse the translated bounding box.

!!! more to come

Gouraud Shading and Z-Buffering

Gouraud shading is a simple technique for modelling lit curved surfaces, which are represented by a series of polygons. To make the surface appear curved, the intensity must vary smoothly, rather than being uniform over each polygon. Gouraud shading approximates to the appearance of the curved surface by computing the intensity at each vertex, using a vertex normal, and some suitable illumination model. The vertex intensity is then linearly interpolated across the polygon edges, and the edge intensities are linearly interpolated across the polygon scan lines.

Gouraud shading is only an approximation to the appearance of the curved surface, and may appear unnatural where there are large intensity changes across single polygons. However, it is much more attractive than not graduating the shading at all. Better shading can be achieved with Phong shading, where the normals are interpolated, but this is much more computationally intensive, and is not feasible within the blitter.

Z-buffering involves attaching a Z value attribute to each pixel, which corresponds to how far away it is from the observer. When pixels are drawn on the screen, their Z values can be compared with the Z of the pixels already there, and the existing data preserved if closer to the observer. Z-buffering therefore provides a simple means of achieving hidden surface removal.

The blitter can perform Gouraud shading and Z-buffering in sixteen bit pixel mode only. Each blit creates one scan line of a polygon, with the graphics processor responsible for re-calculating the start, length and gradient parameters for each scan line. Four pixels and their associated Z values can be calculated as fast as the memory interface can write them out, so the bus rate is always the limiting factor.

To calculate the Z and intensity values, the blitter contains registers which represent the Z and intensity with a sixteen bit integer and sixteen bit fractional part. The intensity integer also contains the colour value, so intensity is prevented from overflowing into the colour information. The TOPBEN and TOPNEN bits enable this overflow, if desired.

There are four of these thirty-two bit values for intensity, and four for Z, so that four pixels may be calculated in parallel. There are also thirty-two bit Z and intensity increment registers, which give the amount added to each pixel for each write.

At each pass round the inner loop; the sixteen-bit fractional part of the intensity increment is added to the fractional parts of the intensity values, held in the source data register. Then the eight-bit integer part of the intensity is added with carry out of the fractional add to the integer pixel values in the pattern data register. Carry is prevented from propagating from intensity to colour. A similar mechanism governs Z.

Both the intensity and the Z values *saturate*. This means that if they reach their lowest or highest values they are clipped there, rather than wrapping round. For example, adding one to a Z value of FFFF hex will give FFFF, not the overflow result 0000.

To take an example, consider blitting an 18 pixel strip of Gouraud shaded Z-buffered pixels. The blitter command registers would be programmed as follows (all other registers need not be written).

Address registers are set up as follows:

```
A1_BASE         0x01600000      The window base address

A1_PITCH        1               Pixel data and Z data alternate
A1_PSIZE        4               16-bit pixels
A1_ZOFFS        1               Z data is one phrase up from pixel data
A1_WIDTH        0x11            20-pixel window: 1.01 x 2^4 - 0100 01
A1_ADDC         0               Add one phrase to address

A1_WIN_X        20              Window width
A1_WIN_Y        5               Window height

A1_PTR_X        1               First pixel at address 0,1
A1_PTR_Y        0
```

Data registers are set up assuming the first pixel has an intensity of C7.2833, and a colour of 00. The intensity gradient is minus 15.9265. The values for the first four pixels have to be set up (the left-most is actually off the edge of the strip, so the intensity gradient is subtracted from it). Similarly, the Z of the first pixel is E7E7.E000, and the Z gradient is minus 1818.1FFF.

```
Pattern         00DC00C700B1009C        Intensity integer parts and colour data
Source          FEDCEAC7D6B1C29C        Intensity fractions
Source Z1       FFFFE7E7CFCFB7B7        Z integer parts
Source Z2       FFFFE000C001A002        Z fractional parts
I Inc           FFA9B66C                Intensity increment (four times minus 15.9265)
Z Inc           9F9F8004                Z increment (four times minus 1818.FFFF)
```

Control information is set up as follows:

```
Inner count     18              Strip width
Outer count     1               Single pixel high strip

DSTEN           1               Read destination data,to restore if necessary
DSTENZ  1                       Read destination Z, to compare with computed Z
DSTWRZ  1                       Write destination Z, restoring or replacing
DISO_A1         1               Clip within window
GOURD           1               Gouraud data computation enabled
GOURZ           1               Z buffer data computation enabled
PATDSEL         1               Write pattern data
ZMODE           3               Overwrite existing data if the new Z value is greater than or
                                equal to the existing Z value
```

The numbers here are pretty arbitrary, but they show the general idea.

8 Appendices

8.1 Data Organisation - Big and Little Endian

The Jaguar system is intended to be useable in either a little-endian, e.g. Intel 80x86, or big-endian, e.g. 680x0, environment. The difference between these two systems is to do with the way in which bytes of a larger operand are stored in memory.

When storing a long-word in memory, a big-endian processor considers that the most significant byte is stored at byte address 0, while a little-endian processor considers that the most significant byte is stored at byte address 3. When both 32-bit processors are fitted with 32-bit memory this is not an issue for the memory interface, as the concept of byte address has no meaning; where it does become a problem is when the data path width is narrower than the operand width.

This document adopts the big-endian convention and Motorola operand ordering convention. Little-endian and Intel operand conventions could equally well have been applied.

IO Bus Interface

The IO Bus Interface is a 16-bit interface. Therefore, 32-bit data such as addresses will be presented differently between the little-endian and big-endian systems. What happens, in effect, is that the sense of A1 is inverted between the two systems. A big-endian system will see the high word of long-word at the low address, a little-endian system will see the high word at the high address.

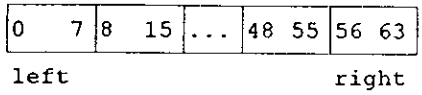
Co-Processor Bus Interface

As the co-processor bus interface is 64-bits wide, there is no problem regarding big and little endian systems, although graphics processor programmers should always use byte, word, or long-word transfers as appropriate to the operand size to avoid having to be aware of whether the CPU is big or little endian.

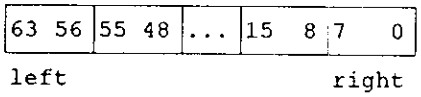
Pixel Organisation

One side effect of the big or little endian philosophies is with regard to the organisation of pixels within a phrase.

In the little-endian system, the left-most pixel is always the least significant. In a phrase of data the left-most pixel includes bit 0. In byte address terms, this is in byte 0.



In the big-endian system, the left-most pixel is always the most significant. The left-most pixel therefore always includes bit 63. In byte address terms this is stored in byte 0.



Consider an eight-bit per pixel mode:

- in pixel mode, the left-most pixel in both systems is at byte address 0.
- in phrase mode, the little-endian left hand pixel is on bits 0-7, the big-endian left hand pixel is on bits 56-63.

(these modes refer to blitter operation, which is described elsewhere)

This difference therefore affects operations that involve addressing pixels within a phrase when transferring a whole phrase at once (blitter phrase mode).