Software Reference Manual

Tom & Jerry

Version 10.0

Stephen Moss

20/12/2021

This document is an amended and updated version of the original Atari documentation, Copyright Atari Corporation 1995.

Contents

Introduction 6
What is the Jaguar? 6
How is the Jaguar used? 7
Jaguar Video and Object Processor (Tom) 8
Overview 8
Object Processor Performance 9
Memory Controller 9
Microprocessor Interface 10
Memory Map 11
Internal Memory Map 12
Peripheral Memory Map 20
Object Definitions 20
Description of the Object Processor/Pixel Path 26
Refresh Mechanism 29
Colour Mapping 30
Introduction 30
The CRY Colour Scheme 30
Gouraud Shading Requirements 30
Colour Space 31
Physical Requirements 31
CRY Colour Scheme 32
RGB to CRY conversion 32
Physical Implementation 33
Graphic Processor Subsystem 34
Memory Map 35
Graphics Processor 37
What is the Graphics Processor? 37
Programming the Graphics Processor 37
Design Philosophy 38
Pipe-Lining 38
Register Score-Boarding 39
Register Write-Back 40
Jump Instructions 42

 © *Stephen Moss*

Memory Interface 42
External View of GPU Space 43
The GPU and Data Ordering Conventions 43
Load and Store Operations 43
Arithmetic Functions 44
Interrupts 45
Atomic Operations 46
Program Control Flow 46
Single Step Operation 47
Illegal Instruction Combinations 48
Conditional Jumps 48
Multiply and Accumulate Instructions 49
Systolic Matrix Multiples 49
Divide Unit 50
Register File 50
External CPU Access 51
Pack and Unpack 51
Internal Registers 52
Blitter 56
What is the Blitter? 56
Programming the Blitter 56
Address Generation 57
Windows 58
Address Generation 58
Pointer Updating 59
Data Path 59
Write Data 60
Data Comparators 61
Bus interface 61
Register Description 61
Address Registers 62
Control Registers 65
Data registers 68
Modes of Operation 70

Block Moves ... 70
Rectangle Moves ... 71
Character Painting ... 71
Image Rotation ... 72
Gouraud Shading and Z-Buffering ... 73
Jaguar Digital Sound Processor (Jerry) ... 75
Frequency dividers ... 75
Programmable Timers ... 77
Jerry Interrupts ... 77
Synchronous Serial Interface ... 78
Asynchronous Serial interface (ComLynx and MIDI) ... 79
Joystick Interface ... 82
General Purpose IO Decodes ... 82
DSP ... 82
Introduction ... 82
Programming the DSP ... 83
Design Philosophy ... 83
Pipe-Lining ... 83
Memory Map ... 83
Wave Table ROM ... 83
Load and Store Operations ... 84
Arithmetic Functions ... 84
Interrupts ... 84
Program Control Flow ... 85
Circular Buffer Management ... 85
Extended Precision Multiply / Accumulates ... 85
Divide Unit ... 85
Register File ... 85
External CPU Access ... 85
Internal Registers ... 86
Appendices ... 89
RISC Instruction Set ... 89
Flags ... 90
Register Usage ... 90

Writing Fast GPU and DSP Programs 104
Data Organisation – Big and Little Endian 106
IO Bus Interface 106
Co-Processor Bus Interface 106
Pixel Organisation 106

Introduction

This document is the Jaguar Software Reference Manual – it is a definitive reference work for the programmers view of the Jaguar ASICs. It is neither a hardware reference work nor a guide to a particular implementation of the Jaguar design.

What is the Jaguar?

The Jaguar is a custom chip set 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 applies to work-station and business uses.

As well as a general purpose CPU, the Jaguar contains four 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 generate 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 which includes 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.

----- **Digital Sound Processor**

The Digital Sound Processor is similar to the Graphics Processor, but is intended primarily for synthesising sound, and for playing back sampled sound. It may also be used for general processing tasks.

The Jaguar provides these blocks with a 64-bit data path to external memory devices, and is capable of a very high data transfer rate into external dynamic RAM.

 © *Stephen Moss*

How is the Jaguar used?

The Jaguar contains two custom chips, code-named Tom and Jerry.

For graphics, Tom contains the Object Processor, the Blitter and the Graphics Processor. For sound, Jerry holds the Digital Sound Processor (DSP). In addition to these, there is an external CPU, currently a 68000. When animating graphics there are therefore four processing elements, and they have all got specific roles to play.

The CPU is used as a manager. It deals with communications with the outside world, and manages the system for the other processors. It is the highest level in the control flow of a Jaguar program, and has complete control of the system.

The Object Processor is at the other end of the chain for generating graphics. It reads the *object list*, and on the basis of the commands there assembles each display line of the video picture. Objects are usually areas of pixels, and these may overlap and may be easily moved from frame to frame. The order in which they are processed in the object list determines how they overlap. Objects can also modify what is already in the display line being assembled, and can scale bit-maps. They may contain transparent pixels.

The object processor performs all the functions of a traditional *sprite engine*, while also offering all the flexibility of a pixel-map based system. It is capable of a range of animation effects, and is a powerful graphics tool its own right.

The Graphics Processor and Blitter provide a tightly-coupled pair of processors for performing a much wider range of animation effects. A design goal of the system was to provide a fast throughput when rendering 3D polygons. The Graphics Processor therefore has a fast instructions throughput, and a powerful ALU with a parallel multiplier, a barrel-shifter, and a divide unit, in addition to the normal arithmetic functions.

The graphics processor has 4 kilobytes of fast internal RAM, which is used for local program and data space. This allows it to execute a program in parallel with the other processing units.

The Blitter is capable of performing a range of blitting operations 64 bits at a time, allowing fast block move and fill operations; it can generate strips of pixels for Gouraud shaded Z-buffered polygons 64 bits at a time. It is also capable of rotating bit-maps, line-drawing, character-painting, and a range of other effects.

The graphics processor and the Blitter will usually act together preparing bit-maps in memory, which are then displayed by the object processor.

The Digital Signal Processor has eight kilobytes of fast internal RAM, which is used for local program and data space. It is tightly-coupled to Jerry’s internal timers, interrupts and audio output to allow fast, independent access.

Jaguar Video and Object Processor (Tom)

Overview

The Jaguar video section has been designed to drive a PAL/NTSC TV. 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 unforeseen) products.

Two colour resolutions are supported, 24-bit and 16-bit. 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, and in the case of CRY (Cyan, Red and Intensity), is simpler to shade and is almost indistinguishable from 24-bit mode.

The 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. The line buffer contains physical pixels these may be either 16 or 24-bit pixels. The line buffers may be swapped over at the start and in the middle of display lines.

In 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.

The video timing is completely programmable in units of the video clock.

The Jaguar uses an Object Processor, this combines the advantages of frame store and sprite based architectures. The Jaguar’s Object Processor is simple yet sophisticated. It has scaled and unscaled bit-map objects, branch objects for controlling its control flow, and interrupt objects. It can interrupt the Graphics Processor to perform more complex operations on its behalf. The Graphics Processor will support perspective, rotation, branches, pallet loads, etc.

The Object Processor can write into the line buffer at up to two pixels per clock cycle. 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.

The 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 flash lamp.

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.

 © *Stephen Moss*

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 phrases fetched.

The 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 one write per system clock tick. 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 twice the processor clock rate.

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 two clock ticks. These are page mode cycles. When the DRAM row addresses must change there is an overhead of between three and seven clock cycles (depending on DRAM speed). These RAS cycles will occur infrequently during object data fetches data 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.

The GPU and Blitter may not be used in high bus priority while the object processor is running. The DMEAN but of G_FLAGS should be 0, and the BUSHI bit of B_CMD should be 0.

No bus master may operate at a higher priority than the object processor. If something else gets the bus between the second and third phrases of an object header, then the line buffer address can be corrupted, causing horizontal black stripes and possibly other artefacts in the display.

Memory Controller

The Jaguar’s memory controller is very fast and flexible. It hides the memory width, speed and type from the other parts of the system.

Memory is grouped into banks that 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 looks 64 bits wide.

There are eight writes strobes – one for each 8 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 or EPROM is 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 and pull-down resistors determine the ROM width during reset.

DRAM is the principal memory type, as it is cheap and fast when used in fast page mode. In fast page mode the DRAM cycles at two ticks per transfer. 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 organized to minimize the number of page changes. The page size is 2 Kbytes.

There are four memory banks; two of ROM and two of DRAM.

Microprocessor Interface

The Jaguar has been designed to work with any 16 or 32-bit microprocessor with (up to) 24 address lines. The interface is based on the 68000 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 I/O 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.

Jerry uses the same microprocessor interface.

The CPU normally has the lowest bus priority but under interrupt its priority is increased.

The following list gives the priorities of all bus masters.

Highest priority

1. Higher priority daisy-chained bus master
2. Refresh
3. DSP at DMA priority
4. GPU at DMA priority
5. Blitter at high priority
6. Object Processor
7. DSP at normal priority
8. CPU under interrupt
9. GPU at normal priority
10. Blitter at normal priority
11. CPU

Lowest priority

Memory Map

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

After a reset the following 2 Mbyte window, corresponding to the ROM0 area, is repeated throughout the 16 Mbyte address space until the memory is configured by the microprocessor by writing to MEMCON1. (This allows the system to boot whether the microprocessor is a 680x0, an 80x68, or a Transputer.) After configuration, this map corresponds to the area defined as ROM0 on the maps below.

1FFFFF	Bootstrap ROM
120000	
118000	Jerry DSP
114000	Joysticks and GPIO0-5
110000	Jerry
100000	Internal Registers
000000	Bootstrap ROM

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

FFFFFF	ROM0 Bootstrap ROM and registers	2 Mbytes
E00000		
800000	ROM1 Cartridge ROM	6 Mbytes
400000	DRAM1 Dynamic RAM	4 Mbytes
000000	DRAM0 Dynamic RAM	4 Mbytes

ROMHI=1

FFFFFF	DRAM 0 Dynamic RAM	4 Mbytes
C00000		
800000	DRAM1 Dynamic RAM	4 Mbytes
200000	ROM 1 Cartridge ROM	6 Mbytes
000000	ROM0 Bootstrap ROM and registers	2 Mbytes

ROMHI=0

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

A 68000 system will naturally operate with RAM at 0, so the ROMHI = 1 map is assumed throughout this document. If the system is operated with ROMHI = 0 then the first digit of all internal addresses should be 1 rather than F.

Internal Memory Map

Internal memory is mostly 16 bits wide to allow operation with 16-bit microprocessors.

32-bit write cycles are allowed to some areas of internal memory notably the line buffer and graphics processor memory. The line buffer supports 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 Memory Configuration Register One *F00000* *RW*

Do NOT Modify: For information only

Bits	Name	Description
0	ROMHI	When set the two ROM decodes address the top 8Mb within the 16Mb window. When clear the ROM decodes address the bottom 8Mb. This document assumes throughout that ROMHI is set when discussing register addresses.
1-2	ROMWIDTH	Specifies the width of ROM: 0 8 bits 1 16 bits 2 32 bits 3 64 bits
3-4	ROMSPEED	Specifies ROM cycle time: 0 10 clock cycles 1 8 clock cycles 2 6 clock cycles 3 5 clock cycles
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

These times are clock cycles.

Bits	Name	Description
7	FASTROM	Sets the ROM cycle time to two clock cycles. This is for test purposes only.
8-10	Unused	Set to zero
11-12	IOSPEED	Specifies the speed of external peripherals. The number of cycles here is the overall cycle time, the control strobes are active for two cycles less than this. 0 18 clock cycles 1 10 clock cycles 2 4 clock cycles 3 6 clock cycles
13	Unused	Set to zero.
14	CPU32	Indicates that the microprocessor is 32 bits.
15	Unused	Set to zero.

All the ROMSPEED bits are set to zero on reset. ROMHI, ROMWIDTH and CPU32 are determined by external pull-up / pull-down resistors. All the other bits are undefined. ROM0 repeats every 2 Mbytes

 © *Stephen Moss*

until this register is written to.

MEMCON2 Memory Configuration Register Two *F00002* *RW*
Do NOT Modify: For information only

Bits	Name	Description
0-1	COLS0	Specifies the number of columns in DRAM0 0 256 1 512 2 1024 3 2048
2-3	DWIDTH0	Specifies the width of DRAM0 0 8 bits 1 16 bits 2 32 bits 3 64 bits
4-5	COLS1	Specifies the number of columns in DRAM1 0 256 1 512 2 1024 3 2048
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 rate of 64 KHz. Refresh cycles occur at the end of object processing. If REFRATE is zero refresh is disabled.
12	BIGEND	Specifies that big-endian addressing should be used. This determines the address of a byte within a phrase and allows the Jaguar to be used comfortably with Big-endian (Motorola) processors or with Little-endian (Intel) processors.
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 for BIGEND which is determined by external pull-up / pull-down resistors.

HC Horizontal Count *F00004* *RW*

This register is comprised 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 <u>only</u> for ASIC test purposes**.

VC Vertical Count F00006 RW

This register is comprised 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 <u>only</u> written to for ASIC test purposes**.

LPH Horizontal Light-Pen F00008 RO

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

LPV Vertical Light-Pen F0000A RO

The low eleven bits of this register gives the vertical position of the light-pen in half lines.

OB [0-3] Object Code F00010-16 RO

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

OLP Object List Pointer F00020 WO

This 32-bit register 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 21 of this address are replaced by the LINK data in the object. The value stored in this register should be word-swapped. **Because the Object Processor could interrupt the 68000 in the middle of a write to this register, the 68000 should never be used to change the OLP. Use the GPU instead.**

OBF Object Processor Flag F00026 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. A write (of anything) to this register restarts the Object Processor after a Graphics Processor interrupt object.

VMODE Video Mode F00028 WO

Bits	Name	Description
0	VIDEN	When set this bit enables the time-base generator. **This should never be set to zero in a Jaguar Console.**
1-2	MODE	Determines how the line buffer contents are translated into physical pixels.
	CRY16 (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 look-up tables and multipliers. CRY16 pixels are arranged as follows: Bit 15 … Bit 0 C C C C R R R R Y Y Y Y Y Y Y Y

<table>
<tr><td></td><td></td><td>The least-significant bit is normally interpreted as the least-significant bit of intensity. If VARMOD is also set, this bit will be cleared to indicate a CRY16 pixel and only the top seven bits will be used to determine intensity.</td></tr>
<tr><td></td><td>RGB24 (1)</td><td>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. RGB24 pixels are arranged as follows:
Bit 31 … Bit 16
G G G G G G G G R R R R R R R R
X X X X X X X X B B B B B B B B
Bit 15 … Bit 0</td></tr>
<tr><td></td><td>DIRECT16 (2)</td><td>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 the video clock. It is assumed that further multiplexing and colour look-up will occur outside the chip. In this mode blanking and video active are output on the two least significant bits of Blue.</td></tr>
<tr><td></td><td>RGB16 (3)</td><td>16-bit RGB. Each 32-bit entry in the line buffer is treated as two 16-bit RGB pixels. RGB16 pixels are arranged as follows:
Bit 15 … Bit 0
R R R R R B B B B B G G G G G G
The least-significant bit is normally interpreted as the least-significant bit of Green. If VARMOD is also set, this bit will be set to indicate a RGB16 pixel and only the top five bits will be used to determine the level of Green.</td></tr>
<tr><td>3</td><td>GENLOCK</td><td>Not supported in the Jaguar console - always write zero.</td></tr>
<tr><td>4</td><td>INCEN</td><td>Enables encrustation. When set, the least-significant bit of the 16 bit data 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.</td></tr>
<tr><td>5</td><td>BINC</td><td>Selects the local border colour if encrustation is enabled.</td></tr>
<tr><td>6</td><td>CSYNC</td><td>Enables composite sync on the vertical sync output.</td></tr>
<tr><td>7</td><td>BGEN</td><td>Clears the line buffer to the colour in the background register after displaying the contents. This only has effect in CRY and RGB16 modes.</td></tr>
<tr><td>8</td><td>VARMOD</td><td>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 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 the Jaguar to support an RGB window against a CRY background for instance.</td></tr>
<tr><td>9-11</td><td>PWIDTH1-8</td><td>This field determines the width of pixels in video clock cycles.
The width is one more then the value in this field.
The video time base generator is programmed in cycles of the video</td></tr>
</table>

		clock and not the pixel clock produced by this divider. The display width should be set to be an integer number of pixels, i.e. an integer multiple of the pixel width programmed here.
12-15	Unused	Write zeroes

BORD1	**Border Colour (Red & Green)**	***F0002A***	***WO***
BORD2	**Border Colour (Blue)**	***F0002C***	***WO***

These registers determine the physical border colour. There are eight bits per primary colour. Red is the least 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**	***F0002E***	***WO***
Do NOT Modify: For Information Only			

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

HBB	**Horizontal Blanking Begin**	***F00030***	***WO***
Do NOT Modify: For Information Only			

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**	***F00032***	***WO***
Do NOT Modify: For Information Only			

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 on the line.

HS	**Horizontal Sync**	***F00034***	***WO***
Do NOT Modify: For Information Only			

This eleven bit register determines the width of the horizontal sync and equalisation pulses. The pulses start when the horizontal count equals the value in this 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 equalisation pulses which are the same width as the horizontal sync but which appear twice per line (for 10 half lines during field blanking).

HVS	**Horizontal Vertical Sync**	***F00036***	***WO***
Do NOT Modify: For Information Only			

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 equalisation pulses but ends when the least significant ten bits of the horizontal count match the HVS register.

 © *Stephen Moss*

HDB1	**Horizontal Display Begin 1**	***F00038***	***WO***
HDB2	**Horizontal Display Begin 2**	***F0003A***	***WO***

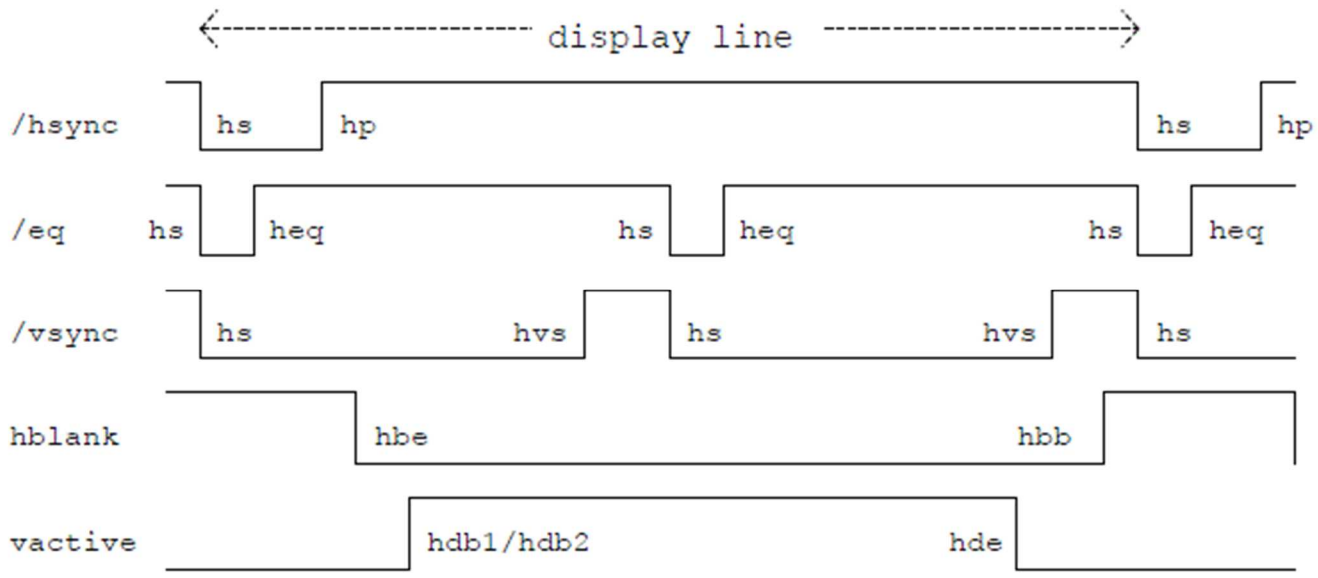
These eleven bit registers control where on the display line the Object Processor starts. When the horizontal count matches either of the above registers the Object Processor starts execution at the address in OLP, the line buffers swap over and pixels are shifted out of the line buffer.

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 line 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**	***F0003C***	***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.



VP	**Vertical Period**	***F0003E***	***WO***
Do NOT Modify: For Information Only			

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	F00040	WO
Do NOT Modify: For Information Only			

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

VBE	Vertical Blanking End	F00042	WO
Do NOT Modify: For Information Only			

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

VS	Vertical Sync	F00044	WO
Do NOT Modify: For Information Only			

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	F00046	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	F00048	WO

This eleven bit register specifies the half line at which object processing ends. **Due to a bug in the Jaguar Console, this register should be set at $FFFF to cause the Object Processor to process every line**.

VEB	Vertical Equalisation Begin	F0004A	WO
Do NOT Modify: For Information Only			

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

VEE	Vertical Equalisation End	F0004C	WO
Do NOT Modify: For Information Only			

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

VI	Vertical Interrupt	F0004E	WO

This eleven bit register specifies the half line on which the VI interrupt is generated. This must be odd if the display is non-interlaced. This interrupt will occur once per frame when interlaced, that is every other field.

PIT [0-1]	Programmable Timer Interrupt	F00050-52	WO

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

 © *Stephen Moss*

The system clock 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 of the second register and the output of this divider generates the interrupt.

HEQ Horizontal Equalisation End *F00054* *WO*
Do NOT Modify: For Information Only

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

BG Background Colour *F00058* *WO*

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

INT1 CPU Interrupt Control Register *F000E0* *RW*

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

Equate	Bit	Interrupt	Description
C_VIDENA	0	Video	This interrupt is generated by the video time-base, on the line selected by the VI register.
C_GPUENA	1	GPU	This interrupt is generated by the Graphics Processor writing to an internal register.
C_OPENA	2	Object	This interrupt is generated by stop objects.
C_PITENA	3	Timer	This interrupt is generated by the PIT.
C_JERENA	4	Jerry	This CPU interrupt is generated by an input to Tom and is intended for use by Jerry (Jerry tells Tom to interrupt the CPU). This is an active high edge-triggered interrupt – the first interrupt will occur on the first rising edge after it has been enabled.
C_VIDCLR	8	Video	When set, this bit clears pending video time-base interrupts.
C_GPUCLR	9	GPU	When set, this bit clears pending GPU interrupts.
C_OPCLR	10	Object	When set, this bit clears pending Object Processor stop object interrupts.
C_PITCLR	11	Timer	When set, this bit clears pending PIT interrupts.
C_JERCLR	12	Jerry	When set, this bit clears pending Jerry interrupts.

When written to bits 0 to 4 enable the individual interrupt sources, i.e. if bit 1 is set the Graphics Processor interrupt is enabled and bits 8 to 12 clear pending interrupts from the corresponding interrupt source. When read bits 0-4 indicate which interrupts are pending, i.e. if bit 3 is set there is a timer interrupt pending, the remaining bits are unused.

Note that the INT2 register must always be written to at the end of a CPU interrupt service routine.

INT2 CPU Interrupt resume Register *F000E2* *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 any value to this register. This should therefore always be done at the end of an interrupt service

routine. After a write to this register the Blitter and GPU may then restart, and no further CPU instructions will be executed until either the next interrupt occurs, or the GPU or Blitter operation completes.

CLUT	Colour Look-Up Table	*F00400-7FE*	*RW*

The colour look-up table translates an eight bit colour index into a 16-bit physical 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. Locations in the range F00400-5FE read from table A. Locations in the range F00600-7FE read from table B. Writing to either range writes to both tables. **Writes to this region of memory may be unreliable when an object with the 'Release' bit is part of the current object list.**

LBUF	Line Buffer	*F00800-0D9E* *F01000-159E* *F01800-1D9E*	*RW*

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 to as two 16-bit words. In 16-bit CRY mode each word is a CRY pixel; the least 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 least significant byte of the word at the lower address is the Red value. The most significant byte is the Green value and the least significant byte of the word at the high address is the Blue value. The forth 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

Jerry and external peripherals occupy the 64K above the internal memory. All peripheral memory is 16 bits wide although it is likely that many devices will have 8 bit busses.

Object Definitions

There are five basic types.

BITOBJ	Bit Mapped Object

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

 © *Stephen Moss*

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 give 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. Please note that for scaled bitmap objects, HEIGHT should actually be the bitmap height -1.
24-42	LINK	This defines the address of the next object. These nineteen bits replace bits 3 to 21 in the OLP register. This allows an object to link to another object within the same 4 Mbytes.
43-63	DATA	This defines where the pixel data can be found. Like LINK this is a phrase address. These twenty-one bits define bits 3 to 23 of the data address. This allows object data to be positioned anywhere in memory. After a line is displayed the new 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: **Value** / **Bits per Pixel** / **Type** / **Video Modes Allowed In** 0 / 1 bit/pixel / CLUT / CRY16, RGB16 & DIRECT16 1 / 2 bits/pixel / CLUT / “ “ “ 2 / 4 bits/pixel / CLUT / “ “ “ 3 / 8 bits/pixel / CLUT / “ “ “ 4 / 16 bits/pixel / Direct / “ “ “ 5 / 32 bits/pixel / Direct / RGB24
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* PITCH 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, i.e. data for the next line of pixels

		can be found at DATA + (8 * DWIDTH).
28-37	IWIDTH	This is the image width in phrases (must be non zero). May be used for clipping.
38-44	INDEX	For images with 1-4 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 an object from right to left.
46	RMW	Flag to add object to data in the line buffer. The values are then signed offsets for intensity and the two colour vectors. It is possible for the last column of pixels of a RMW (Read-Modify-Write) object to be corrupted if it is followed by another bitmap object. This will happen on the right side unless the REFLECT bit is set, in which case it will happen on the left side. To work around this problem, you can ensure that the last pixels of the data source are all transparent (i.e. pad the object data). Or you can make sure that the next object in the object list will not appear on the same scan lines as the RMW object. Or you can place an always-false branch object after the RMW object.
47	TRANS	Flag to make logical colour zero 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 (1 to 8 bits-per-pixel) 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. This bit may be set, however, in 16-bit *scaled* bitmap objects. External bus masters, the refresh mechanism and the Graphic 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.

SCBITOBJ Scaled Bit Mapped Object

This object displays a scaled bit mapped object. The object must be on a **32** byte boundary in 64 bit RAM. Scaled bitmaps will not display properly in 24-bit RGB mode. 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 pixel source.

		HSCALE can be set to values as high as 7.1F (%111.11111), however a 24-bit scaled object will be distorted if HSCALE is set to any value other than 1.0 (%001.0000)
8-15	VSCALE	This eight bit field contain 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. Setting the VSCALE value of a scaled bitmap to greater than 7.0 (%111.00000) will fail.
16-23	REMAINDER	This eight bit field contains a three bit integer part and a five bit fractional part. The number determines how 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. This value should be initialised to the same value as VSCALE to produce a perfectly scaled first line.
24-63		Unused, write zeroes.

GPUOBJ 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 OBF (Object Processor Flag) register.

Bits	Field	Description
0-2	TYPE	GPU object is type two.
3-63	DATA	These bits may be used by the GPU interrupt service routine. They are memory mapped in the object code registers OB[0-3], 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.

BRANCHOBJ 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.
14-16	CC	These bits specify what condition is used to determine where to continue processing: 0 Branch to LINK if YPOS == VC or YPOS == 7FF 1 Branch to LINK if YPOS > VC 2 Branch to LINK if YPOS < VC

		3 Branch to LINK if YPOS if Object Processor flag is set 4 Branch to LINK if on the second half of the display line (HC10 = 1)
17-23	Unused	
24-42	LINK	This defines the address of the next object if the branch is taken. The address is defines as described for the bit mapped object.
43-63	Unused	

STOPOBJ Stop Object

This object stops object processing and interrupts the host.

Bits	Field	Description
0-2	TYPE	Stop object is type four.
3	INT FLAG	When set, CPU stop object interrupts are enabled.
4-63	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.

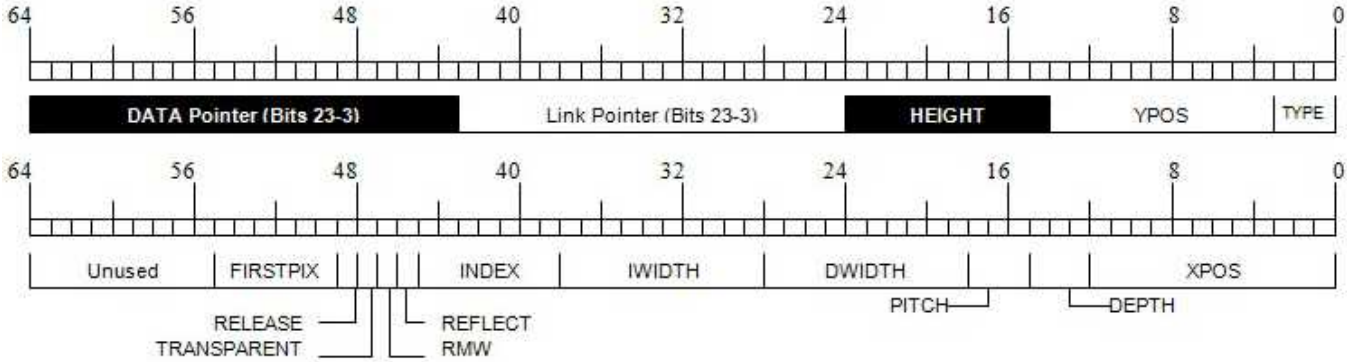
© *Stephen Moss*

Object Processor Quick Reference

(Inverted fields are modified by the Object Processor)

Bitmap Object

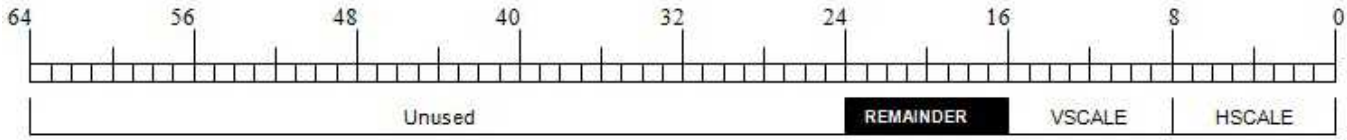
TYPE = 0



Scaled Bitmap Object

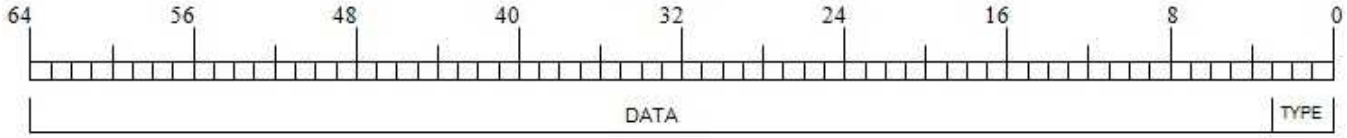
TYPE = 1

(Third phrase only. Phrases one and two are the same as a Bitmap Object)



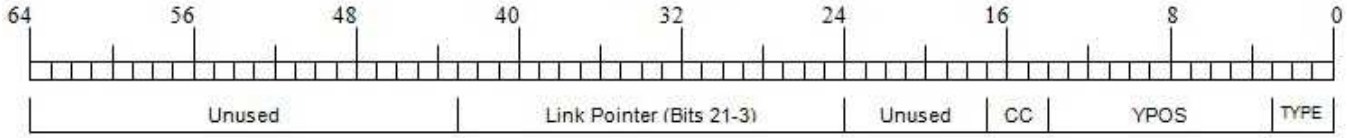
GPU Interrupt Object

TYPE = 2



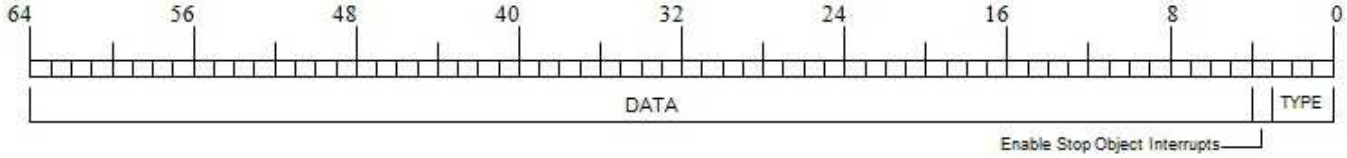
Branch Object

TYPE = 3



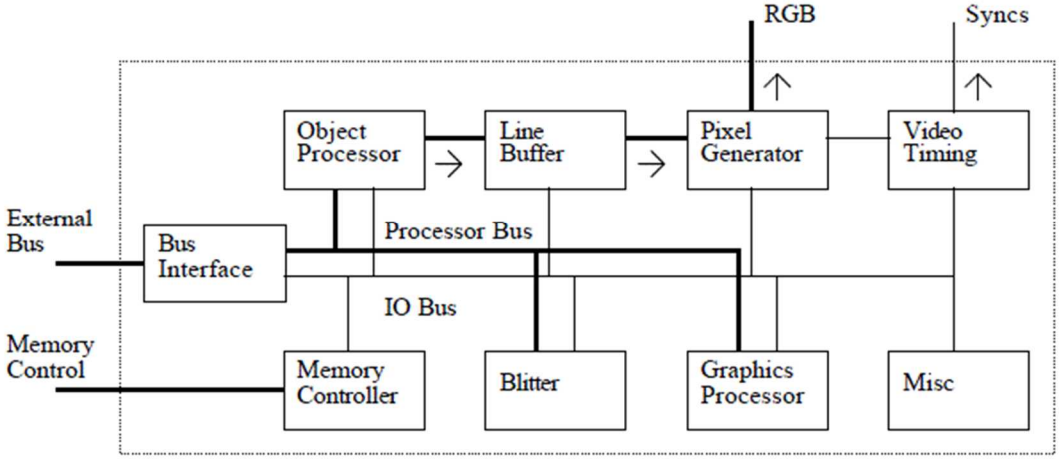
Stop Object

TYPE = 4



Description of the Object Processor/Pixel Path

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

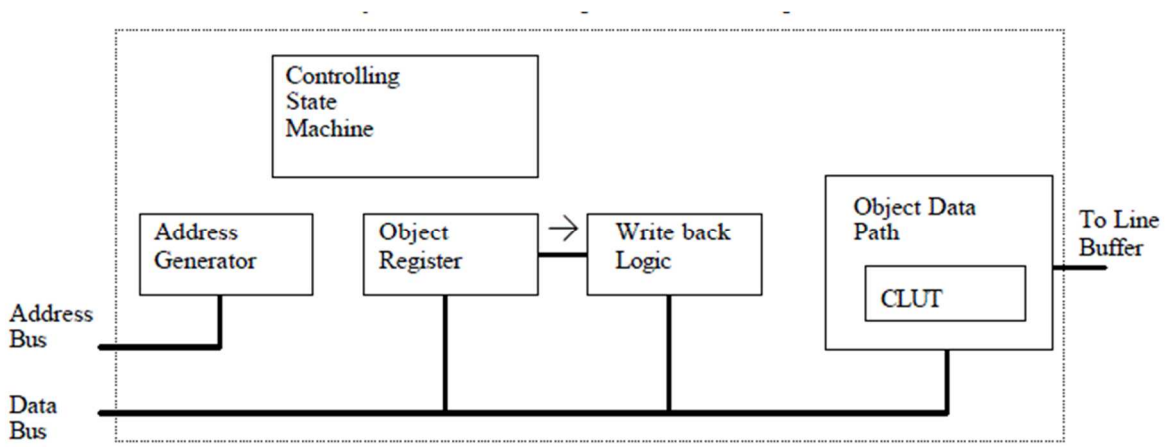


Jaguar Chip Block Diagram

The processor bus is a 64-bit data, 24-bit 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-bit data, 16-bit address bus used for reading and writing to internal memory and registers. The bus interface logic and memory controller allows transfer 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 much 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 360 x 32 RAM with independent write strobes for the high and low words.

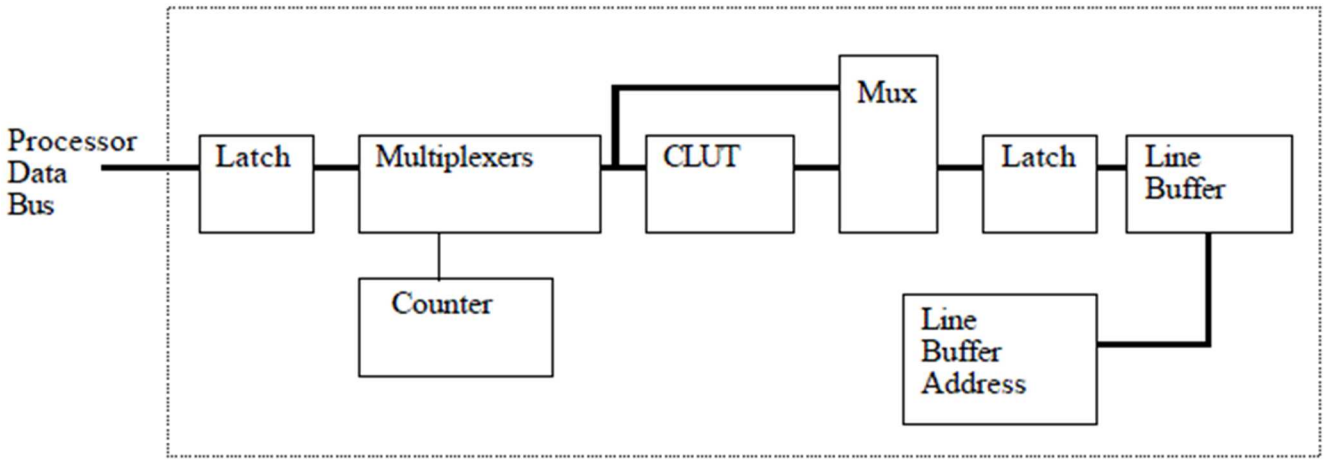
Each location in the line buffer may contain one 24-bit pixel or two 16-bit pixels.



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 in to physical colours by the Colour Look Up Table (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 co-ordinate) 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.

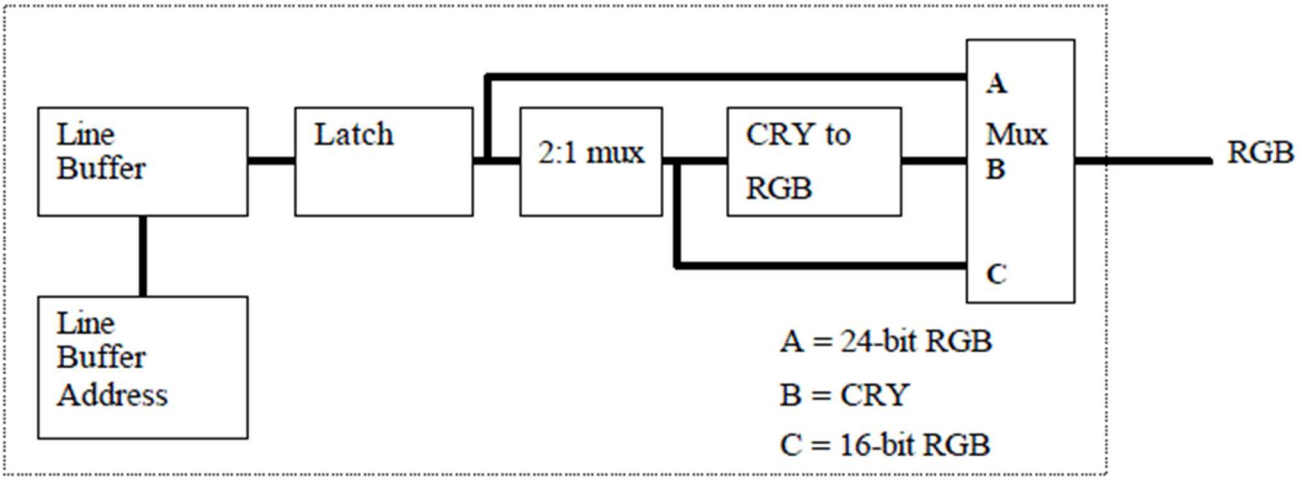
If an object is scaled the Object Processor deals with one pixel at a time, not pairs. Scaling is achieved by incrementing the line buffer address independently of the counter controlling the multiplexer. For instance, if the line buffer address is incremented twice as often as the counter then the image 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 my 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, this is the video clock.



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 most significant eight bits specify the colour and the least significant bits specify the intensity or brightness. The colour value is used as an index to three ROMs; these ROMs contain 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 bit 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 DAC's. 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 multiplexer 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 four times the video clock. 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 but 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.

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 be 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.

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

Refresh cycles are also invoked when the Object Processor reaches the end of the object line. After the Object Processor executes a STOP object the Jaguar performs as many refresh cycles as are necessary to decrement the refresh counter to zero.

This mechanism guarantees that he minimum refresh rate is maintained without interrupting the Object Processor and without creating “dead spots” of more than a few microseconds.

Colour Mapping

Introduction

The 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*.

The 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 the 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 be stored as six bits of green, and five bits each of 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.

The CRY Colour Scheme

Gouraud Shading Requirements

The CRY scheme was derived principally to meet the requirements of *Gouraud Shading*. This is technique that models the appearance of a lit curved surface from a set of polygons. The problem the technique helps to overcome is that 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 concerned, then the only variation is one of luminous intensity, and not one of colour. It is therefore attractive to have a colour scheme that 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.

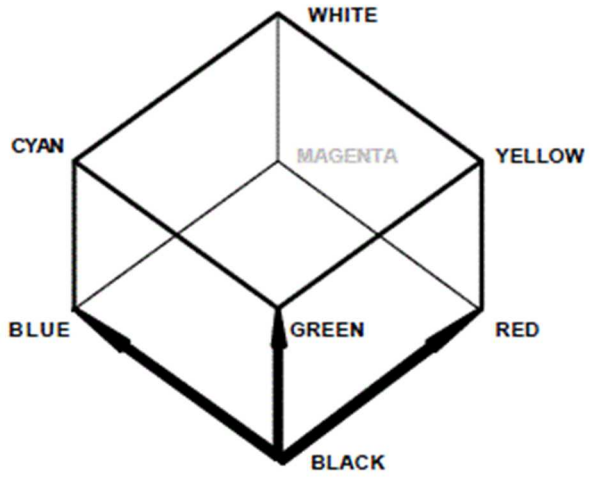
As there is a general agreement that eight bits is enough to give smooth intensity shading (and it is a round number), it was therefore necessary to come up with a scheme that allowed the colour to be

 © *Stephen Moss*

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 represents 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 represent yellow, cyan, and magenta.



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 that 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 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 spaced 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, theta* scheme was discarded as it would not meet requirement two, and so a scheme based on two *x, y* vectors was chosen.

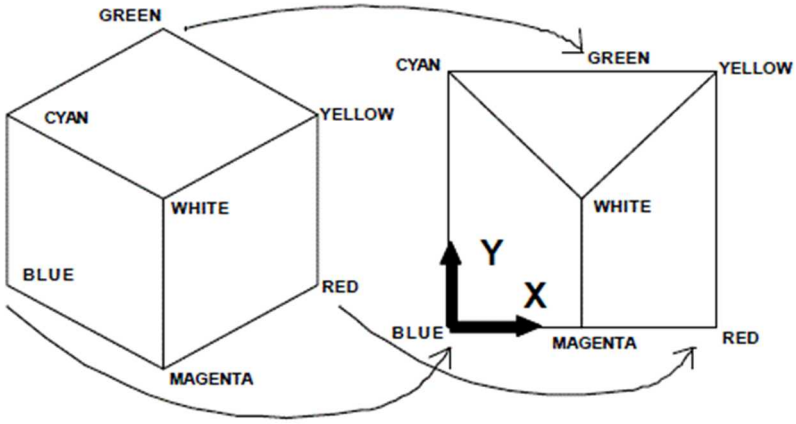
To meet requirement one, the two vectors must describe a point on a square area. As no existing colour space model is square when viewed along the intensity axis, it was necessary to come up with a new one.

The approach chosen, 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 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 into 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 that lies in 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.



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 at 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 midway point 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:

- 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.

RGB to CRY conversion

The best technique is to calculate the intensity value, which is the largest of red, green and blue; and from this the ideal ROM entry for that colour, by scaling the RGB values by 255 / intensity. This can then be matched to the actual ROM tables to find the nearest match. A quick way of doing this is by a look-up table. It is not necessary for this to have 2^{24} entries, it turns out that taking the top five bits of each of red, green and blue values (rounding where appropriate) and using a 32768 element look-up table is adequate.

 © *Stephen Moss*

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 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	255	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	255	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

Graphic Processor Subsystem

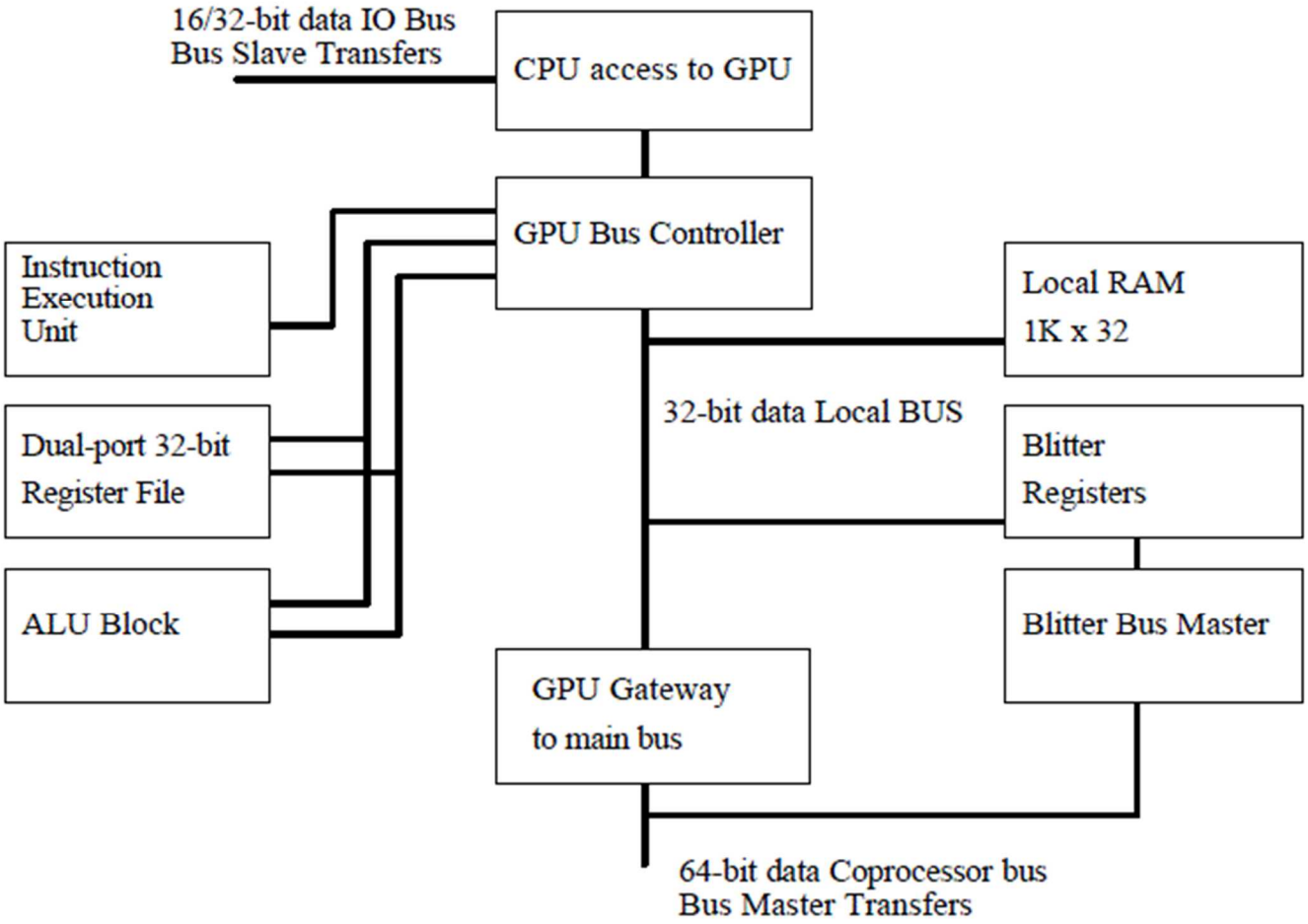
The Graphics Subsystem of the Jaguar is a self-contained processing unit, whose view of the external system processor and memory are controlled by a separate memory controller, which is not part of the graphics system.

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 24-bit address, with byte resolution. This bus has multiple masters, and ownership of it is gained by a bus request / acknowledge system, which is prioritized, 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 and 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:



Note: Local RAM – 1K Bytes (1000 address of 32 bit data)

 © *Stephen Moss*

Memory Map

The Graphics Subsystem address space contains the following locations:

F02100	G_FLAGS	RW	GPU flags
F02104	G_MTXC	W	GPU matrix control
F02108	G_MTXA	W	GPU matrix address
F0210C	G_END	W	GPU big / little endian control
F02110	G_PC	RW	GPU program counter
F02114	G_CTRL	RW	GPU operation control / status
F02118	G_HIDATA	RW	GPU bus interface high data
F0211C	G_DIVCTRL	W	GPU division method
F0211C	G_REMAIN	R	GPU division remainder
F02200	A1_BASE	W	Blitter A1 base
F02204	A1_FLAGS	W	Blitter A1 flags
F02208	A1_CLIP	W	Blitter A1 clipping size
F0220C	A1_PIXEL	RW	Blitter A1 pixel pointer
F02210	A1_STEP	W	Blitter A1 step
F02214	A1_FSTEP	W	Blitter A1 step fraction
F02218	A1_FPIXEL	RW	Blitter A1 pixel pointer fraction
F0221C	A1_INC	W	Blitter A1 pixel pointer increment
F02220	A1_FINC	W	Blitter A1 pixel pointer increment fraction
F02224	A2_BASE	W	Blitter A2 base
F02228	A2_FLAGS	W	Blitter A2 flags
F0222C	A2_MASK	W	Blitter A2 mask
F02230	A2_PIXEL	RW	Blitter A2 pixel pointer
F02234	A2_STEP	W	Blitter A2 step
F02238	B_CMD	W	Blitter command
F0223C	B_COUNT	W	Blitter loop counters
F02240	B_SRCD	W	Blitter source data
F02248	B_DSTD	W	Blitter destination data
F02250	B_DSTZ	W	Blitter destination Z data
F02258	B_SRCZ1	W	Blitter source Z data 1
F02260	B_SRCZ2	W	Blitter source Z data 2
F02268	B_PATD	W	Blitter pattern data
F02270	B_IINC	W	Blitter intensity increment
F02274	B_ZINC	W	Blitter Z increment
F02278	B_STOP	W	Blitter collision stop control
F0227C	B_I3	W	Blitter intensity register 3
F02280	B_I2	W	Blitter intensity register 2
F02284	B_I1	W	Blitter intensity register 1
F02288	B_I0	W	Blitter intensity register 0
F0228C	B_Z3	W	Blitter Z register 3
F02290	B_Z2	W	Blitter Z register 2
F02294	B_Z1	W	Blitter Z register 1
F02298	B_Z0	W	Blitter Z register 0
F03000	G_RAM	RW	Local RAM base

These locations may be accessed by all processors except the GPU for read or write as appropriate at the above addresses, where they appear to the system as 16-bit memory. As they are all actually 32-bits, transfers should always be performed in pairs, in the order low address then high address.

In addition, for high-speed write operations by 32-bit or 64-bit bus masters (especially for blit transfers), they may be written to as 32-bit locations at an offset of plus 8000 hex from the addresses above. They are not readable at these addresses.

The GPU addresses them all directly as 32-bit locations in 32-bit internal memory, and they are not accessible to the GPU at the plus 8000 hex offset.

 © *Stephen Moss*

Graphics Processor

This section describes the Jaguar Graphics Processor (GPU).

What is the Graphics Processor?

The Graphics Processor (called here the GPU – Graphics Processor Unit) is a simple, very fast, micro-processor. It is 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 RISC (Reduced Instruction Set Computer) Processor. 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
- One instruction per tick peak throughput
- Internal program and 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

Programming the Graphics Processor

The GPU is programmed in the same way as any other microprocessor. It has a full instruction set with a broad range of arithmetic instructions, including add, subtract, multiply and divide; Boolean instructions, and bit-wise instructions. It has a range of instructions for loading and storing values in memory, with either register indirect, register indirect plus register offset, or register indirect plus immediate offset addressing modes. It has jump relative and absolute instructions, both of which may be made dependent 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 (addresses) 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.

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. 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.

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:

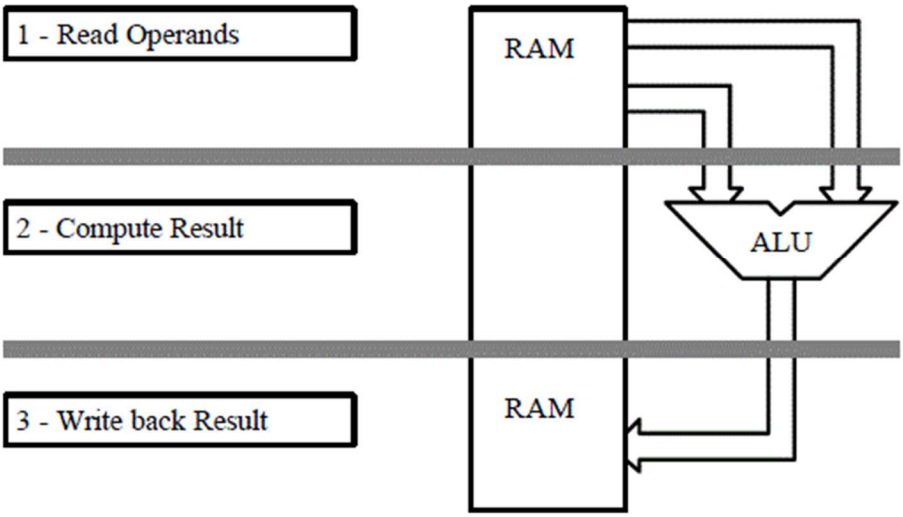
1. Decode instruction
2. Read operands from registers
3. Add operands
4. Write result back to register

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.

 © *Stephen Moss*

Register Score-Boarding

The main side effect of the pipe-lined nature of GPU operation is the interaction of instructions at different stages of the pipe-line. They may affect the same operand, or the same piece of the hardware, and so a conflict can potentially arise.



For instance, if the instruction after an ADD was a second ADD of another value to the same register; 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 that occur during these hold-ups are referred to as *wait states*.

The figure shows the data flow associated with the operands of an arithmetic instruction. The thick lines correspond to a pipeline 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.

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.

The GPU operates what is known as a *score-board* to help the programmer avoid a whole class of these problems. This tags registers that 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 that is still in the process of being computed by the ALU.

--- An instruction would perform a conditional jump, or add or subtract with carry, before the flags have been set as the result of some arithmetic operation.

--- An instruction would read a register that is being read from internal memory.

--- An instruction would read a register that 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 that is waiting to be loaded from slow external memory (which takes a variable amount of time).

The scoreboard mechanism does not work on the data of any indexed store instruction. This means that any indexed store instruction that stores data from a long latency operation (such as a divide or external load) should place an "**or**" instruction prior to the store. For example:

```
div           r0,r3
store  r3,(r14+6)
```

should be written as:

```
div           r0,r3
or            r0,r3
store  r3,(r14+6)
```

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 1 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 that 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.

One technique that 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.

In any instruction where the destination register is written to without being read, the destination register will not be protected by the score-boarding mechanism of the GPU/DSP. This includes MTOI, MORMI, RESMAC, all MOVE variations, and all LOAD variations.

If one of these 'destination write-only' instructions writes to the same destination register as a prior instruction and there have been no intervening reads from that register, it is possible for the second instruction to complete before (or simultaneously with) the first, causing the register to become corrupt. This bug only becomes a problem when doing 'dummy' instructions as shown in the following example:

```
  div     r2,r4   ;Divide starts (takes 18 ticks)
  moveq   #4,r4   ;Move completes before divide
```

Although this code doesn't make much sense, it might appear at the end of a loop as shown below:

```
Loop:
     jr      EQ, loop
     div     r2,r4

;;;
```

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 jump instructions with a NOP (Null Operation), but it is quite reasonable to place almost any other instruction here – but see the notes below on program control flow.

Neither the DSP or GPU will reliably execute '**jr**' or '**jump**' instructions unless they are in internal RAM.

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 four Kilobytes of local memory, organized 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 load up a program into the RAM within the GPU, the best technique is to use the Blitter. Set it to blit phrases, and use the 32-bit GPU address range (see below).

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 programming 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 is requested, a special register is used for the other half of the data.

The address space is organized as follows:

F02000 – F021FF	Graphics processor control registers
F02200 – F022FF	Blitter registers
F02300 – F02FFF	Reserved
F03000 – F03FFF	Local RAM (1K hexadecimal locations)
F04000 – F0FFFF	Reserved

This local address space is also available to external devices via the I/O mechanism.

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

External View of GPU Space

The GPU internal address space is accessible by any other Jaguar bus master, i.e. the CPU, the Blitter and the DSP can all access GPU internal space. This is part of the Jaguar I/O space within Tom. This is normally viewed as 16-bit read/write memory, but by adding 8000 hex to the addresses it is also available as 32-bit write only memory, which is faster to access for a bus master which can perform 32-bit transfers. Specifically, this allows the Blitter to copy data into the GPU space more rapidly that it would using the 16-bit space – for maximum transfer speed use the Blitter in phrase mode, writing to the 32-bit address range. Please note that the 68000 in the Jaguar Console may not address this 32-bit wide memory.

Transfers to/from addresses within the range $F02000-$F07FFF and $F1A000-$F1F000 are executed 32 bits at a time using a latch mechanism and must be handled carefully by external processors (see External CPU Access). When a 16-bit word is read from the GPU at a long-word aligned address, a 32-bit read is performed. The high word is transferred and the low word is latched. Any 16-bit read operation at a GPU long-word aligned address + $2 simply transfers the latched data.

When a 16-bit word is written to a long-word aligned address, the data is latched. When a 16-bit word is written to a long-word aligned address + $2, 32-bits (the written word and latch data) are transferred.

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 either-endian - this means that the first instruction of the pair in a long-word is programmable. This is controlled by the BIG_INST bit.

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 either read to supply data to be stored or is written with load data.

Loads and stores may be performed at byte, word, long-word or 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 one of two simple indexed addressing schemes. These are both based on using either R14 or R15 as a base register, with either a five bit unsigned offset (in long-words) encoded into one of the register fields or another register containing the offset. There is

a two tick overhead involved in using these instructions, as the address has to be 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.

Due to a bug in the Jaguar Console, DMA transfers are not permitted. The DMEAN bit of the G_FLAGS register must be cleared to 0.

The value in the High Data Register of the GPU is changed after ANY external **load**, not just **loadp**. This means that if an interrupt is running in the GPU that loads from external memory the underlying program may not use **loadp**.

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 by 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.

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 RAM address. 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:

#	Interrupt
4	Blitter
3	Object Processor
2	Timing generator
1	Jerry Interrupt
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 normally 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  #G_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 instruction to be executed
     addq   #4,r31         ; update 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 underlying code as they are corrupted (you may choose to use any two registers in bank #0), in addition to R30 and R31 which are always corrupted by the interrupt process itself. Note: R30 is automatically corrupted when an interrupt occurs not just by the interrupt service code as shown.

— 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.

If the interrupt source was the Object Processor, then the interrupt service routine should read the Object Code registers, if required, and then re-start the Object Processor by writing to the Object Processor Flag register, as quickly as possible.

Atomic Operations

It is necessary for certain operations to be atomic, i.e. interrupts may not occur during these operations. 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 the 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 that succeeds them.

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.

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 an 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 being set (this is the read version of the SINGLE_STEP 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).

Illegal Instruction Combinations

- 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.
- Do not place a MOVE PC to register instruction immediately after a jump, the value read cannot be relied upon.
- Do not follow an IMULTN instruction by anything other than an IMACN instruction.
- Do not follow an IMACN instruction by anything other than another IMACN instruction or a RESMAC instruction (see below).
- Do not precede an MMULT instruction by a LOAD or STORE instruction.

Conditional Jumps

Conditional jumps encode from a five bit flag field. This is:

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.

This gives useful jumps as follows (other codes are either jump always or jump never, and are reserved for future modifications)

Code	#	Condition	Description
00000	0		Jump always
00001	1	NZ	Jump if zero flag is clear
00010	2	Z	Jump if zero flag is set
00100	4	NC	Jump if carry flag is clear
00101	5	NC NZ	Jump if carry flag is clear and zero flag is clear
00110	6	NC Z	Jump if carry flag is clear and zero flag is set
01000	8	C	Jump if carry flag is set
01001	9	C NZ	Jump if carry flag is set and zero flag is clear
01010	A	C Z	Jump if carry flag is set and zero flag is set
10100	14	NN	Jump if negative flag is clear
10101	15	NN NZ	Jump if negative flag is clear and zero flag is clear
10110	16	NN Z	Jump if negative flag is clear and zero flag is set
11000	18	N	Jump if negative flag is set
11001	19	N NZ	Jump if negative flag is set and zero flag is clear
11010	1A	N Z	Jump if negative flag is set and zero flag is set
11111	1F		Jump never

 © *Stephen Moss*

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 times 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.

Systolic Matrix Multiples

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, and is the *raison d'etre* of the second 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.

Divide Unit

The divide unit performs unsigned division, taking as operands 32-bit divisor and dividend, giving a 32-bit quotient and a 32-bit remainder. The quotient is the result of the divide instruction, and replaces the dividend in the destination register. Divides are performed at the rate of two bits per tick, so that the complete divide operation completes in sixteen ticks. The divide instruction has no effect on the flags.

If another instruction attempts to read the quotient or start another divide operation while the divide unit is active, then wait states will be inserted until the divide unit has completed.

The remainder register may be read after the divide has completed, this value in this register may either be positive, in which case it contains the actual remainder, or negative, in which case it contains the remainder minus the divisor.

Divides may also be performed on unsigned 16.16 bit values, by setting the offset control flag in the divide control register. The quotient is then also an unsigned 16.16 bit value.

There is a bug in the divider of the GPU and DSP. If you try to do two consecutive divides without there being at least 1 clock cycle of idle time between them, then the result of the second divide will be wrong.

This will only occur when the two divides are separated by less than 16 clock cycles , and the second divide has the quotient of the first divide as one of its register operands.

The work-around should be to either make sure that more than 16 clock cycles occur between divide instructions, or make sure that an instruction ich is dependant on the quotient of the first divide occurs before the second divide.

Example #1:

```
     div      r0,r1
      moveq    #3.r5
      div      r5,r1
```

Should be like this:

```
     div      r0,r1
      moveq    #3.r5
      or       r1,r1
      div      r5,r1
```

Example #2:

```
      div      r0,r1
      moveq    #3.r5
      div      r5,r1
```

Should be like this:

```
      div      r0,r1
      moveq    #3.r5
      or       r1,r1
      div      r5,r1
```

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. This is forced by the IMASK bit, when it is set selection of bank 0 is forced. If IMASK is clear REGPAGE is obeyed.

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

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 reading or writing at the addresses given elsewhere in this document, and these locations are presented as sixteen bit memory, which must always be accessed as long words 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. The 68000 may not access this memory as it transfers data 16-bits at a time.

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.

The **clr.l <ea>** And **move.l <ea>,-(an)** instructions of the 68000 do not work correctly when writing to Jaguar GPU & DSP hardware registers and internal RAM.

The address ranges with this restriction are $F02000 to $F07FFF and $F1A000 to $F1F000. These instructions may be safely used on memory addresses outside these ranges.

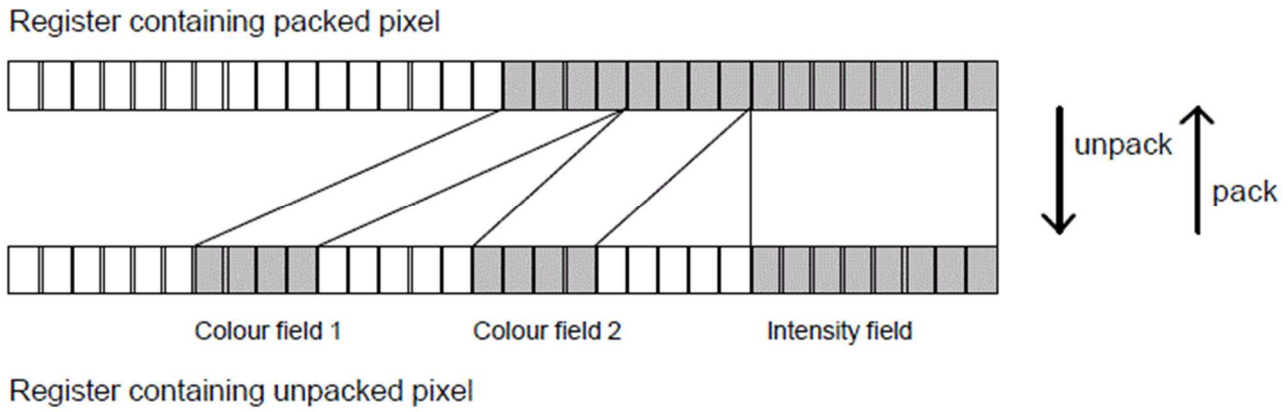
Because the 68000 has a 16-bit data bus, 32-bit writes to memory actually occur as two separate 16-bit writes which happen in succession. With certain instructions such as those shown above, the order in which the high word and low word are written is reversed, which causes problems when writing to these address ranges.

While these are the only ones known about at present, it is possible there are other instruction/address mode combinations that have this problem. The best way around it is to use the GPU and/or DSP instead of the 68000 when you want to write to the Jaguar GPU/DSP registers, and to use the Blitter when you want to copy information into the GPU or DSP RAM.

If you are using a high-level language compiler make sure that it does not generate **clr.l** instructions for code that accesses this address space.

Pack and Unpack

The **pack** and **unpack** instructions provide a means for averaging up to 32 CRY pixels. The unpack operation leaves the intensity value unchanged, shifts the lower colour nibble up 5 bits, and the higher colour nibble up 10 bits. The pack operation reverses this:



There are five unused bits above each field in an unpacked pixel, allowing up to 32 unpacked pixels to be added together. If a power of two unpacked pixel values are added, then a shift can be used to re-align them prior to packing the average value.

The bits that do not contain packed or unpacked pixel data are always set to zero.

This is useful for anti-aliasing and scaling effects.

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.

G_FLAGS GPU Flags Register *F02100* *RW*

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

Bits	Equate(s)	Description
0	ZERO_FLAG	The ALU zero flag is 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 is set or cleared by a carry/borrow out of the adder/subtract, and reflects carry out of some shift operations, but it is not defined after other arithmetic operations.
2	NEGA_FLAG	The ALU negative flag is set if the result of the last arithmetic operation was negative.
3	IMASK	The Interrupt mask is 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	G_CPUENA G_JERENA G_PITENA G_OPENA G_BLITENA	Interrupt enable bits for interrupts 0-4. The status of these bits is overridden by IMASK. The meaning of these bits are: 0 CPU Interrupt 1 Jerry Interrupt 2 Timing Generator

 © *Stephen Moss*

		3 Object Processor 4 Blitter
9-13	G_CPUCLR G_JERCLR G_PITCLR G_OPCLR G_BLITCLR	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	**This bit must not be set due to a bug in the Jaguar Console. Write as Zero only.**

Values written to the G_FLAGS resister may not appear to have changed in the following two instructions due to pipe-lining effects.
Consequently, writing a value to the flag bits and making use of those flag bits in the following instruction will not work properly. If it is necessary to use flags set by a STORE instruction, then ensure that at least two other instructions lie between the STORE and the flags dependent instruction.

If it is necessary to use flags set by an indexed STORE instruction, then ensure that at least four other instructions lie between the STORE and the flags dependent instruction.

G_MTXC Matrix control Register *F02104* *WO*

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

Bits	Equate(s)	Description
0-3	MATRIX3-15	Matrix width, in the range 3 to 15
4	MATCOL	When set, this control bit make the matrix held in memory be accessed down one column, as opposed to along one row.

G_MTXA Matrix Address Register *F02108* *WO*

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

Bits	Equate(s)	Description
2-11	---	Matrix address.

G_END Data Organisation Register *F0210C* *WO*

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

Bits	Equate(s)	Description
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 organization is big-endian. See the discussion elsewhere in this document.
2	BIG_INST	When this bit is set the order of word program fetches is big-endian.

G_PC GPU Program Counter *F02110* *RW*

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.

G_CTRL GPU Control/Status Register *F02114* *RW*

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

Bits	Equate(s)	Description
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. The GPU must not be stopped by an external processor writing directly to the G_CTRL register. Only the GPU should turn off the GPU. If one processor wants to shut down another one, the best way is to ask them to do it themselves. For example, place a special code into a semaphore and then cause an interrupt for the processor you want to shut down. The interrupt handler would see the semaphore and shut down the processor itself.
1	CPUINT	Writing a 1 to this bit causes 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	FORCEINT0	Writing a 1 to this bit causes a GPU interrupt type 0. 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, SINGLE_STOP, indicates whether the GPU has actually stopped, and should be polled before issuing a further single step command. A one means the GPU is awaiting a SINGLE_GO 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	unused	Write zero.
6-10	G_CPULAT G_JERLAT G_PITLAT G_OPLAT	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. The meaning of these

	G_BLITLAT	bits are: 0 CPU Interrupt 1 Jerry Interrupt 2 Timing Generator 3 Object Processor 4 Blitter
11	BUS_HOG	**This bit should not be set in the Jaguar Console, always write zero.**
12-15	VERSION	These bits allow the GPU version code to be read. Current version codes are: 1 Pre-production test silicon 2 First production release 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.

G_HIDATA High Data Register *F02118* *RW*

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.

G_REMAIN Divide unit Remainder *F0211C* *RO*

This 32-bit register contains a value from which the remainder after a division may be calculated. Refer to the section on the Divide Unit.

G_DIVCTRL Divide Unit Control *F0211C* *WO*

Bits	Equate(s)	Description
0	DIV_OFFSET	If this bit is set, then the divide unit performs division of unsigned 16.16 bit numbers, otherwise 32-bit unsigned integer division is performed.

Blitter

This section describes the Jaguar Blitter.

What is the Blitter?

Blitter is an abbreviation for *bit block processor*. Its 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.

Unaligned blits in 2 bits per pixel mode are unreliable, use 1 bit per pixel blits instead.

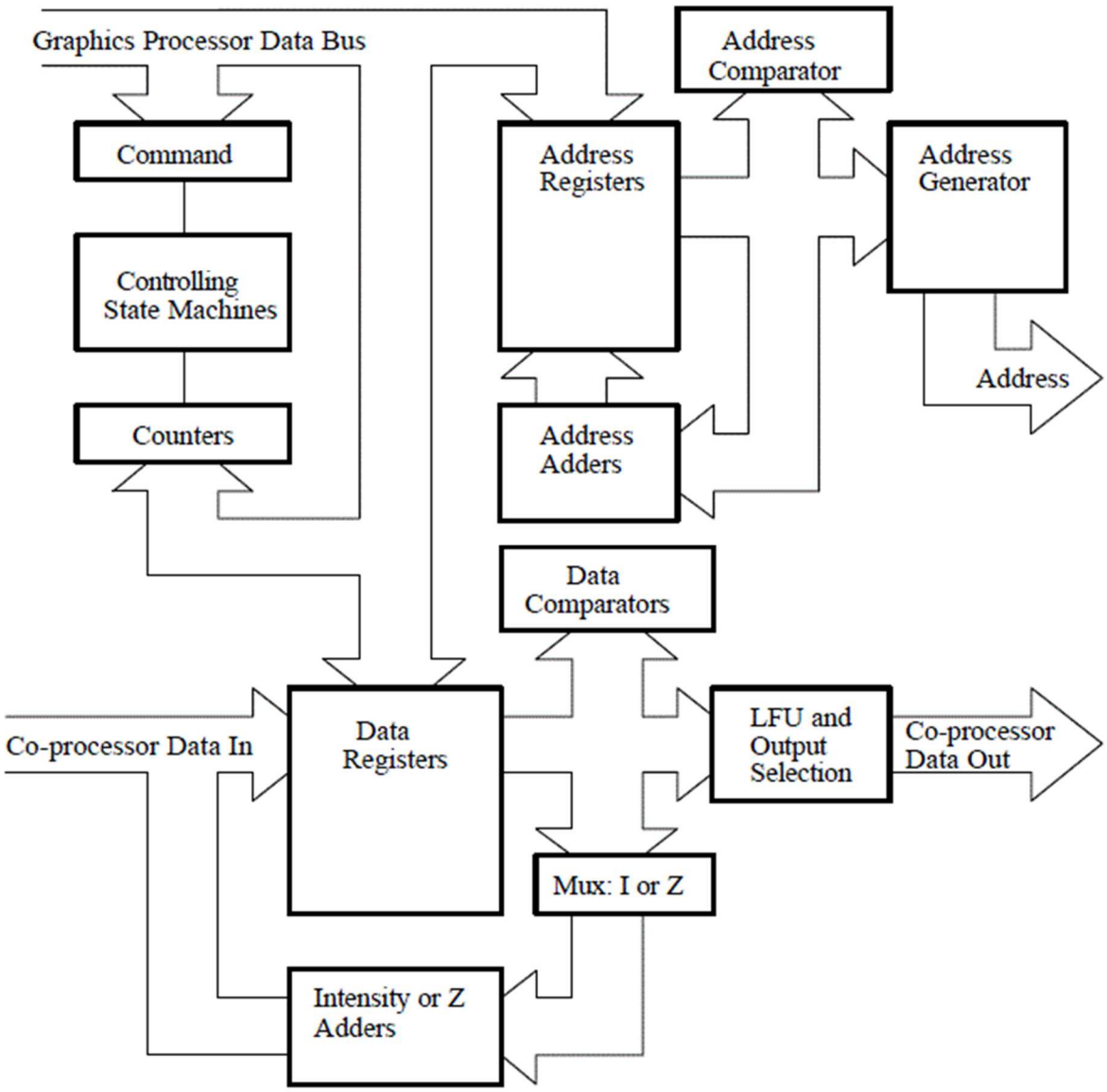
The *tour de force* of the Blitter is its ability to generate Gouraud shaded polygons, using Z-buffering, in sixteen bit pixel mode. A lot of the logic in the Blitter is devoted to its ability to create these pixels four at a time, and to write them at a rate limited only by the bus bandwidth, using 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.

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 on how to use the Blitter, use the examples further on.
The Blitter architecture is summarized in the Figure below:



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:

A1_BASE	F02200	A1 base address
A1_FLAGS	F02204	A1 control flags
A1_CLIP	F02208	A1 clipping size
A1_PIXEL	F0220C	A1 pixel pointer
A1_STEP	F02210	A1 step integer part
A1_FSTEP	F02214	A1 step fractional part
A1_FPIXEL	F02218	A1 pixel pointer fraction
A1_INC	F0221C	A1 increment integer part
A1_FINC	F02220	A1 increment fractional part
A2_BASE	F02224	A2 base address
A2_FLAGS	F02228	A2 control flags
A2_MASK	F0222C	A2 address mask
A2_PIXEL	F02230	A2 pixel pointer
A2_STEP	F02234	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 pixel 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, but only values from 0-32767 are valid in the Blitter generally.

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 described 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 the 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, the outer loop adding the step value. Thus the inner loop length is the block width, and the outer loop length is 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 as source texture or pattern over a large destination area, e.g. filling a wall with a repeated brick pattern.

A1 supports address updates based on a Digital Differential Analyser. This technique produces successive addresses 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 also has integer and fractional parts.

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 be one, two, four, eight sixteen or thirty-two bits wide, and are always stored in a packed manner.

Data registers are:

B_SRCD	F02240	Source data, or computed intensity fractional parts
B_DSTD	F02248	Destination data
B_DSTZ	F02250	Destination Z
B_SCRZ1	F02258	Source Z1, or computed Z integer parts
B_SCRZ2	F02260	Source Z2, or computed Z fractional parts
B_PATD	F02268	Pattern data, or computed intensity integer parts
B_IINC	F02270	Intensity increment
B_ZINC	F02274	Z increment
B_STOP	F02278	Collision control

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 two 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. If two source phrases must be read before a destination phrase can be written, then the SCRENX flag must be set to ensure that enough source data is fetched for the blit to operate correctly.

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 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 for 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 is inhibited if the data already present has higher priority. This applies to sixteen bit pixel mode only.

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

Write Data

Write Data may come from:

- The pattern data register
- The logic function unit
- Computed Gouraud shaded data

The default is the LFU output. The ADDDSEL flag selects adder output, PATDSEL selects the pattern register, and GOURD select the computed data.

Write Z may come from

- 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.

 © *Stephen Moss*

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- un-signed integer Z attribute of a pixel on the screen, the destination Z, with that about to be written the source Z, 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. It normally compares the source data register with the pattern data register, but it may also compare destination data with pattern data.

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 the 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 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.

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 phase 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.

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 idol.

Address Registers

All Address registers are 32-bits unless otherwise indicated.

A1_BASE A1 Base Register *F02200* *WO*

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

A1_FLAGS A1 Flags Register *F02204* *WO*

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

Bits	Equate(s)	Name	Description
0-1	PITCH1-4	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, with one special case; i.e. a pitch of … 0 = 1 Phrase (0 phrase gaps, data phrases are contiguous) 1 = 2 Phrases (1 phrase gap, data is every other phrase) 2 = 4 Phrases (3 phrase gaps, data is every fourth phrase) 3 = 3 Phrases (2 phrase gaps, data is every third phrase) **Note:** 3 is a special case and may be especially useful for double-buffered Z-buffer displays, as it allows two phrases of pixels to each phrase of Z-buffer data – there is no need to double buffer the Z data.
2		Unused	
3-5	PIXEL1 PIXEL2 PIXEL4 PIXEL8 PIXEL16 PIXEL32	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	ZOFFS1-6	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	See Desc.	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. The following is a list of valid width equates: WID2 WID28 WID160 WID896 WID4 WID32 WID192 WID1024 WID6 WID40 WID224 WID1280

 © *Stephen Moss*

			WID8 WID48 WID256 WID1536 WID10 WID56 WID320 WID1792 WID12 WID64 WID384 WID2048 WID14 WID80 WID448 WID2560 WID16 WID96 WID512 WID3072 WID20 WID112 WID640 WID3584 WID24 WID128 WID768
15		Unused	
16-17	See Desc.	X add ctrl.	These control the update of the X pointer on each pass around the inner loop. Values are: XADDPHR (00) - Add phrase width and truncate to phrase boundary (set phase mode) XADDPIX (01) - Add pixel size, effectively add one XADD0 (10) - Add zero XADDINC (11) - Add the increment
18	See Desc.	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. YADD0 (0) - Add zero YADD1 (1) - Add one
19	XSIGNSUB	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	YSIGNSUB	Y sign	Makes the Y add one mode into Y subtract one.

The Y add control bits in the A1 and A2 address generators in the Blitter are not differentiated between correctly. The A2 Y add control bit is ignored. The A1 Y add control bit affects both address generators. However, if the Y sign bits are set in either address, the corresponding add control bit has to be set for the number to be negative.

Either do not use this function, or use it on both address generators.

A1_CLIP A1 Clipping Size *F02208* *WO*

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

The window origin (0,0) is always at the top left hand corner of the window, and so clipping is performed when the pointer values are negative, or when the pointer values are greater than or equal to these values. If the desired clip rectangle does not have its top left hand corner at the window origin, then the window base register should be modified to make it the top left hand corner of the clip rectangle.

If the A1_CLIP *x* is not on a phrase boundary, then clipping occurs on the right side even if the A1_CLIP bit is not set. This applies to the destination even if the DSTA2 bit of the B_CMD register is set.

To avoid this problem, set A1_CLIP to 0 if not clipping, and when using DSTA2 make sure the source is an even phrase width.

A1_PIXEL A1 Pixel Pointer *F0220C* *RW*

This register contains the X (low word) and Y (high word) pointers into the window, and are the location

where the next pixel will be written. They are sixteen-bit signed values. If X and Y values go out of range positively then they will advance through memory (X will wrap onto the next line, Y will go off the end of the window). Only X in the range of 0-32767 and Y values in the range of 0-4095 will produce valid addresses from the address generator, values outside this range are for clipping purposes only.

A1_STEP A1 Step Value F02210 WO

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 inner 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_FSTEP A1 Step Fraction Value F02214 WO

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 scaled or rotated image.

A1_FPIXEL A1 Pixel Pointer Fraction F02218 RW

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

A1_INC A1 Increment F0221C WO

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 two 16 bit signed integer parts of the increment, the X part is in the low word, the Y part is in the high word.

A1_FINC A1 Increment Fraction F02220 WO

This is the fractional part of the increment described above.

A2_BASE A2 Base Register F02224 WO

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

A2_FLAGS A2 Flags Register F02228 WO

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

Bits	Equate(s)	Name	Description
0-1		Pitch	As A1.
2		Unused	As A1.
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 around 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_MASK A2 Window Mask *F0222C* *WO*

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. This causes the address to wrap within a rectangular area and may be used to give fill patterns.

A2_PIXEL A2 Pixel Pointer *F02230* *RW*

This register contains the X (low word) and Y (high word) pointers into the window, and are the location where the next pixel will be written. They are sixteen-bit signed values. If X and Y values go out of range positively then they will advance through memory (X will wrap onto the next line, Y will go off the end of the window). Only X values in the range of 0-32767 and Y values in the range of 0-4095 will produce valid addresses from the address generator, values outside this range are for clipping purposes only.

A2_STEP A2 Step Value *F02234* *WO*

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 inner 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

B_CMD Command Register *F02238* *WO*

This register describes the operation of the Blitter. A write to this register indicates a 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 sources 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 the 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 SCRENZ is set an extra Z read is also performed.
3	DSTEN	Enables a destination data read as part of the 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 the inner loop operation.
5	DSTWRZ	Enables a destination Z write as part of the inner loop operation.
6	CLIP_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	-	Diagnostic use only, prevents writes 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 the 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	ZBUFF	Enables polygon Z buffer updates within the inner loop, i.e. add Z fractions to the Z fraction register (source Z2), then add with carry the Z integer part to the Z integers (source Z1).
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	Selects the sum of the source and destination data as the write data. Note that the source data is a signed offset. Leave TOPBEN and TOPNEN clear and the source data gives three signed offsets for each of the CRY fields, and the intensity value will saturate. Set TOPBEN and TOPNEN and sixteen bit

		saturating adds are performed. This can be used to lighten and darken images. This only works in 16-bit per pixel modes.
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. This can only operate in 16-bit per pixel mode. bit 0 - source **less than** destination bit 1 - source **equal to** destination bit 2 - source **greater than** destination
21-24	-	These bits control the data produced by the logic function unit. The output is the Boolean OR of the following miniterms: Bit 0 – NOT source AND NOT destination Bit 1 – NOT source AND destination Bit 2 – source AND NOT destination Bit 3 – source AND destination The following are assigned equates for combinations of the above: **LFU_CLEAR** Zeros LFU_SAD S & D LFU_NSAND !S & !D **LFU_XOR** S ^ D LFU_NSAD !S & D LFU_D D LFU_NOTS !S LFU_NSORD !S | D LFU_SAND S & !D **LFU_REPLACE** S LDU_NOTD !D LFU_SORND S | !D LFU_N_SXORD !(S ^ D) LFUSORD S | D LFU_NSORND !S | !D LFU_ONE Ones
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. When operating in pixel mode then the write does not occur unless BKGWREN is set, but in phrase mode destination data is always written then the comparator determines that the pixel should not be written.
27	DCOMPEN	Enable write inhibit on the output from the data comparator. This only applies to 8-bit and 16-bit per pixel modes. When operating in pixel mode then the write does not occur unless BKGWREN is set, but in phrase mode destination data is always written then the comparator determines that the pixel should not be written.
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.
29	BUSHI	**This bit should not be set due to a bug in the Jaguar Consol, set to 0.** ~~When set the Blitter accesses the bus at the higher of its two priorities. This allows the Blitter to access the bus at a higher priority than the object processor, and may speed up operations that involve a lot of short blits such as polygon drawing. Setting BUSHI across long blits may disturb the screen.~~
30	SRCSHADE	This bit uses the IINC register to modify the intensity of data read from the source address, and may be used to lighten or darken images. It may be used in conjunction with GOURZ, but not GOURD. The read from the source is modified, so source data should not be selected using the LFU as the write

		data. This is particularly intended for performing flat shading on texture mapped surfaces. SCRSHADE only works if the GOURZ bit is set. No actual Z-buffer data needs to be calculated or written, but GOURZ **must** be set.

B_CMD Status Register *F02238* *RO*

Bit	State	Description
0	IDLE	When set, the Blitter is completely idle and its last bus transaction is completed.
1	STOPPED	When set, the Blitter is stopped in its collision detection mode – see the collision control register below.
2	inner IDLE	Diagnostic only.
3	inner SREADX	Diagnostic only.
4	inner SZREADX	Diagnostic only.
5	inner SREAD	Diagnostic only.
6	inner SZREAD	Diagnostic only.
7	inner DREAD	Diagnostic only.
8	inner DZREAD	Diagnostic only.
9	inner DWRITE	Diagnostic only.
10	inner DZWRITE	Diagnostic only.
11	outer IDLE	Diagnostic only.
12	outer INNER	Diagnostic only.
13	outer A1FUPDATE	Diagnostic only.
14	outer A1UPDATE	Diagnostic only.
15	outer A2UPDATE	Diagnostic only.
16-31	inner count	Diagnostic only.

B_COUNT Counter Registers *F0223C* *WO*

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 directly loaded into the outer loop counter.

The counters both accept values in the range 1 to 65536 (encoded as 0 - 65535)

Data registers

All data registers are sixty-four bit unless otherwise noted.

B_SRCD Source Data Register *F02240* *WO*

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 shading intensity.

B_DSTD Destination Data Registers F02248 WO

This 64-bit register holds the destination data – which may be either read in the inner loop to allow unmodified pixels to be written back correctly when in phrase-mode, or it may be used to give background or paper colours, if it is not read.

B_DSTZ Destination Z Register F02250 WO

This 64-bit register holds the destination Z value, and may be used as the data register.

B_SRCZ1 Source Z Register 1 F02258 WO

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

B_SRCZ2 Source Z Register 2 F02260 WO

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

B_PATD Pattern Data Register F02268 WO

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

B_IINC Intensity Increment Register F02270 WO

This 32-bit register holds the integer and fractional parts of the intensity increment used for Gouraud shading. Note that the top eight bits will modify the colour value, and should therefore normally be left set to zero.

B_ZINC Z Increment Register F02274 WO

This 32-bit register holds the integer and fractional parts of the Z increment used for computed Z polygon drawing.

B_STOP Collision Control Register F02278 WO

This register allows the Blitter to be stopped when an inner loop write inhibit occurs. Blitter stop will occur in paining in pixel-by-pixel mode (X add control is 1), BKGWREN is clear, and one of BCOMPEN, DCOMPEN or ZMODE0-2 is set, along with the matching condition.

The Blitter operation may at that point be resumed or aborted.

Bit	Name	Description
0	RESUME	Writing a one to this bit when the Blitter has stopped under the above conditions will cause the Blitter to resume operations. Writing a zero has no effect.
1	ABORT	Writing a one to this bit when the Blitter has stopped under the above conditions will cause the Blitter to terminate the current operation and revert to its idle state. Writing a zero has no effect.

2	STOPEN	Set this bit to enable Blitter collision stops. Clear it to disable them.

B_I3	**Intensity Register 3**	***F0227C***	***WO***
B_I2	**Intensity Register 2**	***F02280***	***WO***
B_I1	**Intensity Register 1**	***F02284***	***WO***
B_I0	**Intensity Register 0**	***F02288***	***WO***

These four registers provide an alternate view of the computer intensity integer parts (pattern data) and computed intensity fractional parts (source data) registers. They are a convenient way of updating the intensity values for Gouraud shading. Each register is a 24 bit value (8.16 bit number), with the top eight bits unused, that modifies the corresponding fields of the computed intensity integer and fractional part registers.
Note that the colour fields in the pattern data registers are unaffected by writes to these registers.

B_Z3	**Z3 Register**	***F0228C***	***WO***
B_Z2	**Z2 Register**	***F02290***	***WO***
B_Z1	**Z1 Register**	***F02294***	***WO***
B_Z0	**Z0 Register**	***F02298***	***WO***

These registers are analogous to the intensity registers, and are for Z buffer operation. They affect the corresponding parts of the computed Z integer (source Z1) and computed Z fraction (source Z2) registers.
They are 32 bit values (16.16 bit numbers).

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 to 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 loop counter – the number represents the number of pixels, so the largest block that can be copied is 32767 pixels, where 32-bit pixels are set this is 128K. 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 flags registers.

Having set these, a command is stored in the command register, with the SCREN bit set to enable source reads, and the LFUFUNC bits set to 1100 to select source data. If the source is not phrase aligned, then the SRCENX bit must be set.

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 pixels terms, is given by the X pointer plus the width times the Y pointer; the 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.

The Blitter window width is expressed as a floating-point number. The actual value has a four-bit exponent and a three-bit mantissa, whose tip bit is implicit. This allows Blitter window widths to be any value whose binary form has a more than three significant digits followed by some number of zeros.

As an example, hear are how various window widths are encoded:

Value	Binary	Floating-point	Encoded
20	000000010100	1.01 x 2^4	010001
80	000001010000	1.01 x 2^6	011001
128	000010000000	1.00 x 2^7	011100
640	001010000000	1.01 x 2^9	100101
3584	111000000000	1.11 x 2^11	101111

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.

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 greater 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.

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 ever time the inner loop is left, so bit packed data patterns may be up to eight pixels wide.

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 enabled when A1 lies within the bounds of the source image, clipping the rotated form correctly.

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.

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.

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 a curved surface by computing the intensity at each vertex, using a vertex normal, and some suitable illumination model. The vertex intensity is linearly interpolated across the polygon edges, and the edge intensities are linearly interpreted across the polygon scan lines.

Gouraud shading is only an approximation of the curved surface, and may appear unnatural where there are large intensity changes across a single polygon. 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 values 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 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 16 bit integer and 16 bit fractional part. The intensity integer also contains the colour value, so intensity is prevented from overflowing into 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 8 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 in to colour. A similar mechanism governs Z.

Both the intensity and the Z values *saturate*. This means 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 of 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
DSTRWZ          1          Write destination Z, restoring or replacing
CLIP_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.

If Z-buffer operation is enabled and the ADDDSEL or SRCSHADE bits are set, then the data is sometimes corrupted.

To work around this, break the operation into two blits, one to do the SRCSHADE or ADDSEL into an off screen buffer, and then a second one to perform the Z-buffer operation onto the screen.

Jaguar Digital Sound Processor (Jerry)

Jerry is the companionship to Tom in the Jaguar games console. Jerry provides the following functions:

- A second RISC processor (DSP) principally intended to the sound synthesis.
- Frequency dividers for clock synthesis.
- Two programmable timers.
- Stereo PWM DAC (requires few external components).
- Synchronous interface and baud rate generator (I^2S).
- Asynchronous serial interface and baud rate generator (ComLynx).
- Joystick interface decodes.
- Six general purpose IO decodes.
- Two DMA channels (by way of DSP interrupts).

Jerry occupies a 64K by to slot in the Jaguar's address space. It appears as a 16 bit port (as does all IO). The DSP however is a 32 bit processor so all transfers to the DSP are done in pairs.

Frequency dividers

Jerry is responsible for the synthesis of three important clocks.

Chroma Clock	This is 4.43 MHz for PAL and 3.58MHz for NTSC and should have a 50% duty cycle.
Video Clock	This is a multiple of the pixel clock (which is typically between 6 MHz and 12 MHz) and must be tied to the chroma clock in order to avoid the wood grain effect on TVs.
Processor Clock	This determines the speed of the memory interface, the Graphics Processor, the Object Processor and the Digital Sound Processor. This clock is divided by two to provide a clock for an external processor.

Three registers control the logic in Jerry. The ratio between the video clock and the pixel clock is determined by TOM.

CLK1 Processor clock divider ***F10010*** ***WO***
Do NOT Modify: For Information only

This register is only used if the processor clock is generated by PLL. This ten bit register determines the frequency ratio between the processor clock oscillator input (PCLKOSC) and the processor clock divider output (PCLKDIV). In PLL clock synthesis PCLKDIV is typically locked to CHRDIV so the processor clock will be

(N + 1) * CHRDIV

where N is the value written to this register. This register is initialised to one on reset. The PCLKDIV output produces a pulse every N + 1 PCLKOSC cycles

CLK2 Video clock divider ***F10012*** ***WO***
Do NOT Modify: For Information only

This is only used if the processor clock is generated by PLL. This ten bit register determines the frequency ratio between the video clock (VCLK) and the video clock divider output (VCLKDIV). As before in PLL clock synthesis VCLKDIV is typically locked to CHRDIV so the video clock frequency will be

(N + 1) * CHRDIV

where N is the value written to this register. This register is initialised to zero in reset. The VCLKDIV output produces a pulse every N + 1 VCLK cycles.

CLK3 Chroma clock divider ***F10014*** ***WO***
Do NOT Modify: For Information only

This six bit register determines the frequency ratio between the Chroma oscillator (CHRIN, CHROUT) and the Chroma clock divider output (CHRDIV). The divider divides the Chroma oscillator frequency by N + 1 where N is the value written to this register. The CHRDIV has a 50% duty cycle. This register is initialised to 3Fh (divide by 64) on reset.

The most significant bit if this register enables the Chroma oscillator into the VCLK pin. This bit is clear on reset (output disabled).

Where PLL synthesis is used this register is typically left as reset. This provides the lowest reference

frequency for generating PCLK and VCLK.

For non-PLL synthesis the Chroma crystal is some small multiply of the Chroma carrier and this frequency is used as the video clock. This register is written with the appropriate number to generate the Chroma frequency on the CHRDIV pin and bit 15 is set to enable the crystal frequency into VCLK Pin.

Programmable Timers

Jerry contains two identical timers. Each consists of two sixteen bit dividers. The first stage (loosely called the pre-scaler) divides the processor clock by N + 1. The second stage divides this frequency M + 1, where N and M are the values written to their associated registers. It is therefore possible frequency division in the range of four to four billion.

The outputs of the second stages may be used to interrupt either the digital sound processor or the external microprocessor.
It is intended that timer one is used to generate the sample rate frequency for sound synthesis and that timer two is used to generate music tempo frequency. The timers may however be used for other purposes. It should be noted that writing to the associated registers presets the counters so they could be used to provide programmable delays. Also the registers are readable which can be used to measure time accurately. This might be used in development to help profile code or to help measure the time between joystick events.

There are four registers associated with the timers. The read addresses are different to the write addresses.

JPIT1	**Timer 1 Pre-scaler**	***F10000***	***WO***
JPIT3	**Timer 2 Pre-scaler**	***F10004***	***WO***

The pre-scalers divide the processor clock by N + 1 where N is the 16 bit value written to them. The pre-scalers are down counters which are loaded when the register is written and when they reach zero. They are readable, this is really only for chip test purposes, but they might be used by the DSP to measure short events with precision.

JPIT2	**Timer 1 Divider**	***F10002***	***WO***
JPIT4	**Timer 2 Divider**	***F10006***	***WO***

These dividers divide the output from the corresponding pre-scalers by N + 1 where N is the 16 bit value written to them. The dividers, like the pre-scalers, are down counters which are loaded when the register is written and when they reach zero.

When they reach zero they may interrupt either the DSP or CPU, these interrupts are independently maskable.

Jerry Interrupts

There are six interrupt sources which may interrupt the external microprocessor. The interrupt sources are as follows:

• External	A rising edge on the EINT[0] to Jerry may cause an interrupt.
• DSP	The DPS may generate an interrupt by writing to a port.
• Timers	Both timers may generate interrupts.
• Sync.	The synchronous serial interface can generate interrupts as described below.
• UART	The asynchronous serial interface can generate interrupts as described below.

It is likely that only one or two interrupt sources would normally be directed at the microprocessor. Some of the above are mainly of relevance to the DSP in sound synthesis. The interrupt control register enables, identifies and acknowledges CPU interrupts from the six different sources.

JINTCTRL Interrupt Control Register *F10020* *RW*

Bit	Name	Description
0	J_EXTENA	Enable External interrupts.
1	J_DSPENA	Enable DPS Interrupts.
2	J_TIM1ENA	Enable Timer 1 (sample rate) interrupts.
3	J_TIM2ENA	Enable Timer 2 (tempo) interrupts.
4	J_ASYNENA	Enable Asynchronous Serial Interface interrupts.
5	J_SYNENA	Enable Synchronous Serial Interface interrupts.
6	RESERVED	**Set to 0**
7	RESERVED	**Set to 0**
8	J_EXTCLR	Clear pending external interrupts.
9	J_DSPCLR	Clear pending DSP interrupts.
10	J_TMR1CLR	Clear pending Timer 1 (sample rate) interrupts.
11	J_TMR2CLR	Clear pending Timer 2 (tempo) interrupts.
12	J_ASYNCLR	Clear pending Asynchronous Serial Interface interrupts.
13	J_SYNCLR	Clear pending Synchronous Serial Interface interrupts.

Bits 0 – 5 enable the individual interrupt sources When read bits 0 – 5 indicate which interrupts are pending. Bits 8 to 13 clear pending interrupts from the corresponding interrupt source.

Synchronous Serial Interface

The synchronous Serial interface is controlled by seven registers. These are all within the local address space of the DSP, and so may be accessed by the DSP without any external bus overhead. Other processors may access them at theses addresses. All transfers to them should be 32-bit, but the registers themselves are only 16 bit.

SCLK Serial Clock Frequency *F1A150* *WO*

This eight bit register determines the frequency of the internally generated serial clock. The frequency is given by:

Serial Clock Frequency = System Clock Frequency / (2* (N+1))

where N is the number written to this register.

SMODE	Serial Mode	F1A154	WO

Bit	Name	Description
0	INTERNAL	When set the bit enables the serial clock and word strobe outputs.
1	RESERVED	**Set to 0**
2	WSEN	This bit enables the generation of word strobe pulses. When set Jerry produces a word strobe output which is alternately high for 16 clock cycles and low for 16 clock cycles. When cleared Jerry will not generate further high pulses. This bit is ignored when INTERNAL is cleared.
3	RISING	Enables interrupt in the rising edge of word strobe.
4	FALLING	Enables interrupts on the falling edge of word strobes.
5	EVERYWORD	Enables interrupts on the MSB of every word transmitted or received.

R_DAC	Right Transmit data (to DACs)	F1A148	WO
L_DAC	Left transmit data (to DACs)	F1A14C	WO

These two sixteen bit registers hold data to be transmitted. Note that these registers have right and left swapped on purpose.

LTXD	Left transmit data (to I^2S)	F1A148	WO
RTXD	Right transmit data (to I^2S)	F1A14C	WO

These two sixteen bit registers hold data to be transmitted.

LRXD	Left receive data (from I^2S)	F1A148	WO
RRXD	Right receive data (from I^2S)	F1A14C	WO

These two sixteen bit register hold received data.

SSTAT	Serial Status	F1A150	RO

Bit	Name	Description
0	WS	This bit reflects the state of the Word strobe pin. **Do not** use this to check for data ready, use the interrupt control register.
1	Left	

Asynchronous Serial interface (ComLynx and MIDI)

The asynchronous serial interface consists of two wires, UARTI, the receive data input and UARTO, the transmit data output. This interface is primarily designed to support ComLynx but can also be used for MIDI transmit and receive.

A prescaler register is used to allow programmable baud rates.

The data transmitter is double buffered, allowing a character to be written into the data register before the transmission of the previously written character is complete. The data receiver is also double buffered, a second character can be received on the UARTI pin before the previous character has been

read from the data register.

Data is both transmitted and received in the format shown below:

```
        ___ ___ ___ ___ ___ ___ ___ ___ ___ ________
\      / 0 \/ 1 \/ 2 \/ 3 \/ 4 \/ 5 \/ 6 \/ 7 \/   \/
 \___/\___/\___/\___/\___/\___/\___/\___/\___/
 One                                            One
 Start|------------ 8 Data bits ------------|Parity Stop
 Bit                                         Bit    Bit
```

The parity can be ODD, EVEN or none. The parity on both the output and input can be programmed to be active high or low. The polarity shown is active low.

Two classes of interrupt can be generated by the asynchronous serial interface, namely receiver or transmitter interrupts. Each of these classes can be individually enabled. The table below summarises the interrupts in each class.

Receiver Interrupts:

- Parity Error
- Framing Error
- Overrun Error
- Receive Buffer Full

Transmitter Interrupts:

- Transmit Buffer Empty

ASICLK Asynchronous Serial Interface Clock *F10034* *RW*

This sixteen bit register determines the baud rate at which the asynchronous serial interface works. The frequency generated is given by:

Clock Frequency = System Clock Frequency / (N+1)

where N is the number written to this register.

The frequency generated by this register is further divided by sixteen to give the baud rate.

ASICTRL Asynchronous Serial Control *F10032* *WO*

Bit	Name	Description
0	ODD	Writing a 1 to this bit selects odd parity.
1	PAREN	Parity enable. When parity is disabled the value of the EVEN bit is transmitted in the parity bit time.
2	TXOPOL	Transmitter output polarity. Writing a 1 to this bit causes the UARTO output to be active low.
3	RXIPOL	Receiver Input polarity. Writing a 1 to this bit makes the UARTI into an inverting input.
4	TINTEN	Enable Transmitter Interrupts. Note that the asynchronous serial interface

		bit in the Interrupt Control Register also needs to be set.
5	RINTEN	Enable Receiver Interrupts. As for TINTEN the asynchronous serial interface bit in the Interrupt Control Register also needs to be set.
6	CLRERR	Clear Error. Writing a 1 to this bit clears any parity, framing or overrun errors conditions.
14	TXBRK	Transmit Break. Setting this bit to 1 causes a break level to be transmitted on the UARTO pin. It forces the UARTO output active. This may be high or low depending on the state of the TXOPOL bit.

All unused bits are reserved and should be written 0.

ASISTAT Asynchronous Serial Status *F10032* *RO*

Bit	Name	Description
0-5		These bits reflect the state of the corresponding bits in the ASICRTL register.
7	RBF	Receive Buffer Full. When set this bit indicates that a character has been received and is available in the ASIDATA register.
8	TBE	Transmit Buffer Empty
9	PE	Parity Error. This bit indicates that a parity error occurred on a received character.
10	FE	Framing Error. A framing error is detected when a non zero character is received without a stop bit at the expected time.
11	OE	Overrun Error. An overrun error is detected when a character is received on the input before the last character was read from the ASIDATA register.
13	SERSIN	Serial Input. This bit reflects the state of the UARTI pin. Its sense can be inverted by setting the RXIPOL bit in the ASICTRL register.
14	TXBRK	Transmit Break. This bit reflects the state of the corresponding bit in the ASICTRL register.
15	ERROR	Error. This bit is the logical OR of the PE, FE and OE bits. This allows a single test for error conditions.

All unused bits are reserved and may return any value.

ASIDATA Asynchronous Serial DATA *F10030* *RW*

When this register is read it returns the last character received in bits [0..7] and zero in bits [8..15]. The act of reading this register clears the receive buffer full condition leaving the way clear for subsequent characters to be received.

When the ASIDATA is written to bits [0..7] are transmitted from the UARTO pin. Bits [8..15] are not used and should be written as zeros.

There is a bug in the Jaguar UART. If a start bit is detected at a certain phrase in the UART's divide by 16 timer, it will be shifted twice, resulting in a left shift of the data byte.

The problem may be avoided by preceding a data packet with a dummy byte where the MSB is set (e.g. $80). The receiver code should discard this dummy byte. /subsequent bytes should be aligned (i.e. 2, 3, or 4 stop bits exactly, before the next start bit).
This will result in causing the falling edge of the next start bit to miss the phrase of the UART counter which causes the problem.

If a gap is left after a byte which is more than 2 bit times long, or is not exactly aligned with the previous byte, then the dummy byte must be re-transmitted (to align the UART counter again).

Joystick Interface

Jerry has four outputs which together control four external TTL IC's to provide the Joystick Interface. There are two registers.

JOYSTICK	Joystick Register	F14000	RW

When read the joystick input buffers are enabled and the data reflects the state of the sixteen joystick inputs. Output JOYLO is asserted (active low) during the read.

When written the low eight data bits are latched into the joystick output latch. Output JOYL2 is asserted (active low) during the write. The most significant bit (15) is used to enable the joystick outputs. This bit is cleared (disabled) by reset. Output JOYL3 is the inverse of the value in but 15.

JOYBUTS	Button Register	F14002	RW

When read the button input buffer is enabled and the data reflects the state of the four button inputs. Output JOYL1 is asserted (active low) during the read.

General Purpose IO Decodes

Jerry has six general purpose IO decode outputs which are asserted (active low) in the following address ranges.

GPI00	F14800 - F14FFFh	RESERVED
GPI01	F15000 - F15FFFh	RESERVED
GPI02	F16000 - F15FFFh	RESERVED
GPI03	F17000 - F177FFh	RESERVED
GPI04	F17800 - F17BFFh	RESERVED
GPI05	F17C00 - F17FFFh	RESERVED

The term "General Purpose" is a misnomer because most of the outputs are reserved.

DSP

Introduction

The DSP is part of the Jerry chip in the Jaguar, and is a variant if the GPU within Tom. It uses a very similar instruction set and programming model, but there are certain differences. The DPS has full access to the system memory map as a bus master, and its internal memory may be accessed by other bus masters within the Jaguar system.

The DSP performs two roles within the Jaguar, its primary function is sound synthesis, however it may also be available for additional graphics processing.

Sound synthesis may be the playback of sampled sounds or algorithmic sound generation, or a mixture

of the two. As the DSP is a fast, general purpose processor it may be used for a broad range of synthesis techniques.
It contains several optimisations for sound processing when compared to the GPU, in particular higher precision multiply / accumulate operations, circular buffer management, audio wave tables in local ROM, additional local fast RAM, and audio hardware within its internal address space.

As many sound generation techniques will not require anything like the full power of the DSP, it may also be used as an additional graphics processor. It has full access to the entire systems address space, although its bus bandwidth is lower as it has a 16-bit interface to external memory. It might well be used with sound synthesis occurring under an interrupt at sample rate, with the underlying code performing something like matrix multiplies for 3D object rotation.

This section assumes an understanding of the GPU, and outlines the differences between the GPU and the DSP.

Programming the DSP

Refer to the “Programming the Graphics Processor*” section in the GPU description.*

Design Philosophy

Refer to the “Design Philosophy*” section in the GPU description.*

Pipe-Lining

Refer to the “Pipe-Lining*” section in the GPU description.*

Memory Map

Refer to the “Memory Interface*” section in the GPU description for a discussion of the basics of the DSP memory interface.*

The DSP has 8K bytes of local fast RAM (twice as much as the GPU), and 2K bytes of wave tables to help with sound synthesis. These are laid out as follows:

F1A000 – F1A1FF	DSP control registers
F1B000 – F1CFFF	Local RAM
F1D000 – F1DFFF	Wave table ROM

Wave Table ROM

The wave table ROM contains eight 128 entry wave tables. These are signed 16-bit values, and are sign-extended to 32-bits, so that the ROM appears to occupy 1K 32-bit locations. Only the bottom 16 bits are significant.

The waves available are as follows:

F1D000	ROM_TRI	A Triangle wave
F1D200	ROM_SINE	A full wave SINE
F1D400	ROM_AMSINE	An amplitude modulated SINE wave
F1D600	ROM_12W	A sine wave and its second order harmonic
F1D800	ROM_CHIRP16	A chirp – this is a sine wave increasing in frequency
F1DA00	ROM_NTRI	A triangle wave with noise superimposed
F1DC00	ROM_DELTA	A spike
F1DE00	ROM_NOISE	White Noise

Load and Store Operations

Refer to the "Load and Store Operations*" section in the GPU description.*

Arithmetic Functions

Refer to the "Arithmetic Functions*" section in the GPU description.*

The DSP replaces the unsigned saturation functions of the GPU with two signed operations. SAT16S takes a signed 32-bit operand and saturates it to a signed 16bit value, i.e. if it is less than $FFF8000 it becomes $FFFF8000 and if it is greater than $00007FFF it becomes $00007FFF. SAT32S takes a signed 40-bit operand (see the section below entitle 'Extended Precision Multiply / Accumulates') and saturates it to a signed 32 bit values in a similar manner.

Interrupts

Refer to the "Interrupts*" section of the GPU for a general discussion of how the DSP interrupts behave.*

There are six interrupt sources within the DSP. These are allocated as follows:

#	Interrupt
5	External interrupt 1
4	External interrupt 0
3	Timer interrupt 2
2	Timer interrupt 1
1	I^2S Interface interrupt
0	CPU interrupt

The external interrupts are interrupts from additional Jaguar hardware outside the Tom and Jerry system. The Timer interrupts are from Jerry's local programmable timers, the I^2S interrupt is from the local synchronous serial interface, and the CPU interrupt is generated by any processor writing to the DSP control register.

Program Control Flow

Refer to the “Program Control Flow” *section in the GPU description*

Circular Buffer Management

As circular buffers are common in DSP algorithms, for sample-looping, FIFO’s and so on; there is hardware support for addressing circular buffers. These must be 2^n words long, and aligned to a 2^n boundary, where n is any practical value.

The support takes the form of two variants of ADDQ and SUBQ, namely ADDQMOD and SUBQMOD. These allow pointers to be updated with the value wrapping in the form of counting modulo 2n. This is controlled by the modulo register which is a mask on the result of these instructions. Where a bit is 1 in this register, the result of the ADDQMOD or SUBQMOD is unaffected by the instruction, where it is 0 the add may modify it. Normally the high bits of this register are set to one, and the low bits set to zero as appropriate.

Extended Precision Multiply / Accumulates

Refer to the “Multiply and Accumulate Instructions” *and the* “Systolic Matrix Multiplies” *section in the GPU description for an introduction to and explanation of these instructions.*

When multiply and accumulate operations are performed, using the IMULTN, IMACN and RESMAC instructions, or the MMULT instruction, the accumulated result is actually calculated as a forty-bit signed integer. The top eight bits are affectively overflow bits, after as RESMAC, they are at F1A120. However, the SAT32S instruction takes as its forty-bit input the register operand as the low thirty-two bits and the eight overflow bits of the accumulator as its top eight bits, and saturates the forty-bit signed integer to thirty-two bits; i.e. if it is less than FF80000000 to becomes FF80000000 and if it is more than 007FFFFFFF it becomes 007FFFFFFF.

The SAT32 instruction should therefore only be applied to the result of a multiply/accumulate operation, and before any further multiply/accumulate operations are performed. The SAT16S instruction operates only on its thirty-two bit register operand and takes no account of the overflow bits.

Divide Unit

Refer to the “Divide Unit” *section in the GPU description*

Register File

Refer to the “Register File” *section in the GPU description*

External CPU Access

Refer to the "External CPU Access" *section in the GPU description*

Addresses in the DSP are only available as 16-bit memory into which 32 bit transfers must be performed in the order low address then high address.

Internal Registers

D_FLAGS DSP Flags Register F1A100 RW

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

Bits	Equates	Description
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/subtract, and reflects carry out of some shift operations, but it is not defined 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	D_CPUENA D_I2SENA D_TIM1ENA D_TIM2ENA D_EXT0ENA	Interrupt enable bits for interrupts 0-4. The status of these bits is overridden by IMASK. These bits correspond to: 0 CPU 1 1^2S 2 Timer 1 3 Timer 2 4 EINT[0]
9-13	D_CPUCLR D_I2SCLR D_TIM1CLR D_TIM2CLR D_EXT0CLR	Interrupt latch clear bits for interrupts 0-4. 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	**This bit must not be set due to a bug in the Jaguar console – always write as 0.**
16	D_EXT1ENA	Interrupt enable bit for interrupt 5. Functions as bits 4-8. This is EINT[1]
17	D_EXT1CLR	Interrupt latch clear bit for interrupt 5. Functions as bits 9-13.

Values written to the D_FLAGS resister may not appear to have changed in the following two instructions due to pipe-lining effects.
Consequently, writing a value to the flag bits and making use of those flag bits in the following instruction will not work properly. If it is necessary to use flags set by a STORE instruction, then ensure that at least two other instructions lie between the STORE and the flags dependent instruction.

If it is necessary to use flags set by an indexed STORE instruction, then ensure that at least four other instructions lie between the STORE and the flags dependent instruction.

D_MTXC DSP Matrix Control Register F1A104 WO

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

Bits	Equates	Description
0-3	MATRIX3-15	Matrix width, in the range 3 to 15.
4	MATCOL	When set, this control bit makes the matrix held in memory to be accessed down one column, as opposed to along one row.

D_MTXA DSP Matrix Address Register F1A108 WO

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

Bits	Equates	Description
2-11	-	Matrix Address

D_END DSP data Organisation Register F1A10C WO

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

Bit	Equates	Description
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.
2	BIG_INST	When this bit is set the DSP does word program fetches like a big-endian processor.

D_PC DSP Program Counter F1A110 RW

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

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

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

D_CTRL DSP Control/Status Register F1A114 RW

This register governs the interface between the CPU and DSP.

Bit	Equates	Description
0	DSPGO	This bit stops and starts the DSP. The CPU or DSP may write to this register at any time. The status of this bit after a system reset may be externally configured.

		The DSP must not be stopped by an external processor writing directly to the D_CTRL register. Only the DSP should turn off the DSP. If one processor wants to shut down another one, the best way is to ask them to do it themselves. For example, place a special code into a semaphore and then cause an interrupt for the processor you want to shut down. The interrupt handler would see the semaphore and shut down the processor itself.
1	CPUINT	Writing a 1 to this bit causes the DSP 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	FORCEINT0	Writing a 1 to this bit causes a DSP interrupt type 0. 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 DSP 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, SINGLE_STOP, indicates whether the DSP has actually stopped, and should be polled before issuing a further single step command. A one means the DSP is awaiting a SINGLE_GO command.
4	SINGLE_GO	Writing a 1 to this bit advances the DPS 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	Unused	Write zero.
6-10	D_CPULAT D_I2SLAT D_TIM1LAT D_TIM2LAT D_EXT0LAT	Interrupt latches for interrupts 0-4, The status of these bits indicates 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. These bits correspond to: 0 CPU 1 I^2S 2 Timer 1 3 Timer 2 4 EINT[0]
11	BUS_HOG	**This bit must not be set in the Jaguar console, always write as 0.**
12-15	VERSION	These bits allow the DSP version code to be read. Current version codes are: 2 - First production release Future variants of the DSP 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 super set of this DSP.
16	D_EXT1LAT	Interrupt latch for interrupt 5. Has the same function for interrupt 5 as bits 6-10 have for interrupts 0-4. This is EINT[1]

D_MOD DSP Modulo instruction mask *F1A118* *WO*

This 32-bit register holds the value which governs which bits are modified by the ADDQMOD and SUBQMOD instructions. A 1 means that the bit will be unaffected, a 0 means that it may be changed.

Normally, the higher bits are set to 1 and the lower bits set to 0. This allows addresses to be readily generated for circular buffers of size 2^n bytes, where n is between 0 and 31.

D_REMAIN DSP Divide unit remainder *F1A11C* *RO*

This 32-bit register contains a value from which the remainder after a division may be calculated. Refer to the section on the divide unit.

D_DIVCTRL DSP Divide unit Control *F1A11C* *WO*

Bit	Equates	Description
0	DIV_OFFSET	If this bit is set, then the divide unit performs divisions of unsigned 16.16 bit numbers, otherwise 32-bit unsigned integer division is performed.

D_MACHI DSP Multiply & Accumulate High Bits *F1A120* *RO*

This 32-bit register allows the high bits of the accumulated result to be read. After a RESMAC instruction the results register of the RESMAC contains the bottom 32-bits of the accumulated value, and this register contains the top eight bits, which are sign extended to 32 bits.

In the DSP, certain peripheral I/O functions are mapped into the internal DSP space for higher efficiency when the DSP is controlling them. These are effectively 32-bit locations. These are the PWM DAC's and the Synchronous Serial interface.

Appendices

RISC Instruction Set

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode						reg1					reg 2				

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

The *reg2* and *reg1* fields usually hold a register number but have other meanings in some instructions.

The instruction set is as follows, where the syntax is

<Op code name> <Source> <Destination>

Note: To remain compatible with future versions of the Jaguar chipset, always clear the *reg1* field of single operand instructions and leave both fields of NOP cleared.

Flags

The description of each instruction indicates how it affects the flags. The flags are valid when the result is written. This is discussed further under “Writing fast GPU and DSP programs”.

Register Usage

The description of register usage shows where it uses a register port. Cycle 1 is the clock cycle at which the instruction is considered to be “executing” and is generally the pipe-line stage at which its register operands are read. It is the only pipe-line stage occupied by NOP. Where an instruction affects the flags, these are valid at the clock cycle when the result is written. This is discussed further under “Writing fast GPU and DSP programs”.

No	Syntax	Description
22	ABS Rn	**Absolute Value** 32-bit integer absolute value. Has the same effect as NEG if the operand is negative, otherwise does nothing. Note that this instruction does not work for value 8000000h, which is left unchanged, and with the negative flag set. Z – set if result is Zero N – cleared C – set if the operand was negative *Register Usage* Cycle 1: Destination register read Cycle 2: Destination register write
0	ADD, Rn,Rn	**Add** 32-bit two’s compliment integer add, result is destination register contents added to 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 *Register Usage* Cycle 1: Source and Destination register read Cycle 3: Destination register write
1	ADDC Rn,Rn	**Add with Carry** 32-bit two’s compliment 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 *Register Usage* Cycle 1: Source and Destination register read Cycle 3: Destination register write
2	ADDQ n,Rn	**Add with Quick Data** 32-bit two’s compliment 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 *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
63	ADDQMOD n,Rn **(DSP only)**	**Add with Quick Data using Modulo Arithmetic** 32-bit two's compliment integer add like ADDQ, except that the result bits may be unmodified data if the corresponding modulo register bits are set. This allow circular buffer management (for 2^n size buffers), where the high bits of the modulo register are set, and the low bits left clear. Z – Set if the result is zero N – set if the result is negative C – represents carry out of the adder *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
3	ADDQT n,Rn	**Add with Quick Data, Transparent** 32-bit two's compliment integer add, like ADDQ except that it is transparent to the flags, which retain their previous values. ZNC – unaffected *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
9	AND Rn,Rn	**Logical AND** 32-bit logical AND, the result is the Boolean AND of the source and destination register contents, and is written back to the destination register. *Flags* Z – set if the result is zero N – set if the result is negative C – ~~not defined~~ src.31 & dst.31 *Register Usage* Cycle 1: Source and Destination register read Cycle 3: Destination register write
15	BCLR n,Rn	**Bit Clear** 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. *Flags* Z – set if the all destination register bits are zero N – set from bit 31 of the result ~~C – not defined~~ Bit 31 of the source *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
14	BSET n,Rn	**Bit Set** 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.

		Flags Z – set if the result is zero N – set if the result is negative ~~C – not defined~~ C - (1<<n) & src.31 *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
13	BTST n,Rn	**Bit Test** Test the bit in the destination register selected by the immediate data in the source field, which is in the range 0-31. *Flags* Z – set if the selected bit is zero ~~N – not defined~~ N - (1<<n) & src.31 C – not defined C - (1<<n) & src.31 *Register Usage* Cycle 1: Destination register read Cycle 3: (flags are valid)
30	CMP Rn,Rn	**Compare** 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. *Flags* 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 subtract *Register Usage* Cycle 1: Source and Destination register read Cycle 3: Flags are valid
31	CMPQ n,Rn	**Compare with Quick Data** 32-bit compare with immediate data in the range -16 to +15. *Flags* 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 subtract *Register Usage* Cycle 1: Destination register read Cycle 3: Flags are valid
21	DIV Rn,Rn	**Unsigned Divide** The 32-bit unsigned integer dividend in the destination register is divided by the 32-bit unsigned integer divisor in the source register, yielding a 32-bit unsigned integer quotient as the result, like normal microprocessor division. The remainder is available, and division may also be performed on 16.16 bit unsigned integers. Refer to the section on arithmetic functions. *Flags* ZNC – unaffected *Register Usage*

		Cycle 1: Source and Destination register read Cycle 18: Destination register write
20	IMACN Rn,Rn	**Signed Integer Multiply/Accumulate, no Wrtie-Back** 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 user after IMULTN to give a multiply/accumulate group. *Refer to the section on Multiply and Accumulate instructions. *Flags* ZNC – unaffected Any register write directly after IMACN gets the ACC loaded. *Register Usage* Cycle 1: Source and Destination register read
17	IMULT Rn,Rn	**Signed Integer Multiply** 16-bit signed integer multiply, the 32-bit result is the signed integer product of the bottom 16-bits of the source and destination registers, and is written back to the destination register. *Flags* Z – set if the result is zero N – set if the result is negative C – ~~not defined~~ dst.b31 *Register Usage* Cycle 1: Source and Destination register read Cycle 3: Destination register write
18	IMULTN Rn,Rn	**Signed Integer Multiply, no Write-Back** Like IMULT, but the result is not written back to the 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. *Flags* Z – set if the result is zero N – set if the result is negative C – ~~not defined~~ dst.b31 *Register Usage* Cycle 1: Source and Destination register read
53	JR cc,n	**Jump Relative** Relative jump to the location given by the sum of the address of the next instruction and the immediate in data 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. *Flags* ZNC – unaffected *Register Usage* Cycle 1: Flags must be valid Neither the DSP or GPU will reliably execute '**jr**' or '**jump**' instructions unless they are in internal RAM.
52	JUMP cc,(Rn)	**Jump Absolute** Jump to a location pointed to by the source register, destination field is in the conditional code, where the bits are encoded 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 5 – 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). *Flags* ZNC – unaffected *Register Usage* Cycle 1: Flags must be valid
41	Load (Rn),Rn	**Load Long** 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. *Flags* ZNC – unaffected *Register Usage* Cycle 1: Source register read Cycle n: Destination register write (internal memory at cycle 3 or 4, external memory subject to bus latency)
43 44	LOAD (R14+n),Rn LOAD (R15+n),Rn	**Load Long, with Indexed Address** 32-bit memory read, as LOAD, except that the address 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 the normal LOAD operation due to the two-tick overhead of computing the address. *Flags* ZNC – unaffected *Register Usage* Cycle 1: R14 or R15 register read Cycle n: Destination register write (internal memory at cycle 5 or 6, external memory subject to bus latency)
58 59	LOAD (R14+Rn),Rn LOAD (R15+Rn),Rn	**Load Long, from Register with Base Offset** 32-bit memory load from the byte address given by the sum of R14 (or R15) and the source register (the address should be on a long word boundary). Otherwise like instructions 43 and 44. *Flags* ZNC – unaffected *Register Usage* Cycle 1: R14 or R15 and Source register read Cycle n: Destination register write (internal memory at cycle 5 or 6, external memory subject to bus latency)
39	LOADB (Rn),Rn	**Load Byte** 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. *Flags* ZNC – unaffected *Register Usage* Cycle 1: Source register read Cycle n: Destination register write (external memory subject to bus latency)
40	LOADW (Rn),Rn	**Load Word** 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. *Flags* ZNC – unaffected *Register Usage* Cycle 1: Source register read Cycle n: Destination register write (external memory subject to bus latency)
42	LOADP (Rn),Rn (**GPU only**)	**Load Phrase** 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 of the register. This applies to external memory only, internal memory will perform a 32-bit read. *Flags* ZNC – unaffected *Register Usage* Cycle 1: Source register read Cycle n: Destination register write (external memory subject to bus latency)
48	MIRROR Rn **(DSP Only)**	**Mirror Operand** The register is mirrored, i.e. bit 0 goes to bit 31, bit 1 goes to bit 30, bit 2 to bit 29 and so on. This is helpful for address generation in Fast Fourier Transform (FFT) operations. *Flags* Z – set if the result is zero N – set if the result is negative C – not defined *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
54	MMULT Rn,Rn	**Matrix Multiply** 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: *Flags* Z – set if the result is zero

		N – set if the result is negative C – represents carry out of the adder *Register Usage* Refer to the discussion of multiply/accumulate DSP matrix multiples only work in the lower 4K of DSP RAM. The DSP matrix register can only point to memory locations in the first 4K of DSP RAM. Only address lines 2-11 are programmable; the rest of the matrix address is hard-wired to $F1Bxxx.
34	MOVE Rn,Rn	**Move Register to Register** 32-bit register to register transfer. *Flags* ZNC – unaffected *Register Usage* Cycle 1: Source register read Cycle 2: Destination register write
51	MOVE PC,Rn	**Move Program Count to Register** 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/DSP to read its own PC. *Flags* ZNC – unaffected *Register Usage* Cycle 2: Destination register write
37	MOVEFA Rn,Rn	**Move from Alternate Register** 32-bit alternate register to register transfer, the source register lying in the other bank of 32 registers. *Flags* ZNC – unaffected *Register Usage* Cycle 1: Source register read Cycle 2: Destination register write
38	MOVEI n,Rn	**Move Immediate** 32-bit register load with next 32-bits of instruction stream. The first word in the instruction stream is the low word, the second is the high word.. *Flags* ZNC – unaffected *Register Usage* Cycle 3: Destination register write
35	MOVEQ n,Rn	**Move Quick Data** 32-bit register load with immediate value in the range 0-31. *Flags* ZNC – unaffected *Register Usage* Cycle 2: Destination register write
36	MOVETA Rn,Rn	**Move to Alternate Register** 32-bit register to alternate register transfer, the destination register

		lying in the other bank of 32 registers. *Flags* ZNC – unaffected *Register Usage* Cycle 1: Source register read Cycle 2: Destination register write
55	MTOI Rn,Rn	**Mantissa to Integer** 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 register. The most significant bit is bit 32, but it is sign extended. *Flags* Z – set if the result is zero N – set if the result is negative C – ~~not defined~~ set if result is negative *Register Usage* Cycle 1: Source register read Cycle 3: Destination register write
16	MULT Rn,Rn	**Multiply** 16-bit unsigned integer multiply, the 32-bit result is the unsigned integer product of the bottom 16-bits of both the source and destination registers, and is written back to the destination register. *Flags* Z – set if the result is zero N – set if the bit 31 of the result is one C – ~~not defined~~ set if result is not zero *Register Usage* Cycle 1: Source register read Cycle 3: Destination register write
8	NEG Rn	**Negate** 32-bit two's compliment negate, the result is the destination register contents subtracted from zero, and is written back to the destination register. Note that 80000000h cannot be negated. *Flags* Z – set if the result is zero N – set if the result is negative C – represents borrow out of the subtract *Register Usage* Cycle 1: Source register read Cycle 3: Destination register write
57	NOP	**Do Nothing** *Flags* ZNC – unaffected *Register Usage* none
56	NORMI Rn,Rn	**Normalisation Integer** Give 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. *Flags* Z – set if the result is zero N – set if the result is negative C – ~~not defined~~ set if src < $10 ?! *Register Usage* Cycle 1: Source register read Cycle 3: Destination register write
12	~~NOR~~ Rn NOT Rn	**Logical NOT** 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. *Flags* Z – set if the result is zero N – set if the result is negative C – ~~not defined~~ C = 1 unless source == 0 *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
10	Or Rn,Rn	**Logical OR** 32-bit logical OR operation, the result is the Boolean OR of the source and destination register contents, and is written back to the destination register. *Flags* Z – set if the result is zero N – set if the result is negative C – ~~not defined~~ src.31 & dst.31 *Register Usage* Cycle 1: Source and Destination register read Cycle 3: Destination register write
63	PACK Rn (**GPU only**)	**Pack CRY Pixel** Takes an unpacked pixel value and packs it into a 16-bit CRY pixel. Bits 22 to 25 are mapped into bits 12 to 15; bits 13 to 16 are mapped onto bits 8 to 11; and bits 0-7 are mapped onto bits 0-7. The *reg1* fields should be set to *zero* to differentiate this from UNPACK. See the section on Pack and Unpack. *Flags* ZNC -unaffected *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
19	RESMAC Rn	**Multiply/Accumulate Result Write** 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 *Flags* ZNC - unaffected *Register Usage* Cycle 3: Destination register write

28	ROR Rn,Rn	**Rotate Right** 32-bit rotate right by the bottom 5 bits of the source register. Can be used for ROL functions by complementing the value. *Flags* Z – set if the result is zero N – set if the result is negative C – represents bit 31 of the un-shifted data *Register Usage* Cycle 1: Source and Destination register read Cycle 3: Destination register write
29	RORQ n,Rn	**Rotate Right by Immediate Count** Immediate data version of ROR, Shift count may be in the range 1-32. *Flags* Z – set if the result is zero N – set if the result is negative C – represents bit 31 of the un-shifted data *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
32	SAT8 Rn (**GPU only**)	**Saturate to Eight bits** 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. *Flags* Z – set if the result is zero N – cleared C – ~~not defined~~ cleared *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
33	SAT16 Rn (**GPU only**)	**Saturate to Sixteen bits** Saturate the 32-bit signed integer operand value to a 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. *Flags* Z – set if the result is zero N – cleared C – ~~not defined~~ cleared *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
33	SAT16S Rn (**GPU only**)	**Saturate to Sixteen bit Signed** Saturate the 32-bit signed integer operand value to a 16-bit signed integer. If it is negative it is less than 8000h and it is set to that, if it is greater than 7FFFh it is set to that. *Flags* Z – set if the result is zero

		N – cleared C – not defined *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
62	SAT24 Rn (**GPU only**)	**Saturate to Twenty-Four bits** Saturate the 32-bit signed integer operand value to a 24-bit unsigned integer. If it is negative it is set to zero, if it is greater than 16,777,215 it is set to 16,777,215. This is particularly useful for computed intensities, to counteract the effect of rounding errors. *Flags* Z – set if the result is zero N – cleared C – ~~not defined~~ cleared *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
42	SAT32S Rn (**DSP only**)	**Saturate Multiple/Accumulate Result** Saturate the 40-bit signed integer operand value to a 32-bit signed integer. This uses the overflow bits from the multiply/accumulate operations as the top eight bits of the source value. If the accumulated value is less than 80000000h it saturates to that, if it is greater than 7FFFFFFFh it is set to that. *Flags* Z – set if the result is zero N – set if the result is negative C – not defined *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
23	SH Rn,Rn	**Shift** 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. *Flags* 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. *Register Usage* Cycle 1: Source and Destination register read Cycle 3: Destination register write
26	SHA Rn,Rn	**Shift Arithmetic** As SH but right shift is arithmetic, i.e. sign shifted in. *Flags* 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.

 © *Stephen Moss*

		Register Usage Cycle 1: Source and Destination register read Cycle 3: Destination register write
27	SHARQ n,Rn	**Shift Arithmetic Right** As SHRQ but arithmetic shift right, i.e. sign shifted in. Best mnemonic. *Flags* Z – set if the result is zero N – set if the result is negative C – represents bit 0 of the un-shifted data *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
24	SHLQ n,Rn	**Shift Left with Immediate Shift Count** 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). *Flags* Z – set if the result is zero N – set if the result is negative C – represents bit 31 of the un-shifted data *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
25	SHRQ n,Rn	**Shift Right with Immediate Shift Count** As SHQL but shift right, zero shifted in. *Flags* Z – set if the result is zero N – set if the result is negative C – represents bit 0 of the un-shifted data *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
47	STORE Rn,Rn	**Store Long** 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. *Flags* ZNC – unaffected *Register Usage* Cycle 1: Source and Destination register read
49 50	STORE Rn,(R14+n) STORE Rn,(R15+n)	**Store Long, with Indexed Address** 32-bit memory write, write as STORE, with address generation in the same manner as the equivalent LOAD instructions. *Flags* ZNC – unaffected *Register Usage* Cycle 1: R14 or R15 register read Cycle 2: Source register read
60	STORE Rn,(R14+Rn)	**Store Long, to register with Base Offset Address**

61	STORE Rn,(R15+Rn)	32-bit memory store to the byte address given by the sum of R14 (or R15) and the destination register (the address should be on a long-word boundary). Otherwise like instruction 49 and 50. *Flags* ZNC – unaffected *Register Usage* Cycle 1: R14 or R15 and Destination register read Cycle 2: Source register read
45	STOREB Rn,(Rn)	**Store Byte** 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. *Flags* ZNC – unaffected *Register Usage* Cycle 1: Source and Destination registers read
46	STOREW Rn,(Rn)	**Store Word** 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. *Flags* ZNC – unaffected *Register Usage* Cycle 1: Source and Destination register read
48	STOREP Rn,(Rn) (**GPU only**)	**Store Phrase** 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. *Flags* ZNC – unaffected *Register Usage* Cycle 1: Source and Destination register read
4	SUB Rn,Rn	**Subtract** 32-bit two's compliment integer subtract, the result is the source register contents subtracted from the destination registers contents, and is written to the destination register. The carry flag represents borrow out of the subtract, and the zero flag is set if the result is zero. *Flags* Z – set if the result is zero N – set if the result is negative C – represents borrow out of the subtract *Register Usage* Cycle 1: Source and Destination register read Cycle 3: Destination register write

5	SUBC Rn,Rn	**Subtract with Borrow** 32-bit two's compliment integer subtract with borrow in according to the carry flag, otherwise like SUB. *Flags* Z – set if the result is zero N – set if the result is negative C – represents borrow out of the subtract *Register Usage* Cycle 1: Source and Destination register read Cycle 3: Destination register write
6	SUBQ n,Rn	**Subtract with immediate Data** 32-bit two's compliment integer subtract, where the source field is immediate data in the range 1-32, otherwise like SUB. *Flags* Z – set if the result is zero N – set if the result is negative C – represents borrow out of the subtract *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
32	SUBQMOD n,Rn (**DSP only**)	**Subtract with Immediate Data** 32-bit two's compliment integer subtract like SUBQ, except that the result bit may be unmodified data if the corresponding modulo register bits are set. This allows circular buffer management (for 2^n size buffers), where the high bits of the modulo register are set, and the low bits left clear. *Flags* Z – set if the result is zero N – set if the result is negative C – represents borrow out of the subtract prior to the modulo masking *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
7	SUBQT n,Rn	**Subtract with immediate Data, Transparent** 32-bit two's compliment integer subtract, like SUBQ except that it is transparent to the flags, which retain their previous values. *Flags* ZNC - unaffected *Register Usage* Cycle 1: Destination register read Cycle 3: Destination register write
63	UNPACK Rn (**GPU only**)	**Unpack CRY Pixel** Take a packed CRY pixel value and unpacks it into a 32-bit integer. Bits 12 to 15 are mapped onto bits 22 to 25; bits 8 to 11 are mapped into bits 13 to 16; bits 0 to 7 are mapped onto bits 0-7. All other bits are set to zero. The *reg1* field should be set to one to differentiate this from PACL. See the section on Pack and Unpack. *Flags* ZNC - unaffected Z - set if result zero N - cleared C - cleared

		Register Usage Cycle 1: Destination register read Cycle 3: Destination register write
11	XOR Rn,Rn	**Logical XOR** 32-bit logical exclusive OR, the result is the Boolean XOR of the source and destination register contents, and is written back to the destination register. *Flags* Z – set if the result is zero N – set if the result is negative C – ~~not defined~~ src.31 & dst.31 *Register Usage* Cycle 1: Source and Destination register read Cycle 3: Destination register write

Writing Fast GPU and DSP Programs

To get the most out of the Atari RISC processors, it is important to avoid **wait states**. Each processor can execute one instruction per tick in ideal circumstances, but it is very easy for code to be subject to so many wait states that it can only achieve around half this figure. It will be worthwhile for programmers to tune the innermost loops of their code for maximum performance, and the rules given there should help do that. A well written program can usually achieve an instruction throughput of around two-thirds of the peak figure.

Wait states usually occur either because an instruction would otherwise use some system resources, such as a register or flag, which is not valid; or it would use a piece of hardware that is currently still active from an earlier operation, such as the external memory interface. This is because the chipset makes significant use of *pipe-lining* to improve performance.

Wait states are incurred when:

- an instruction reads a register containing the result of the previous instruction, one tick of wait is incurred until the previous operation completes.
- An instruction uses the flags from the previous instruction, one tick of wait is incurred until the previous operation completes.
- A result has to be written back and neither register operand of the instruction about to be executed matches, one tick of wait is incurred while the data is being written.
- Two values are to be written back at once, one tick of wait is incurred.
- An instruction attempts to use the result of a divide instruction before it is ready. Wait states are inserted until the divide unit completes the divide, between one and sixteen wait states can be incurred.
- A divide instruction is about to be executed and the previous one has not completed, between one and sixteen wait states can be incurred.
- An instruction read a register which is awaiting data from an incomplete memory read, this will be no more than one tick from internal memory, but can be several ticks from external memory.
- A load or store instruction is about to be executed and the memory interface has not completed the transfer from the previous ones (one internal load/store or two external load/stores can be pending without holding up instruction flow).

- After a store instruction with an indexed addressing mode (one tick).
- After a jump or jr (three ticks if executing out of internal memory).
- If the next instruction has not been read, this will when only occur executing out of external memory.
- During a matrix multiply if the CPU accesses the internal space of Tom or Jerry (whichever made the call).

The most common cause of wait-states is using a register which was altered in the previous instruction. For example, consider this code fragment:

```
1    add  r3,r0                ; add offset to X
2    shrq #1,r0                ; apply scaling factors
3    add  r0,r4                ; add to base
4    add  r5,r1                ; add offset to Y
5    shrq #1,r1                ; apply scaling factor
6    add  r1,r6                ; add to base
```

Wait states will be incurred after instructions 1, 2, 4 and 5. If the code were written like this:

```
1    add  r3,r0                ; add offset to X
2    add  r5,r1                ; add offset to Y
3    shrq #1,r0                ; apply scaling factors
4    shrq #1,r1                ; apply scaling factor
5    add  r0,r4                ; add to base
6    add  r1,r6                ; add to base
```

No wait state would occur. This is an example of *interleaving*, and this is a powerful technique for speeding up code. It is well worth the performance enhancement – 6 ticks instead of 10 in this example – so ensure that you code is laid out like this. Obviously, there is a considerable overhead in thinking this out, but for loops that are executed many times it is well worth doing.

The DSP must not do an external write unless it is proceeded by an external read that will complete before the write starts.
This problem is intermittent and could be missed by testing. Be careful in any DSP code that writes to external memory.

Example #1
```
     load    (r1),r2
     or      r10,r11
     store   r11,(r3)
```

Example #2
```
     load    (r1),r2
     or      r2,r11
     store   r11,(r3)
```

Example #2
```
     load    (r1),r2
     or      r2,r2
     or      r10,r11
     store   r11,(r3)
```

Example 1 will not work correctly but example 2 will. This is because the result of the load is required for the **or** operation to be performed. To make example 1 work, change it to example #3.

Data Organisation – Big and Little Endian

The Jaguar system is intended to be usable in either a little-endian, e.g. Intel 80x86, or big-endian, e.g. 680xx0, environment. The difference between these two systems is to do with the way in which bytes of a larger operand are stored in memory. There is potential for considerable confusion here, so this section attempts to explain the difference.

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 (less bits) 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 a long-word at the low address, a little-endian system will see the high word at the high address.

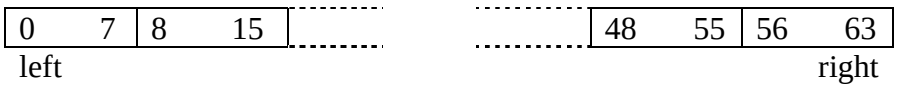
Co-Processor Bus Interface

As the co-processor bus interface is 64-bits wide, there is no problem regarding big and little endian systems, although graphic 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.

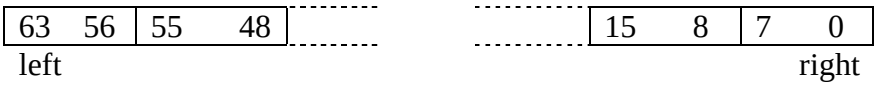
Pixel Organisation

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 includes bit 0. In byte address terms, this is 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 byte 0.



Consider an eight-bit per pixel mode:

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

 © *Stephen Moss*

(these modes refer to Blitter operation, which is describes elsewhere)

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