JAGUAR

MIDSUMMER

Technical Reference Manual

26 October, 2002
Revision 6
Atari Corp.

ATARI

Table of Contents

Introduction

Midsummer is the project name for the second generation Jaguar system (Jaguar Two).

This document is the Midsummer Technical Reference Manual - it is a definitive reference work for the programmer's view of the Midsummer chips. It is neither a hardware designer's reference work nor a guide to the Jaguar console. It is written by the hardware designers, and so is not ideal as an introduction to Midsummer or an explanation of how best to use it, but it should be used as **the** definitive reference work.

Regarding This Documentation

This document is still work in progress. It was based on the equivalent Jaguar One document and may still contain information that is no longer true. It may also have errors, omissions and things that are not clear. If so, I want to know if you find any. I apologise to my British readers for the encroachment of American spelling.

What is Midsummer?

"More strange than true. I never may believe
These antique fables, nor these fairy toys."

Act V. Scene 1.

Midsummer is based around a pair of custom chips, called Oberon and Puck, which are primarily intended to be the heart of a mega high-performance computer for games and leisure. Oberon and Puck replace Tom and Jerry from the original Jaguar system.

Oberon is the King of the fairies and Puck is Robin Goodfellow, his side-kick, from "A Midsummer Night's Dream" by William Shakespeare.

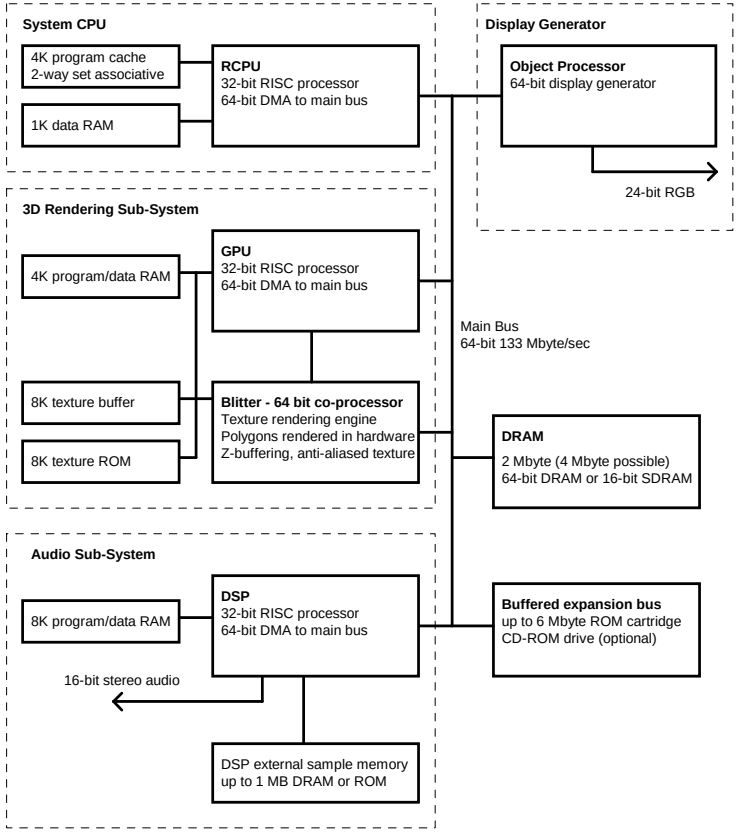
Midsummer Overview

Midsummer is an evolutionary development of Jaguar to give significant performance gains for 3D games. It offers greatly improved performance for a small increase in system cost. It is intended to be software compatible with Jaguar and so will run the existing library of games. The following areas of the system have substantially improved performance:

- polygon rendering speed
- texture mapped polygons
- computational ability
- audio synthesis

Midsummer is intended to be easy to program in a high-level language. It has an additional RISC processor, the RCPU, with an instruction cache to improve the performance of C programs.

This diagram summarises the system architecture of Midsummer. It does not show the peripheral connections, or the 68000, which is still present only for compatibility reasons and to boot the system.



The main system bus is 64-bit, and the object display processor and blitter are both 64-bit co-processors on this bus.

The RCPU, GPU and DSP are all based on the same Jaguar RISC architecture. All three processors are 32-bit RISC, executing close to one instruction per clock cycle. They are tuned for graphics and audio processing; and offer single cycle multiply operations as well as normal RISC functions.

RCPU

The RCPU is new for Midsummer, and has been specifically tuned for running C code. It is intended to act as the CPU of the system, and is the geometry engine for 3D graphics.

- 32-bit RISC processor
- 4K bytes of 2-way set-associative cache
- 1K bytes fast local data RAM
- cache line fill operations at the full 64-bit bus rate (133 MB/s)
- extended precision (16 x 32) single cycle multiplier, and fast divider
- 64-bit DMA engine to and from system DRAM at full bus rate

GPU

The GPU and Blitter are the rendering engine for 3D graphics. The GPU is very similar to the RCPU, and is coupled on a fast local 32-bit bus to the blitter. The GPU is intended to calculate blitter polygon parameters while the blitter is operating.

- 32-bit RISC processor
- 4K bytes of fast local program/data RAM
- 8K bytes of RAM either for texture buffers or for further program/data RAM
- single cycle multiplier, and fast divider

- DMA engine to and from system DRAM at full bus rate

Blitter

The Blitter is a 64-bit rendering engine. It can render triangles as a single operation, and these triangles may be any combination of Gouraud shaded, texture mapped and Z-buffered.

- 64-bit flexible rendering co-processor
- 8K bytes of texture buffer RAM (can be shared with GPU)
- 8K bytes of generic texture ROM
- texture mapping from local texture buffer or from main memory
- triangle draw as a single operation
- Z-buffering
- anti-aliased texture mapping (bi-linear interpolation)
- true-perspective texture mapping
- Gouraud shading, fog effects, colour blending and alpha-mixing all possible on texture data

Object Processor

The Object Processor is a very flexible 64-bit list processor which generates the display. It builds the display in a local line buffer from multiple bit-maps, which may be at different color resolutions. It can perform scaling, shading and fog effects on bit-map data. It can behave like a traditional sprite engine, but is far more flexible and programmable.

- 64-bit display generator
- up to 24-bits per pixel
- supports bit-maps at mixed color depths
- smooth image scaling (8.8 bit resolution)
- bit-map darkening, lightening and fog effects
- supports RGB or CRY color schemes

The CRY color scheme uses 8-bits for intensity and 8-bits for chroma, allowing smooth Gouraud shading from 16-bit pixels.

DSP

The DSP is a 32-bit RISC processor, based on the same RISC core as the GPU and RCPU. It contains a local PCM sample generator coupled to private sample memory which can generate 24 voices at 44 KHz in parallel with the flexibility and power of the DSP.

- 32-bit RISC processor
- 8K bytes of fast local program/data RAM
- single cycle multiply/accumulate, with 40-bit accumulator precision
- PCM sample generator from private memory, up to 1 Mbyte DRAM or ROM
- PCM samples can be interpolated 8-bit, 16-bit and 8-bit μ-law compressed samples
- synchronous serial interface to CD quality DAC
- 64-bit DMA engine to and from system DRAM at full bus rate

System Performance

The Midsummer system is intended to be run from a 33 MHz clock. This figure is not yet confirmed, and could possibly be higher or lower. At 33 MHz the main system bus has a sustained burst rate of 133 Mbytes / sec. Assuming 16-bit pixels, which may be RGB or CRY, the blitter and object processor can both write and read pixels at 66 Mpixels/second.

This gives the blitter a shaded, texture-mapped polygon rate of 750K polygons/second. This assumes a 10x10 triangle containing 50-pixels. Of course, realistic system performance will be lower as this assumes no overhead for computation or for display generation time.

The Jaguar RISC processors can execute one instruction per clock cycle. They therefore have a peak instruction through-put of 33 MIPs, and a realistic performance level of 25-30 MIPs. This gives a combined system performance approaching 100 MIPs, as all three processors run in parallel from local memory. They can also execute code from main DRAM, although only the RCPU is well suited to this as it has an instruction cache.

Each of the RISC processors contains a 64-bit DMA engine which can transfer data to and from their local RAM at the full bus rate of 133 MB/sec. The data stream from the CD, if present, can be DMA transferred in system DRAM. Many other small but significant improvements have been made, and some restrictions and bugs in Jaguar have been removed.

Midsummer compared to Jaguar

Midsummer is closely based on the original Jaguar system. It is intended to be software compatible with it, and is a superset of the Jaguar system. It uses newer technology to speed up the Jaguar system, address short-comings in its architecture, and to make major improvements to the specification.

Large parts of this documentation cover areas of the design that have not changed, so you should look out for the following changes:

1. There is an additional Jaguar RISC (J-RISC) processor, known as the RCPU, with a simple program cache. It is intended to perform the functionality of the CPU, acting as a geometry engine, and it is well suited to executing compiled code.
2. The blitter can now draw polygons as a single operation. These may be just filled, or any combination of Gouraud shaded, Z-buffered, and texture mapped.
3. The blitter can now draw texture maps at full bus speed — a maximum of one phrase per two clock cycles, from internal texture memory, and can also operate from external texture RAM more efficiently then before.
4. The blitter can anti-alias textures as it renders them.
5. The texture mapping and Gouraud shade modes can be combined to give shaded texture-mapped polygons, with Z-buffering as well if required. These can also be drawn at full bus speed. The shading is a multiplicative mix of the texture data and another colour, allowing lightening, darkening, distance-haze and other effects.
6. The intensity calculations are now carried out with an extended range, using an eleven bit signed integer to represent intensity, this value being clipped (saturated) only when the pixels are drawn.
7. A subset of the blitter registers are double-buffered, so that a polygon drawing engine can program the parameters for a polygon blit while the previous blit is still under way.
8. There is no need to initialise all four I and Z values (or texture pointers) for a phrase mode blit, the blitter can automatically initialise them appropriately.
9. The blitter address generators now both have clip window and mask functions. Formerly A1 had a clip window and A2 had a mask.
10. The GPU has an overflow flag which reflects signed arithmetic overflow from add or subtract operations, and also gives the state of the bit modified by bit clear and set operations before the clear or set.
11. The jump condition codes have been extended to cope with the new overflow flag, and now include all the conditions available on general purpose micro-processors, e.g. the 68000.
12. The NOP instruction has been extended, so that if its operands are not zero then it becomes an unconditional jump relative with a ten bit signed jump offset, giving an increased range.
13. Byte and word transfers to GPU RAM are now possible.
14. The J-RISC processors all contain a simple DMA transfer engine, which allows full bus rate phrase mode transfers between internal and external memory. This speeds up program loads and data set transfers.
15. The PACK and UNPACK instruction can now operate on RGB16 pixels as well as CRY.
16. The object processor can now clip at a right hand side value of less than 720 by setting the limit register.
17. The object processor can force the select bit for mixed CRY/RGB screens on a per-object basis.
18. The object processor supports line-doubling so that a TV picture can be displayed on a VGA monitor.
19. The object processor can multiplicatively mix the pixel color with a “fade to” color according to a mix control value. A new object type defines the mixer control value and the mixed color.
20. RMW objects can now have double the “strength”.
21. Scaled objects may now be controlled to a higher precision, and the horizontal remainder may now be defined.
22. Some additional extended jump condition codes allow debug functions, such as interrupt, stop and pause.

In addition, some bugs that created problems for Jaguar One programmers have been fixed:

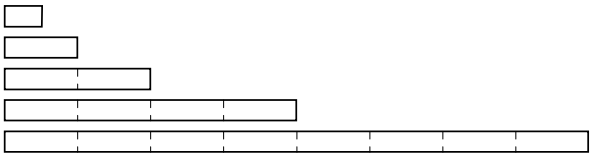
1. Score-board protection for writes is available, so that writes do not occur out of order. This is enabled by the GPU enhanced mode bit.
2. GPU code can be executed out of external RAM.
3. The blitter address flags for Y add control are now properly differentiated, there is an enable bit in the Collision control and Mode register that has to be set to fix this bug.

4. The data register of an indexed store instruction now has full score-board protection.
5. Problems related to MOVEI instructions at the beginning of a program, particularly when single stepping, have been resolved.
6. Unscaled objects are now fetched at full bus speed.
7. The pixel pre-scaler is now reset on the last line of the display, so the display need not be over-scanned to conceal it.
8. Two divides may follow each other when one uses the quotient of another.
9. The DSP external DMA interface has been completely overhauled, and will now support low and high priority transfers; as well as arbitrary load/store combinations and alignments.
10. A variety of problems related to blitter window clipping have been resolved.

Jaguar Terminology

The computer world has launched into the 64 bit era without a sensible naming convention for a 64 bit datum. Double-long-word is ridiculous and confusing. We therefore refer to 64 bits of data as a *phrase*. This is logical and short. In Jaguar the various sizes of data are named as follows:

Bits	Name
4	Nibble
8	Byte
16	Word
32	Long
64	Phrase



You may be used to calling 32 bits a long-word, or sometimes just a word. We call it a long word too sometimes, but usually just long for short! As far as I know, nobody else uses phrase or has a better name for it.

Memory

The Midsummer system has a 24 bit address bus, and so has a 16 Megabyte address range. This space contains the DRAM, cartridge ROM, boot ROM, on-chip SRAM and hardware registers. These have a variety of speeds and data bus widths, so the memory controller is flexible enough to support bus masters (processors) from 16 to 64 bits, and memory widths from 8 to 64 bits. All the processors can access all of the memory, and so there is a single system wide memory map. The processor do not usually have to be aware of the memory width, as the memory controller will take care of this for them.

The main system bus, which is connected to the DRAM and ROM as well as to both the ASICs and the 68000, can only be owned by one processor at a time. It is therefore a precious resource, and should be shared carefully so that all the processors can make use of it. This is not too big a problem, as the bus is very fast. 64 bit DRAM can be read from or written to every two clock cycles within the same DRAM page, so the transfer rate is 106 MB/sec at the Jaguar One clock speed of 26.6 MHz.

The three J-RISC processors all have private internal busses, so they can access their local memory without using the main bus, allowing them to execute in parallel with the main system.

Bus Arbitration

Bus ownership is controlled by the bus arbiter. A processor requests the bus, and if the current owner is at a lower priority level than the requesting processor, then the current owner loses the bus at the end of the current memory cycle, and the requesting processor is granted the bus. When this higher level processor has completed its transfer(s) it releases its request, and the bus is handed back to the lower priority processor. If a higher priority processor has the bus, then the requesting processor has to wait. The J-RISC processors can have pending data transfers in the background to some extent (i.e. execution continues), but ultimately they will get held up in these circumstances.

The bus is prioritised as follows:

Highest priority

1. Refresh
2. CD DMA at high priority
3. DSP at DMA priority
4. RCPU at DMA priority
5. GPU at DMA priority
6. Blitter at high priority
7. Object Processor
8. CD DMA at normal priority
9. DSP at normal priority
10. RCPU at normal priority
11. RCPU cache fetches at high priority
12. CPU under interrupt
13. GPU at normal priority
14. Blitter at normal priority
15. RCPU cache fetches at normal priority
16. Bus hold by the cache controller (no fetches occur)
17. CPU

Lowest priority

Efficient use of this bus is important to getting the best performance out of the Midsummer system. Video and Audio present real time requirements; the object processor must complete processing the object list within one video line, and audio sample and control data may need to be fetched within one sample period. The allocation of priority levels where these are selectable should be made carefully, and may require some thought and experimentation.

Dynamic RAM

The main RAM in the system is one or two banks of sixty-four bit wide dynamic RAM. Currently the system contains a single two megabyte bank of DRAM. This RAM is currently *fast page-mode* DRAM, which means that within a page transfers can be made very rapidly.

DRAM is organised internally in a rectangle of storage elements, each holding one bit. These bits are laid out in rows and columns. A row read involves transferring an entire row of bits from the main storage area into a local row buffer in the DRAM, from which the bits in the required column are selected. This row read is relatively slow, because the transistors in the main storage area are small and therefore weak. Once a row is in the local buffer, bits from different columns within it can be selected much more rapidly. In the current system, reading or writing data from a new row takes five clock cycles, while reading or writing data from the same row as the previous transfer takes two clock cycles. Each row in the current implementation contains 2048 bytes, and these are usually referred to as pages.

The DRAM is used most efficiently when most transfers are in the same page as the previous transfer. This suits video fetches, which are normally consecutive pixel reads; it suits blitter screen clears, shaded polygons, and textured polygons from internal memory; and it suits the DMA controllers in each of the J-RISC processors. It does not suit things like blitter copies which perform successive reads and writes from locations that are not in the same DRAM page. The most efficient way to move a linear block of memory is not to blit it, but to use one of the J-RISC processor DMA controllers to transfer it into local RAM, then to transfer it out again to the new location.

The system will give you the most memory bandwidth, and therefore the best performance by one measure, if DRAM transfers are mostly within the same row as the previous cycle, and are mostly sixty-four bit.

Cartridge and Boot ROM

Compared to DRAM, ROM is slow and narrow. The boot ROM is only eight bits wide, and cartridges are typically thirty-two bits. ROM is much slower then DRAM, especially compared to page mode transfers. ROM is best used as a storage medium when you can, with its contents being transferred into DRAM before use.

IO Space - on-chip registers and memory

All the registers and memory in the ASICs, as well as the joystick and other IO, are memory-mapped within the 16 Mbyte address space. They are accessed over the internal IO bus. This is a separate sixteen bit bus within the ASICs, and its speed is separately controllable and may have to be changed dynamically depending on the peripheral.

Certain IO locations within Oberon may also be written to as thirty-two bit locations, this is discussed later.

J-RISC Processor Local Space

The J-RISC processors each have a local internal thirty-two bit bus. These busses run in parallel with the main bus, and all transfers over them complete in one clock cycle. This means that the J-RISC

processors can execute code and transfer data within their internal space without using the main bus at all. This greatly increases their throughput. Transfers on these local busses may be:

- slave transfers for another processor
- DMA transfers from the local DMA controller
- operand data transfers
- program fetches

Because the local bus is thirty-two bits, IO bus transfers must always be performed in pairs in the order low address then high address. The actual read occurs on the first of the pair, the actual write on the second of the write pair; the data is just buffered in the other transfers.

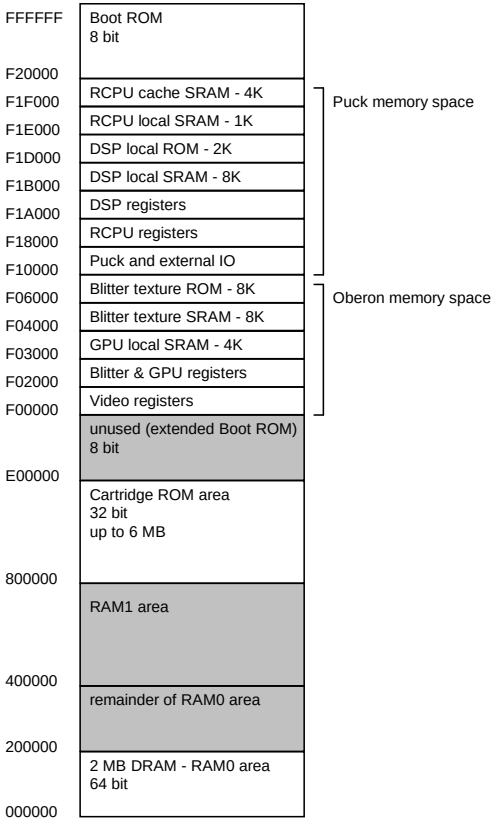
All the J-RISC processor local memory is available to every processor in the system over the IO bus, and when another processor accesses the local space of a J-RISC processor this is considered a slave transfer cycle. Heavy use of slave transfers may have a small effect on performance, but overall this impact is not very significant.

The address ranges that are subject to these restraints are:

```
F02000 - F07FFF   GPU
F18000 - F19FFF   RCPU
F1A000 - F1DFFF   DSP
F1E000 - F1FFFF   RCPU
```

Memory Map

The system memory map is normally configured as follows:



Areas marked in grey are not used in the current implementation.

Register Map

This is a complete list of every register in the Midsummer ASICs. All these registers are discussed in greater detail further on in this document. They are all 16-bit registers unless otherwise marked.

F00000	MEMCON1	RW	Memory Configuration Register One
F00002	MEMCON2	RW	Memory Configuration Register Two
F00004	HC	RW	Horizontal Count
F00006	VC	RW	Vertical Count
F00008	LIMIT	WO	Object processor clip limit
F00008	LPH	RO	Horizontal Light-pen
F0000A	LPV	RO	Vertical Light-pen
F00010	OB0-3	RO	Object Code
F00020	OLP0-1	WO	Object List Pointer
F00026	OBF	WO	Object Processor flag
F00028	VMODE	WO	Video Mode
F0002A	BORD1	WO	Border Colour (Red & Green)
F0002C	BORD2	WO	Border Colour (Blue)
F0002E	HP	WO	Horizontal Period
F00030	HBB	WO	Horizontal Blanking Begin

F00032	HBE	WO	Horizontal Blanking End
F00034	HS	WO	Horizontal Sync
F00036	HVS	WO	Horizontal Vertical Sync
F00038	HDB1	WO	Horizontal Display Begin 1
F0003A	HDB2	WO	Horizontal Display Begin 2
F0003C	HDE	WO	Horizontal Display End
F0003E	VP	WO	Vertical Period
F00040	VBB	WO	Vertical Blanking Begin
F00042	VBE	WO	Vertical Blanking End
F00044	VS	WO	Vertical Sync
F00046	VDB	WO	Vertical Display Begin
F00048	VDE	WO	Vertical Display End
F0004A	VEB	WO	Vertical Equalization Begin
F0004C	VEE	WO	Vertical Equalization End
F0004E	VI	WO	Vertical Interrupt
F00050	PIT0	WO	Programmable Interrupt Timer pre-scaler
F00052	PIT1	WO	Programmable Interrupt Timer divide
F00054	HEQ	WO	Horizontal equalization end
F00056	TEST1	RW	Diagnostic Test Register 1
F00058	BG	WO	Background Colour
F000E0	INT1	RW	CPU Interrupt Control Register
F000E2	INT2	WO	CPU Interrupt resume register
F00400	CLUT	RW	Color look-up table. 256 16-bit locations
F00800	LBUFA	RW	Line buffer A. 360 32-bit locations
F01000	LBUFB	RW	Line buffer B. 360 32-bit locations
F01800	LUBUFC	RW	Current line buffer (either A or B). 360 32-bit locations.
F02000	GPU_REGS	RW	GPU registers, sixty-four 32 bit locations
F02100	GPU_FLAGS	RW	GPU flags
F02104	GPU_MTXC	WO	GPU matrix control
F02108	GPU_MTXA	WO	GPU matrix address
F0210C	GPU_BIGEND	WO	GPU big / little endian control
F02110	GPU_PC	RW	GPU program counter
F02114	GPU_CTRL	RW	GPU operation control / status
F02118	GPU_HIDATA	RW	GPU bus interface high data
F0211C	GPU_REMAIN	RO	GPU division remainder
F0211C	GPU_DIVCTRL	WO	GPU divide control register
F02120	GPU_DMACNT	WO	GPU DMA transfer count
F02124	GPU_DMACTL	WO	GPU DMA control register
F02124	GPU_DMASTAT	RO	GPU DMA status
F02128	GPU_DMAEA	WO	GPU DMA external address
F0212C	GPU_DMAIA	WO	GPU DMA internal address
F02200	A1_BASE	WO	Blitter A1 base
F02204	A1_FLAGS	WO	Blitter A1 flags
F02208	A1_CLIP	WO	Blitter A1 window size
F0220C	A1_PIXEL	RW	Blitter A1 pointer
F02210	A1_STEP	WO	Blitter A1 step
F02214	A1_FSTEP	WO	Blitter A1 step fraction
F02218	A1_FPIXEL	RW	Blitter A1 pointer fraction
F0221C	A1_INC	WO	Blitter A1 pointer increment
F02220	A1_FINC	WO	Blitter A1 pointer increment fraction
F02224	A2_BASE	WO	Blitter A2 base
F02228	A2_FLAGS	WO	Blitter A2 flags
F0222C	A2_MASK	WO	Blitter A2 mask
F02230	A2_PIXEL	RW	Blitter A2 pointer
F02234	A2_STEP	WO	Blitter A2 step
F02238	BLIT_CMD	WO	Blitter command
F0223C	BLIT_COUNT	WO	Blitter loop counters
F02240	BLIT_SRCD	WO	Blitter source data
F02248	BLIT_DSTD	WO	Blitter destination data
F02250	BLIT_DSTZ	WO	Blitter destination Z data
F02258	BLIT_SRCZ1	WO	Blitter source Z data 1
F02260	BLIT_SRCZ2	WO	Blitter source Z data 2
F02268	BLIT_PATD	WO	Blitter pattern data
F02270	BLIT_IINC	WO	Blitter intensity increment
F02274	BLIT_ZINC	WO	Blitter Z increment
F02278	BLIT_STOP	WO	Blitter collision stop control
F0227C	BLIT_I0	WO	Blitter intensity register 0
F02280	BLIT_I1	WO	Blitter intensity register 1
F02284	BLIT_I2	WO	Blitter intensity register 2
F02288	BLIT_I3	WO	Blitter intensity register 3
F0228C	BLIT_Z0	WO	Blitter Z register 0
F02290	BLIT_Z1	WO	Blitter Z register 1
F02294	BLIT_Z2	WO	Blitter Z register 2
F02298	BLIT_Z3	WO	Blitter Z register 3
F0229C	BLIT_FINNER	WO	Fractional part of the inner counter & extended command
F022A0	BLIT_IDELTA	WO	Inner counter initial value delta
F022A4	A1_XSD	WO	A1 X step delta value
F022A8	A1_YSD	WO	A1 Y step delta value
F022AC	BLIT_ISTEP	WO	Intensity step value
F022B0	BLIT_ISD	WO	Intensity step value delta
F022B4	BLIT_ZSTEP	WO	Z step value
F022B8	BLIT_ZSD	WO	Z step value delta.
F022BC	BLIT_X0	WO	Texture X address pointer 0
F022C0	BLIT_X1	WO	Texture X address pointer 1
F022C4	BLIT_X2	WO	Texture X address pointer 2
F022C8	BLIT_X3	WO	Texture X address pointer 3
F022CC	BLIT_Y0	WO	Texture Y address pointer 0
F022D0	BLIT_Y1	WO	Texture Y address pointer 1
F022D4	BLIT_Y2	WO	Texture Y address pointer 2
F022D8	BLIT_Y3	WO	Texture Y address pointer 3
F022DC	BLIT_XINC	WO	Texture X inner loop increment
F022E0	BLIT_XSTEP	WO	Texture X outer loop step
F022E4	BLIT_XSD	WO	Texture X outer loop step delta
F022E8	BLIT_YINC	WO	Texture Y inner loop increment
F022EC	BLIT_YSTEP	WO	Texture Y outer loop step
F022F0	BLIT_YSD	WO	Texture Y outer loop step delta
F022F4	BLIT_TBASE	WO	Texture base address
F022F8	BLIT_IINCX	WO	Alternate intensity increment register
F022FC	A1_MASK	WO	A1 window address mask.
F02300	A2_CLIP	WO	A2 clipping window size
F02304	A1_X	WO	Alternate view of A1 X pixel pointer and its fraction
F02308	A1_Y	WO	Alternate view of A1 Y pixel pointer and its fraction
F0230C	A2_X	WO	Alternate view of A2 X pixel pointer
F02310	A2_Y	WO	Alternate view of A2 Y pixel pointer
F02314	A1_XSTEP	WO	Alternate view of A1 X step pixel pointer and its fraction
F02318	A1_YSTEP	WO	Alternate view of A1 Y step pixel pointer and its fraction
F0231C	BLIT_COLOR	WO	Background color and data path control
F02320	BLIT_TXTD	WO	The texture data registers
F02400	BLIT_TCLUT	WO	Blitter texture CLUT - 16 words packed into 8 longs
F03000	GPU_RAM	RW	GPU local program and data RAM base, 1024 x 32 bits
F04000	TXT_RAM	RW	Blitter texture RAM, 2048 x 32 bits
F06000	TXT_ROM	RW	Blitter texture ROM, 2048 x 32 bits
F10000	JPIT1	WO	Timer 1 Pre-scaler
F10002	JPIT2	WO	Timer 1 Divider
F10004	JPIT3	WO	Timer 2 Pre-scaler
F10006	JPIT4	WO	Timer 2 Divider
F10010	CLK1	WO	Processor clock divider
F10012	CLK2	WO	Video clock divider
F10014	CLK3	WO	Chroma clock divider
F10020	INT	RW	Interrupt Control Register
F10030	ASIDATA	RW	Asynchronous Serial Data
F10032	ASISTAT	RO	Asynchronous Serial Status

F10032	ASICTRL	WO	Asynchronous Serial Control
F10034	ASICLK	RW	Asynchronous Serial Interface Clock
F10036	JPIT1	RO	Timer 1 Pre-scaler
F10038	JPIT2	RO	Timer 1 Divider
F1003A	JPIT3	RO	Timer 2 Pre-scaler
F1003C	JPIT4	RO	Timer 2 Divider
F10040	MEMCONP1	WO	Puck Memory Configuration Register One
F10042	MEMCONP2	WO	Puck Memory Configuration Register Two
F10080	CD_CTRL	WO	CD DMA Control Register
F10080	CD_STAT	RO	CD DMA Status
F10084	CD_FLOW	WO	CD and I^2S data flow control
F10088	CD_ ACTN	WO	CD DMA Action contro;
F1008C	CD_PATH	RW	High word of pattern-recogniser long
F1008E	CD_PATL	RW	Low word of pattern-recogniser long
F10090	CD_STARTH	RW	High word of CD DMA controller start address
F10092	CD_STARTL	RW	Low word of CD DMA controller start address
F10094	CD_ENDH	RW	High word of CD DMA controller end address
F10096	CD_ENDL	RW	Low wordof CD DMA controller end address
F10098	CD_MASK	RW	Mask applied to the DMA address
F1009C	CD_CURH	RO	High word of CD DMA controller current transfer address
F1009E	CD_CURL	RO	Low word of CD DMA controller current transfer address
F1009C	CD_FAKEH	WO	High word of ‘fake’ input long
F1009E	CD_FAKEL	WO	Low word of ‘fake’ input long
F14000	JOY1	RW	Joystick register
F14002	JOY2	RW	Button register
F14800	GPIO0	RW	General purpose IO decodes
F15000	GPIO1	RW	
F16000	GPIO2	RW	
F17000	GPIO3	RW	
F17800	GPIO4	RW	
F17C00	GPIO5	RW	
F18000	RCPU_REGS	RW	RCPU registers, sixty-four 32 bit locations
F18100	RCPU_FLAGS	RW	RCPU flags
F18104	RCPU_MTXC	WO	RCPU matrix control
F18108	RCPU_MTXA	WO	RCPU matrix address
F1810C	RCPU_BIGEND	WO	RCPU big / little endian control
F18110	RCPU_PC	RW	RCPU program counter
F18114	RCPU_CTRL	RW	RCPU operation control / status
F18118	RCPU_HIDATA	RW	RCPU bus interface high data
F1811C	RCPU_REMAIN	RO	RCPU division remainder
F1811C	RCPU_DIVCTRL	WO	RCPU divide control register
F18120	RCPU_DMACT	WO	RCPU DMA transfer count
F18124	RCPU_DMACTL	WO	RCPU DMA control register
F18128	RCPU_DMAEA	WO	RCPU DMA external address
F1812C	RCPU_DMAIA	WO	RCPU DMA internal address
F18130	RCPU_CACTRL	RW	RCPU cache control register
F18134	RCPU_CAILO	WO	RCPU cache ignore range lower limit
F18138	RCPU_CAIHI	WO	RCPU cache ignore range upper limit
F1813C	RCPU_UART_C	RW	RCPU UART control register
F18140	RCPU_UART_D	RW	RCPU UART data register
F18144	RCPU_SBASE	RW	RCPU base pointer for rolling stack cache
F1A000	DSP_REGS	RW	DSP registers, sixty-four 32 bit locations
F1A100	DSP_FLAGS	RW	DSP flags
F1A104	DSP_MTXC	WO	DSP matrix control
F1A108	DSP_MTXA	WO	DSP matrix address
F1A10C	DSP_BIGEND	WO	DSP big / little endian control
F1A110	DSP_PC	RW	DSP program counter
F1A114	DSP_CTRL	RW	DSP operation control / status
F1A118	DSP_MMASK	RW	DSP modulo instruction mask
F1A11C	DSP_REMAIN	RO	DSP division remainder
F1A11C	DSP_DIVCTRL	WO	DSP divide control register
F1A120	DSP_ACCUM	RO	DSP MAC operation result high bits
F1A120	DSP_DMACNT	WO	DSP DMA transfer count
F1A124	DSP_DMACTL	WO	DSP DMA control register
F1A124	DSP_DMASTAT	RO	DSP DMA status
F1A128	DSP_DMAEA	WO	DSP DMA external address
F1A12C	DSP_DMAIA	WO	DSP DMA internal address
F1A130	PCM_LISTP	WO	DSP PCM list pointer
F1A134	PCM_CTRL	RW	DSP PCM control/status
F1A148	LTXD	WO	Left transmit data
F1A14C	RTXD	WO	Right transmit data
F1A148	LRXD	RO	Left receive data
F1A14C	RRXD	RO	Right receive data
F1A150	SCLK	WO	Serial Clock Frequency
F1A150	SSTAT	RO	Serial status
F1A154	SMODE	WO	Serial Mode
F1B000	DSP_RAM	RW	DSP data RAM, 8K bytes, byte addressable
F1E000	RCPU_DRAM	RW	RCPU data RAM, 1K bytes, byte addressable
F1E800	RCPU_TRAM	RW	RCPU cache tag RAM, 256 bytes, long addressable
F1F000	RCPU_PRAM	RW	RCPU cache data RAM, 4K bytes, long addressable

Video Generator and Object Processor

Overview

The video section has been designed to drive a PAL/NTSC TV. The display normally has a horizontal resolution from 200 up to 720 pixels, and a vertical resolution of about 220-280 lines non-interlaced or 440-560 lines interlaced. However by adopting a flexible approach to the design, the chip can be used with a range of display standards outside these values through VGA to Workstation.

Three colour resolutions are supported, 24 bit RGB, 16 bit RGB, and our own standard 16 bit CRY (Cyan, Red, Intensity). The 24 bit mode is useful for applications requiring true colour. The 16 bit modes are designed for animation. They consume less memory, and fit better into 64 bit phrases. The CRY mode is simple to shade and both 16 bit modes are more or less indistinguishable from 24 bit mode. The pixels are packed thus (in a big-endian system):

16-Bit CRY Pixel Organisation

R3	R2	R1	R0	C3	C2	C1	C0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
15															0

16-Bit RGB Pixel Organisation

R4	R3	R2	R1	R0	B4	B3	B2	B1	B0	G5	G4	G3	G2	G1	G0
15															0

24-Bit RGB Pixel Organisation

G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	low word
31															16	

								B7	B6	B5	B4	B3	B2	B1	B0	high word
15															0	

The video generator 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 is unaffected by gen-locking. There are actually two line buffers; one is displayed, while the other is prepared by the Object Processor. Each line buffer is a 360 x 32 bit RAM which is cycled at the system clock rate. The line buffer contains physical pixels, which have been expanded by the CLUT where necessary — these may be either 16 bit RGB, 16 bit CRY pixels or 24 bit RGB pixels. The line buffers may be swapped over at the start and in the middle of display lines.

The 16 bit CRY pixels at the output of the line buffer are converted to 24 bit RGB pixels using a combination of look-up tables and small multipliers.

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

The video generator uses an Object Processor, this combines the advantages of frame store and sprite based architectures. Oberon'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 supports rotation, branches, palette 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.

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

The Object Processor can also ignore data that 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.

Object Processor Performance

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

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

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

The Object Processor writes into the line buffer at one write per system clock cycle. 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 system 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 cycles. These are page mode cycles. When the DRAM row address 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 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.

Memory controller

Oberon'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 look 64 bits wide.

There are eight write strobes - one for each group of eight bits. There are three output enables corresponding to d[0-15],d[16-31] and d[32-63]. Three memory types are supported: DRAM, SRAM and ROM.

ROM 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 clock cycles 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 cycles). The memory controller decides on a cycle by cycle basis whether the next cycle can be a fast page mode cycle. Data and algorithms should be organised to minimise the number of page changes.

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

Microprocessor Interface

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 that differ. All peripherals are memory mapped; there is no separate IO space.

The width of the microprocessor is determined during reset by a pull-up / pull-down resistor. Variation in the address of the cold boot code or start-up 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. Puck uses the same microprocessor interface.

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

The following list gives the priorities of all bus masters.

Highest priority

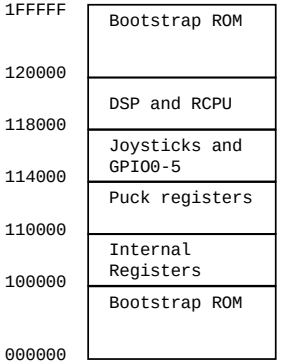
1. Refresh
2. DSP at DMA priority
3. CD DMA transfers
4. RCPU at DMA priority
5. GPU at DMA priority
6. Blitter at high priority
7. Object Processor
8. DSP at normal priority
9. RCPU at high priority
10. CPU under interrupt
11. GPU at normal priority
12. Blitter at normal priority
13. RCPU at normal priority
14. CPU

Lowest priority

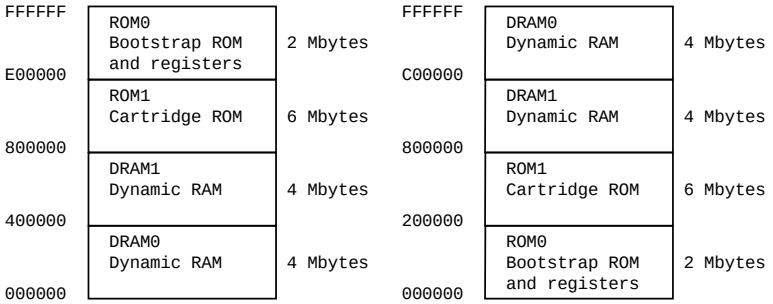
Memory Map

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

Following reset the following 2 Mbyte window, corresponding to the ROM0 area, is repeated throughout the 16 Mbyte address space until memory is configured by the microprocessor by writing to MEMCON1. After configuration, this map corresponds to the area defined as ROM0 on the maps below.



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



ROM0 is the bootstrap ROM but internal (on chip) 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 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. This is **not** recommended.

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 the graphics processor memory. The line buffers support 32 bit writes primarily in order to accelerate Blitter writes to the line buffer. The graphics processor supports 32 bit writes to accelerate program and data loads.

MEMCON1 Memory Configuration Register One F00000 RW

Bit 0	ROMHI	When set the two ROM decodes address the top 8M within the 16M window. When clear the ROM decodes address the bottom 8M. This document assumes throughout that ROMHI is set when discussing register addresses.
Bits 1,2	ROMWIDTH	Specifies the width of ROM: 0 8 bits 1 16 bits 2 32 bits 3 64 bits
Bits 3,4	ROMSPEED	Specifies the ROM cycle time: 0 10 clock cycles 1 8 clock cycles 2 6 clock cycles 3 5 clock cycles
Bits 5,6	DRAMSPEED	Specifies the DRAM Speed. The page mode cycle time is always two clock cycles. These bits determine RAS related timing as follows (the times are clock cycles):

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

Bit 7	FASTROM	Sets the ROM cycle time to two clock cycles. This is for test purposes only.
Bits 8-10	unused	Set to zero.
Bits 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
Bit 13	unused	Set to zero.
Bit 14	CPU32	Indicates that the microprocessor is 32 bits.
Bit 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 until this register is written to.

MEMCON2 Memory Configuration Register Two F00002 RW

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

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

HC Horizontal Count F00004 RW

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

VC Vertical Count F00006 RW

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

LIMIT Object processor clip limit F00008 WO

This register defines the line buffer pixel position at which line buffer writes are clipped, and the object processor will move on to the next object. The line buffer holds 720 16 bit pixels. When an object is copied into the line buffer for display any data which would extend beyond this limit is discarded and the object processor moves onto the next object. Many games display far fewer than 720 pixels horizontally so this "clipping" process will commence much sooner and improve efficiency if this register is set up appropriately.

This register is set to 720 after reset.

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 graphics processor object to pass parameters to the GPU interrupt service routine.

OLP1(OLP) Object List Pointer Low Word F00020 WO
OLP2 Object List Pointer High Word F00022 WO

This pair of 16 bit registers point 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.

You can write to this register pair with a single long transfer, but note that the word ordering is little-endian, so you will have to swap the words of the data before doing the long write.

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

Bit	Name	Description
0	VIDEN	When set enables time-base generator
1-2	MODE	Determines how the line buffer contents are translated into physical pixels. 0 16 bit CRY. Each 32 bit entry in the line buffer is treated as two 16 bit CRY pixels on successive clock cycles. Each is converted into eight bits of red, green & blue using a combination of lookup tables and multipliers. 1 24 bit RGB. Each 32 bit entry in the line buffer is treated as one physical pixel with eight bits of red, eight bits of blue, eight bits of green and eight bits unused. 2 16 bit direct. Each 32 bit entry in the line buffer is divided into two 16 bit words which are output directly onto the red and green outputs on alternate phases of the video clock. This mode is for applications requiring a dot clock in excess of 40 MHz. It is assumed that further multiplexing and colour lookup will occur outside the chip. In this mode blanking and video active are output on the two least significant bits of blue. 3 16 bit RGB. Each 32 bit entry in the line buffer is treated as two 16 bit RGB pixels. Bits [0-5] are green, bits [6-10] are blue and bits [11-15] are red.
3	GENLOCK	When set this bit enables digital genlocking. This means that external syncs will reset the internal time-base generators. On its own this mechanism does not give satisfactory genlocking because there is a jitter of up to one pixel. However this mechanism is used to quickly lock onto a new video source. An external Phase Locked Loop is required for true genlocking.
4	INCEN	Enables encrustation. When set the least significant bit of the CRY intensity is used to switch between local and external video sources using an external video multiplexer. This allows the video source to be switched on a pixel by pixel basis.
5	BINC	Selects the local border colour if encrustation is enabled.
6	CSYNC	Enables composite sync on the vertical sync output.
7	BGEN	Clears the line buffer to the colour in the background register after displaying the contents. This only has effect in CRY and RGB16 modes.
8	VARMOD	Enables variable colour resolution mode. When this bit is set the least significant bit of each word in the line buffer is used to determine the colour coding scheme of the other 15 bits. If the bit is clear the bits the word is treated as a CRY pixel. If the bit is set then bits [1-5] are green, bits [6-10] are blue and bits [11-15] are red. This mechanism allows JAGUAR to support an RGB window against a CRY background for instance.
9-11	PWIDTH	This field determines the width of pixels in video clock cycles. The width is one more than the value in this field. The video time base generator is programmed in cycles of the video 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	DBL_SCAN	In order to overlay a TV/games quality image over a VGA display. It is desirable to display the line buffer twice. This makes each pixel twice as high without the overhead of scaling and it gives the object processor two lines in which to prepare the next line. This mode is enabled by setting the DBL_SCAN bit. It should be noted that the object processor is invoked every other line. This means that the YPOS field in bit mapped objects must be incremented by two to move an object down by one pixel. Also because the object processor skips alternate lines branch objects should be an even number of lines from the start line.
13-15	unused	Write zero.

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 less significant byte of BORD1. This colour is displayed between the active portions of the screen and

blanking. It is not necessary to display a border. The border area is defined by the video time-base registers.

HP	**Horizontal Period**	**F0002E**	**WO**

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

HBB	**Horizontal Blanking Begin**	**F00030**	**WO**

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

HBE	**Horizontal Blanking End**	**F00032**	**WO**

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

HS	**Horizontal Sync**	**F00034**	**WO**

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

HVS	**Horizontal Vertical Sync**	**F00036**	**WO**

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

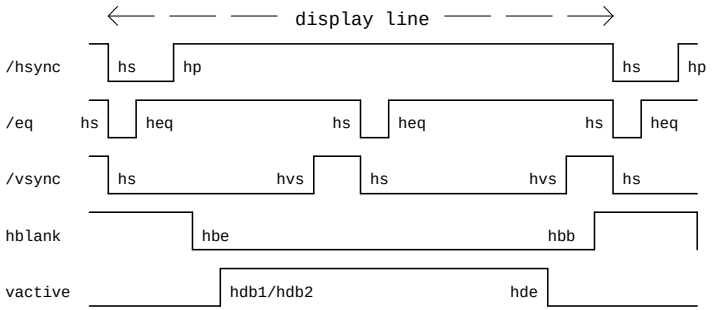
HDB1	**Horizontal Display Begin 1**	**F00038**	**WO**
HDB2	**Horizontal Display Begin 2**	**F0003A**	**WO**

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 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**

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**

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

VBE	**Vertical Blanking End**	**F00042**	**WO**

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

VS	**Vertical Sync**	**F00044**	**WO**

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

VDB	**Vertical Display Begin**	**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.

VEB	**Vertical Equalization Begin**	**F0004A**	**WO**

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

VEE	**Vertical Equalization End**	**F0004C**	**WO**

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

VI	**Vertical Interrupt**	**F0004E**	**WO**

This eleven bit register specifies a half line on which the VI interrupt is generated. This number must be odd for non-interlaced set-ups.

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

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

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

HEQ Horizontal equalization end F00054 WO

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

TEST1 Diagnostic Test Register 1 F00056 RW

This register is for chip test purposes only and must **never** be written to in normal operation. The boot-strap code may alter it to initialise the system for an application.

Bit	Description
0	enables the vertical and horizontal counters
1	starts object processing
2	disables the CRY ROMs for testing the multipliers
3	latches the vertical count
4	enables the NAND tree output onto XINTL
5	enable the Jaguar One Jerry interface
6	delay the DRAM write strobes by one half clock cycle
7	sets the timer and prescaler to run off VCLK instead of PCLK
8	enables interrupt vectors based on 40h + bits 0-4 of INT1
9	enables the current bus owner onto the bottom 4 bits of blue, for debug only, as follows: 0. CPU 1. low priority Puck 2. blitter 3. GPU 4. CPU under interrupt 5. normal priority Puck 6. object processor 7. high priority blitter 8. high priority GPU 9. high priority Puck 10. refresh Note that this is the Oberon view, further arbitration is done in Puck.

BG Background Colour F00058 WO

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

INT1 68000 Interrupt Control Register F000E0 RW

This register enables, identifies and acknowledges interrupts from the five different 68000 interrupt sources. The interrupts sources are as follows:

0	Video	This interrupt is generated by the video time-base, on a line selected by the VI register.
1	GPU	This interrupt is generated by the graphics processor writing to an internal register.
2	Object	This interrupt is generated by stop objects.
3	Timer	This interrupt is generated by the programmable timer (PIT) in OBERON.
4	Puck	This interrupt is generated by an input to Oberon and is intended for use by Puck. This is an active high edge-triggered interrupt - the first interrupt will occur on the first rising edge after it has been enabled.

Bits 0 to 4 enable the individual interrupt sources, i.e. if bit 1 is set the graphics processor interrupt is enabled. When read bits 0 to 4 indicate which interrupts are pending, i.e. if bit 3 is set there is an timer interrupt pending. Bits 8 to 12 clear pending interrupts from the corresponding interrupt source.

Note that INT2 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 the write to this port the Blitter or GPU may then restart, and no further instructions will then 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 (CRY or 16 bit RGB). 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. Addresses in the range F00600-7FE read from table B. Writing to either address range writes to both tables.

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

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

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

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

Object definitions

There are six basic object types

Bit Mapped Object

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

First Phrase

Bits	Field	Description
0-2	TYPE	Bit mapped object is type zero
3-13	YPOS	This field gives the value in the vertical counter (in half lines) for the first (top) line of the object. The vertical counter is latched when the Object Processor starts so it has the same value across the whole line. If the display is interlaced the number is even for even lines and odd for odd lines. If the display is non-interlaced the number is always even. The object will be active while the vertical counter >= YPOS and HEIGHT > 0.
14-23	HEIGHT	This field gives the number of data lines in the object. As each line is displayed the height is reduced by one for non-interlaced displays or by two for interlaced displays. (The height becomes zero if this would result in a negative value.) The new value is written back to the object.
24-42	LINK	This defines the address of the next object. These nineteen bits replace bits 3 to 21 in the register OLP. 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 data address is written back to the object.

Second Phrase

Bits	Field	Description

0-11	XPOS	This defines the X position of the first pixel to be plotted. This 12 bit field defines start positions in the range -2048 to +2047. Address 0 refers to the left-most pixel in the line buffer.
12-14	DEPTH	This defines the number of bits per pixel as follows:
		0 1 bit/pixel
		1 2 bits/pixel
		2 4 bits/pixel
		3 8 bits/pixel
		4 16 bits/pixel
		5 24 bits/pixel
15-17	PITCH	This value defines how much data, embedded in the image data, must be skipped. For instance two screens and their common Z buffer could be arranged in memory in successive phrases (in order that access to the Z buffer does not cause a page fault). The value 8 * 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 8 * (DATA + DWIDTH)
28-37	IWIDTH	This is the image width in phrases (must be non zero), and may be used for clipping.
38-44	INDEX	For images with 1 to 4 bits/pixel the top 7 to 4 bits of the index provide the most significant bits of the palette address.
45	REFLECT	Flag to draw object from right to left.
46	RMW	Flag to add object to data in line buffer. The values are then signed offsets for intensity and the two colour vectors.
47	TRANS	Flag to make logical colour zero and reserved physical colours transparent.
48	RELEASE	This bit forces the Object Processor to release the bus between data fetches. This should typically be set for low colour resolution objects because there is time for another bus master to use the bus between data fetches. For high colour resolution objects the bus should be held by the Object Processor because there is very little time between data fetches and other bus masters would probably cause DRAM page faults thereby slowing the system. External bus masters, the refresh mechanism and graphics processor DMA mechanism all have higher bus priorities and are unaffected by this bit.
49-54	FIRSTPIX	This field identifies the first pixel to be displayed. This can be used to clip an image. The significance of the bits depends on the colour resolution of the object and whether the object is scaled. The least significant bit is only significant for scaled objects where the pixels are written into the line buffer one at a time. The remaining bits define the first pair of pixels to be displayed. In 1 bit per pixel mode all five bits are significant, In 2 bits per pixel mode only the top four bits are significant. Writing zeroes to this field displays the whole phrase.
55	FORCE_LSB	The mixed CRY-RGB display mode was principally created to mix real-time RGB data from a camera say, with computer generated CRY images without the need for colour space conversion. However the blitter shading logic does not have logic to protect the least significant bit of each pixel. When this bit is set, the object processor will set or clear the LSB of every pixel within an object. If bit 55 of the first phrase of a bit mapped object is set then the LSB of each pixel is taken from bit 38 of this phrase (the least significant bit of the index). In mixed mode the LSB should be set to display the pixel as RGB or clear to display the pixel as CRY.
56	MIXER	This bit enables the object data mixer. See the discussion below under the Mixer Object.
57	HI_SCALE	Enables high precision scaling. The third phrase of the scaled bit map object uses the second higher precision form when this bit is set, as shown below.
58	DBL_RMW	When this bit is set, the strength of RMW objects is doubled. This allows a single RMW object to fade all the way to black.
59-63	unused	Write zero

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. The first 128 bits are identical to the bit mapped object except that TYPE is one. An extra phrase is appended to the object. If the HI_SCALE bit is set in the second phrase of the object description, then the object scales take a secondary form shown below. This mode also allows the horizontal remainder to be initialised.

Bits	Field	Description
0-7	HSCALE	This eight bit field contains a three bit integer part and a five bit fractional part. The number determines how many pixels are written into the line buffer for each source pixel.
8-15	VSCALE	This eight bit field contains a three bit integer part and a five bit fractional part. The number determines how many display lines are drawn for each source line. This value equals HSCALE for an object to maintain its aspect ratio.
16-23	REMAINDER	This eight bit field contains a three bit integer part and a five bit fractional part. The number determines how many display lines are left to be drawn from the current source line. After each display line is drawn this value is decremented by one. If it becomes negative then VSCALE is added to the remainder until it becomes positive. HEIGHT is decremented every time VSCALE is added to the remainder. The new REMAINDER is written back to the object.
24-63		Unused write zeroes.

This is the alternative form of the third phrase used when the HI_SCALE bit is set. This higher precision, form of the third phrase is used to define the scaling factors. The format above uses an eight bit number to define the scale. This eight bit number comprises a three bit integer and a five bit fraction. This allows scaling between 1/32 and 7. This high precision format uses a 16 bit number comprised of an eight bit integer and an eight bit fraction. This allows scaling between 1/256 and 256 but more importantly it allows more precise definition of scale.

This form of the third phrase also carries a horizontal remainder. This determines the width of the first pixel to be displayed. This can be used to get more control over scaling e.g. to ensure symmetry in scaled objects.

The bits in the third phrase of higher precision scaled objects are as follows:

Bits	Field	Description
0-15	HSCALE	This sixteen bit field contains an eight bit integer part and an eight bit fractional part. The number determines how many pixels are written into the line buffer for each source pixel.
16-31	VSCALE	This sixteen bit field contains an eight bit integer part and an eight 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.
32-47	VREMAINDER	This sixteen bit field contains an eight bit integer part and an eight bit fractional part. The number determines how many display lines are left to be drawn from the current source line. After each display line is drawn this value is decremented by one. If it becomes negative then VSCALE is added to the remainder until it becomes positive. HEIGHT is decremented every time VSCALE is added to the remainder. The new VREMAINDER is written back to the object.
48-63	HREMAINDER	This determines the width of the first pixel to be displayed, in a similar manner to VREMAINDER, so that the first pixel can be narrower than the HSCALE.

Graphics Processor Object

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

Bits	Field	Description
0-2	TYPE	GPU object is type two
3-63	DATA	These bits may be used by the GPU interrupt service routine. They are memory mapped as the object code registers OB0-3, 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.

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 whether to branch as follows: 0 Branch if YPOS == VC or YPOS == 7FF 1 Branch if YPOS > VC 2 Branch if YPOS < VC 3 Branch if Object Processor flag is set 4 Branch if on second half of 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 defined as described for the bit mapped object.
43-63	unused	

Stop Object

This object stops object processing and interrupts the host.

Bits	Field	Description
0-2	TYPE	Stop object is type four
3	STOP_INT	Enables the CPU interrupt
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.

Mixer Object

It is now possible to blend pixels in real time using the object processor. The technique can be used for distance haze or depth cueing, shadows, mist, flame, cross fading.

The mixer object type loads a "haze" colour and a fraction. This colour and fraction are applied to all subsequent objects with distance haze enabled. Bit 56 of the first phrase in bit mapped objects enables distance haze.

The mixer object is one of a class of object with type five (bits 0..2 = 5). Bits 3..7 provide 32 subtypes for future use. Subtype zero is used for the distance haze parameters. The fraction F is held in bits 8..15 and is in the range 0 to 255/256. The color H is held in bits 16..31 and will be treated as CRY or RGB depending on the display mode. When distance haze is enabled a pixel with color P is replaced with a new color given by the formula

Color = F * P + (1-F) * H

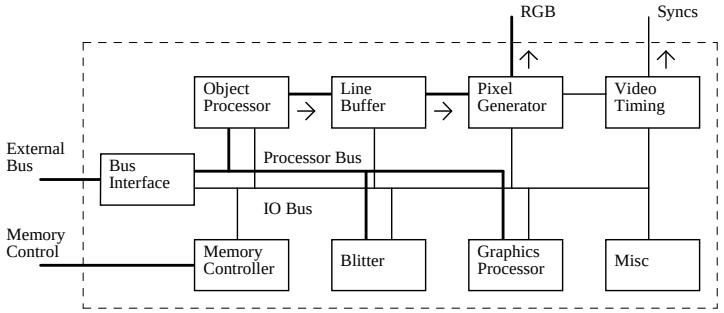
Clearly the higher the value of F the closer the color will be to the original, the smaller the closer it is to the "distance haze".

Distance haze slows down unscaled bit mapped objects to the same speed as scaled objects (1 pixel per clock cycle).

Bits	Field	Description
0-2	TYPE	Mixer object is type five.
3-7	SUBTYPE	Mixer object is sub-type zero.
8-15	MIX_FRAC	This the mixer control fraction, which controls the mix between the object data and mixer color.
16-31	MIX_COLOR	This is the mixer color, which is mixed with the object data according to the fraction, when the mixer enable bit is set.

Description of Object Processor/Pixel path

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

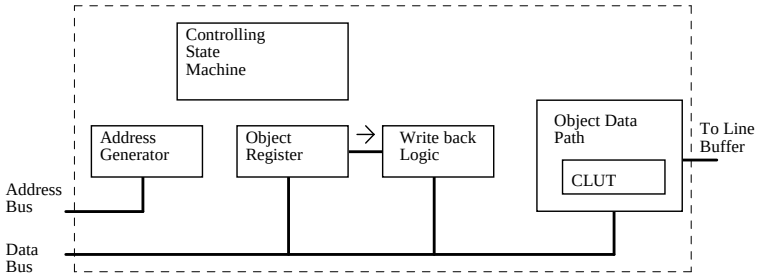


Oberon Chip Block Diagram

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 data 16 address bus used for reading and writing to internal memory and registers. The bus interface logic and memory controller allows transfers of any width (one to eight bytes) to be made to any width of external memory. The bus interface accommodates 16 and 32 bit microprocessors. The bus interface also generates a multiplexed address for dynamic RAMs. The multiplexed address is a function of memory width and number of columns. The memory controller only performs RAS cycles when the row address changes. This allows contiguous regions of memory to be accessed 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 360x32 RAM with independent write strobes for the high and low words.

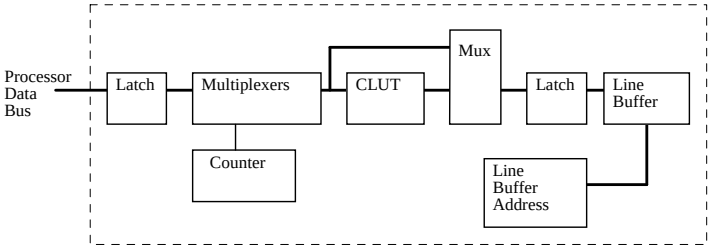
Each location in the line buffer may contain one 24 bit pixel or two 16 bit pixels.



Object Processor Block Diagram

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

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



Object Data Path

The Object Processor fetches data one phrase at a time until the image data, for that header, is exhausted or until the line buffer address (X 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.

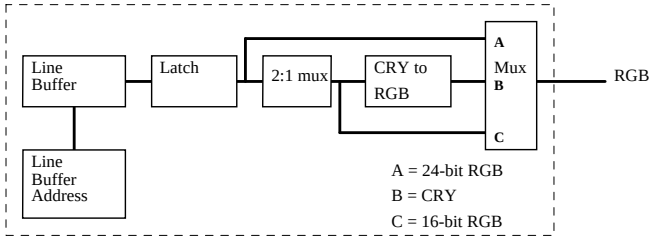
If an object is scaled the Object Processor deals with one pixel at a time not pairs. Scaling is achieved by incrementing the line buffer address independently of the counter controlling the multiplexer. For instance if the line buffer address is incremented twice as often as the counter then the image 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 by multiplexers on all the signals attached to the line buffers.

The above description is complicated by the following:

- If a pair of pixels must be written to an odd location in the line buffer they must be swapped and one pixel delayed.
- The line buffer address decrements if the object is reflected.
- The colour to be written into the line buffer can be added to the previous value instead.
- One colour may be used as transparent and is not written into the line buffer.
- The line buffers also appear as memory to the rest of the system.

The pixel data path is shown in the following diagram. All the logic in this box runs from a different clock to the previous logic, 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 more significant eight bits specify the colour and the less significant bits specify the intensity or brightness. The colour value is used as an index to three ROMs. These ROMs contain the relative amounts of red, green and blue for each colour. The outputs of the ROMs are multiplied by the brightness to get a final eight bits of red, green and blue.

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

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

A fourth mode exists to allow the system to support very high pixel rates using external multiplexers and DACs. This is called direct mode. In this mode the line buffer is read at the video clock frequency and the 2:1 multiplexer is driven by the video clock directly. The output of the 2:1 mux is connected directly to the red and green outputs of the chip. This allows 16 bit values to be output at twice the maximum video clock frequency. This provides a video bandwidth of up to 4 times the video clock rate (in bytes per second). These values should be re-synchronised, de-multiplexed and converted to analogue outside the chip. In this mode the blanking and border signals are output on the blue pins.

The above picture is slightly complicated by the following:

- The least significant bit in CRY and RGB16 modes can be sacrificed (treated as zero) and used to control an external video switch through the incrust output pin.
- In CRY and RGB16 modes a background colour may be written into the line buffer after it has been read.
- In CRY and RGB16 modes the least significant bit may be used to determine whether the mode is CRY or RGB16. This could be used to drop a decompressed RGB picture into a CRY picture without having to do a RGB to CRY conversion.

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.

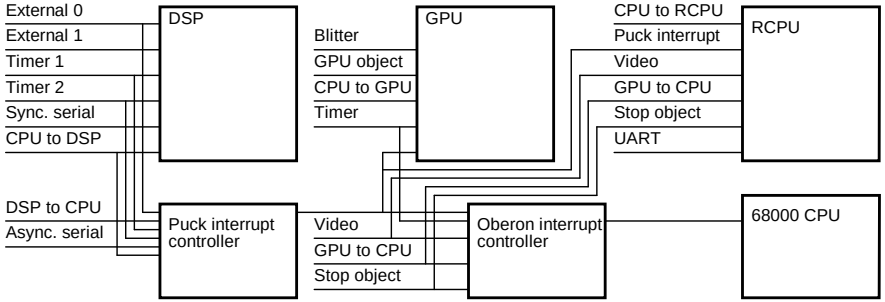
Oberon uses a counter to accumulate a count of refresh cycles. When this counter reaches eight then eight refresh cycles are done and the counter is set to zero.

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

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

Interrupts

There are a variety of interrupt sources in the system, and three micro-processors which can be interrupted: the CPU, the RCPU, the GPU and the DSP. The interrupt structure is summarised in this diagram:



The DSP and GPU both contain interrupt control logic to allow each of their interrupt inputs to be individually masked. The two interrupt controller both allow any of their interrupt inputs to be masked. The interrupt sources are:

External 0	Interrupt from the expansion connector
External 1	Interrupt from the expansion connector
Timer 1	Interrupt from Puck programmable timer 1
Timer 2	Interrupt from Puck programmable timer 2
Sync. serial	Interrupt from the synchronous serial / I^2S interface.
CPU to DSP	Interrupt to the DSP generated by a write to the DSPINT0 bit of the DSP control/status register
DSP to CPU	Interrupt to the host generated by a write to the CPUINT bit of the DSP control/status register
Async. Serial	Interrupt from the Asynchronous Serial Interface
Blitter	Interrupt generated by the blitter on blitter completion
GPU object	Interrupt generated by the object processor on processing a GPU object
CPU to GPU	Interrupt to the GPU generated by a write to the GPUINT0 bit of the GPU control/status register
Timer	Interrupt generated by Oberon's Programmable Interrupt Timer
Video	Interrupt generated by the video time-base, on a line selected by the VI register
GPU to CPU	Interrupt to the host generated by a write to the CPUINT bit of the GPU control/status register
Stop object	Interrupt generated by the object processor on processing a stop object
CPU to RCPU	Interrupt to the RCPU generated by a write to the RCPUINT0 bit of the RCPU control/status register
Puck interrupt	Composite interrupt signal from the Puck interrupt controller
UART	RCPU speicific interrupt from the asynchronous serial interface

Colour Mapping

Introduction

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

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

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

In normal operation, the Jaguar display system is based on sixteen bit pixels. Images in memory may be stored either as sixteen bit pixels, or 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 for red and blue, but this no longer allows smooth shading. There is therefore an additional scheme, known as the CRY scheme (cyan, red and intensity, see below) which still allows smooth intensity shading. This CRY scheme is now discussed in greater detail.

The CRY Colour Scheme

Gouraud Shading Requirements

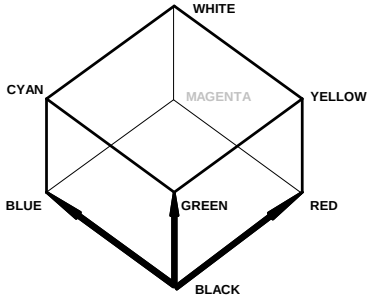
The CRY scheme was derived principally to meet the requirements of *Gouraud Shading*. This is a technique that models the appearance of a lit curved surface from a set of polygons. The problem the technique helps to overcome is that if the intensity due to a light source is calculated for each polygon 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 considered, 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 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 expressed in eight bits.

Colour Space

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



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 spread out across the colour space.
3. Colours should be able to be mixed by linearly averaging their colour values.
4. An intensity value of zero must be black.

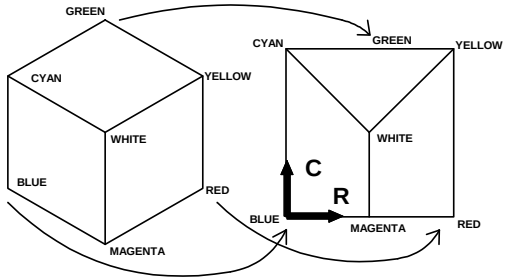
As the remaining colour space without intensity is two-dimensional, two vectors are required to represent a point in it. An *r, theta* scheme was discarded as it would not meet requirement two, and so a scheme based on two *x, y* vectors was chosen.

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



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

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

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

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

Advantages of CRY

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

Disadvantages

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

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 lookup table. It is not necessary for this to have 2^{24} entries, it turns out that taking the top 5 bits of each of the red, green and blue values (rounding where appropriate) and using a 32768 element lookup table is adequate.

Physical Implementation

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

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

The Jaguar RISC Processors

"The wildest hath not such heart as you.
Run when you will."

Act II. Scene 1.

Midsummer contains three RISC processors. These are a proprietary Atari design optimised for graphics animation and sound, and known as the J-RISC processors. A custom design is used because game console requirements are quite different to work-station requirements, which is the target application of most commercial RISC processors. The J-RISC processors will control all aspects of Midsummer operations. The three J-RISC processors are:

- the Graphics Processing Unit, GPU, which is tightly coupled to the Blitter and is the rendering engine
- the Digital Sound Processor, DSP, which creates audio and has a DAC and private sound memory
- the RISC Central Processor, RCPU, which is the system controller and geometry engine

These three processing units are based on the same design, have identical instruction throughputs, and have nearly identical instruction sets and control registers. However they are all intended for quite separate tasks. The RCPU is new to Midsummer, the GPU and DSP were both present in Jaguar One.

This section describes all three processors, with differences between them marked appropriately.

What is a Jaguar RISC Processor?

The Jaguar RISC Processor is a simple, very fast, micro-processor. It is intended for performing the functions associated with generating graphics and sound, such as three-dimensional modelling, shading, fast animation, unpacking compressed images, rendering, and audio synthesis and sample playing.

The Jaguar RISC processors correspond to the accepted notion of a RISC Processor (Reduced Instruction Set Computer). This means that:

- most instructions execute in one clock cycle
- 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 clock cycle peak throughput
- internal program and data RAM
- register score-boarding
- sixty-four thirty-two bit registers
- barrel shifter for fast shifts of any length
- one clock cycle sixteen bit multiplies and multiply/accumulates
- high speed matrix multiplication
- fast hardware divide unit
- high-speed interrupt response, including video object interrupts
- close coupling with the Blitter (in the case of the GPU)

The J-RISC processor also has many of the characteristics of a DSP (digital signal processor), in that it can perform very fast multiply and multiply/accumulate operations. These are characteristically used in graphics for 3D transforms, and in audio for digital filters. Some would consider the processor to be a RISC / DSP hybrid.

Programming the J-RISC Processor

The J-RISC processor is programmed in the same way as any other micro-processor. It has a full instruction set with a broad range of arithmetic instructions, including add, subtract, multiply and divide; Boolean instructions; logical and arithmetic shifts; 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 dependant on combinations of the zero, carry, overflow and negative flags. There are also some more specialist instructions suited to computing matrix multiplies, and some useful aids to floating-point calculations.

The J-RISC processor is a full thirty-two bit processor in that all internal data paths are thirty-two bits wide, and all arithmetic instructions (except multiply) perform thirty-two bit computations. The instructions are sixteen bits wide.

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

Design Philosophy

The J-RISC processor normally executes one instruction per clock cycle, and is therefore capable of very high instruction throughput. The RISC versus CISC debate is a complex one, and has now been largely resolved in that RISC seems to have won. The RISC approach was chosen principally because it occupies less silicon area. This leads to a processor design without micro-code, effectively the instruction set is the micro-code, and most instructions execute in one clock cycle. The advantage is that instructions are executed quicker, but the disadvantage is that some operations require more instructions to execute.

The J-RISC processor 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 well in excess of one million floating point operations per second.

The J-RISC processor was originally intended to be programmed in assembly language, rather than in a compiled language, as the tasks it is intended to perform are simple repetitive operations, best written in assembly language. It is therefore a great deal more "programmer friendly" then many RISC processors. The RCPU has some specific enhancements to make it suitable for running C code at high speed.

Pipe-Lining

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

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

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

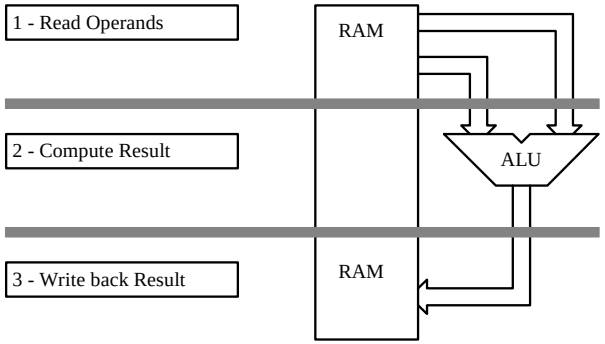
In addition to these stages, a pre-fetch unit attempts to maintain a small queue of unexecuted instructions, to keep the instruction execution unit busy.

Register Score-Boarding

The main side effect of the pipe-lined nature of its 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; then if the two instructions were just to follow each other through the pipe-line, then the second ADD would use the old value (the value from before the first ADD). Fortunately, the processor 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 *pipe-line stalls* and are a fancy RISC designer name for what are more traditionally known as wait states.

The figure shows the data flow associated with the operands of an arithmetic instruction. The thick lines correspond to a pipe-line stage, so that when an instruction is at the **Read Operands** stage, a previous arithmetic instruction may be at the **Compute Result** stage, and the potentially another one before that at the **Write Back Result** stage.



Two problems arise from this architecture:

1. The RAM used for the 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 J-RISC processor operates what is known as a *score-board t*o 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 stall (wait) if an instruction reads a tagged register. This mechanism also applies to the flags, and will stall 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).

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 - this is known as *data forwarding*. 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 clock cycle.

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.

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 on page 8.

Memory Interface

The J-RISC Processors are intended to operate in parallel with the other processing elements in the Jaguar system. In order to do this, a well-behaved program should only make occasional use of the main memory bus. The J-RISC processor therefore has some local fast thirty-two bit static memory.

This memory is intended to be used for both program and data. It can be cycled at the main 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 J-RISC processor is executing code out of internal RAM, program fetch cycles will usually occupy about half the RAM bandwidth.

To load up a program into the RAM within the GPU, the best technique is to use the local DMA engine, described later.

RCPU only: The RCPU fetches its instructions through an I-cache, which will cache instruction fetches only. This is described in more detail on page 8.

To the programmer the local RAM, local hardware registers, and external memory all appear in the same address space. The internal memory controller determines whether a transfer is local or external, and generates the appropriate cycle. The only difference to the programmer is that only 8, 16, or 32 bit transfers are possible within the local address space, whereas 8, 16, 32 or 64 bit transfers are permitted externally. Within the local address space, only 32 bit transfers may be performed to registers (and the GPU/blitter texture memory), but 8, 16 or 32 bit transfers may be performed to local memory.

The local RAM sits on an internal 32 bit bus. Also present on this bus are various hardware control registers. When a 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. This gateway also contains a simple DMA engine that allows fast transfers between internal and external memory.

This local address space is also available to external devices via the 16 bit I/O bus, see below.

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

1. CPU I/O access
2. Local DMA transfer
3. Operand data transfer
4. Instruction fetch

External View of Local Memory Space

The internal address space is accessible by any other Jaguar bus master. This is part of the system I/O space. All of the I/O space is normally viewed as 16 bit read/write memory, but because the memory is actually 32 bits wide, all transfers must be performed in word pairs, in the order low address then high address.

GPU only: by adding 8000 hex to the I/O address the internal memory space is available to external bus masters 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 than it would using the 16 bit space - for maximum transfer speed use the blitter in phrase mode, writing to the 32 bit address range.

Data Ordering Conventions

The J-RISC processor 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 instruction execution order may be little-endian or big-endian - with the exception that move immediate data is inherently little endian, i.e. its word ordering is least significant word then most significant word. (Big-end ordering sucks)

Load and Store Operations

The J-RISC processor 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.

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

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 clock cycle overhead involved in using these instructions, as the address has to computed.

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

Operand data transfers may occur at two bus priorities in external memory, either at the normal priority, or at the higher DMA priority level. This is controlled by the DMAEN flag. This does not affect program reads, which are always at normal priority. Bus priority is discussed elsewhere. This priority control bit must **not** be changed while an external memory cycle is active. Note that these occur in the background, so be very careful about changing this flag dynamically, and do not modify it in an interrupt service routine.

Note that it is quite safe to use the same register as both operands of a load (or store) operation. These operations are quite legal:

```
load    (r1),r1          ; over-write r1 with data after using it as address
load    (r14+2),r14      ; similarly, this is perfectly safe
store   r2,(r2)          ; as is this, though less useful
```

DMA Controller

The J-RISC processor has a simple DMA controller as part of the interface between its internal and external space (this interface is called the bus gateway). This allows phrase-mode transfer to be performed between internal and external memory at a rate limited solely by the external bus speed. This allows a maximum speed of one phrase transfer every two clock cycles to and from of external DRAM.

This DMA engine is intended to speed up program loads, and also to reduce processor usage of the external bus by allowing data structures to be block transferred between internal and external memory. As internal memory can be accessed using byte and word transfers, as well as longs, data structures can be easily manipulated internally.

The controller can only perform transfers either from internal memory to external memory, or *vice versa*. It cannot move things within the internal space, or perform transfers within external memory. It can only transfer a whole number of phrases, and the start address must lie on a phrase boundary in external memory, and on a long-word boundary within internal memory.

The DMA controller is very easily driven. An internal address and an external address must be written for each transfer, because they count during the transfer; some mode bits should be set to give the direction and bus priority of the transfer, unless these have not changed since the previous transfer; and the a byte length count is written, and writing this initiates the transfer. The DMA controller **must** only be started by the processor to which it belongs, unless that processor is not running.

The individual control registers are discussed further on in this document.

Arithmetic Functions

The J-RISC processor contains a powerful ALU section, which contains a thirty-two bit adder/subtracter, a thirty-two bit Boolean function unit, a sixteen bit parallel multiplier, and a thirty-two bit barrel shifter, all of which perform their respective functions in one clock cycle.

It also contains a divide unit. This performs serial division at the rate of two bits per clock cycle, on thirty-two bit unsigned operands, producing a thirty-two bit quotient. The operation of this runs in parallel with normal 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.
V	overflow	set if arithmetic overflow, i.e. carry or borrow has occurred into the sign bit of a two's complement number; also indicates the state of the bit set or cleared by a bit set or clear instruction prior to the operation.

There are also some specialist arithmetic functions:

- Saturate — the thirty-two bit operand is clipped to an unsigned value of eight, sixteen or twenty-four bits. This is helpful for dealing with accumulated rounding errors, etc. These are SAT8, SAT16 and SAT24.
- Floating Point support — functions available are an instruction which gives the amount by which a value has to be shifted to re-normalise it (NORMI), and an instruction which removes the exponent from a floating-pint value (MTOI).
- Pixel Averaging — sixteen bit RGB or CRY pixels can be unpacked so that their fields are separated to allow the values to be added together without overflow occurring from one field to the next for up to 32 pixel. See the description of PACK and UNPACK below.

DSP only: 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 16 bit value, i.e. if it is less than $FFFF8000 it becomes $FFFF8000 and if it is greater than $00007FFF it becomes $00007FFF. SAT32S takes a signed 40 bit operand (see the section below entitled 'Extended Precision Multiply / Accumulates') and saturates it to a signed 32 bit value in a similar manner.

Interrupts

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

Interrupts are allocated as follows:

GPU:

4 Blitter, indicating Blitter completion
3 Object Processor
2 Timing generator
1 DSP interrupt, the interrupt output from Puck
0 CPU interrupt

RCPU:

5 UART interrupt
4 Video interrupt
3 Object processor CPU interrupt
2 GPU to CPU interrupt
1 Puck interrupt

0 CPU interrupt

DSP:

5 External interrupt 1, from the expansion bus
4 External interrupt 0, from the expansion bus
3 Timer interrupt 1, from the Puck programmable timer 1
2 Timer interrupt 0, from the Puck programmable timer 0
1 I^2S interface interrupt, from the synchronous serial interface
0 CPU interrupt, from a write to the DSP control register

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 clock cycles of each other, the higher numbered will be serviced first. Beyond this, interrupt prioritization is under software control, as described above.

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

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

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

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

- Registers R28 and R29 may not be used by the under-lying code as they are corrupted, in addition to R30 and R31 which are always for interrupts only.
- Interrupts are re-enabled on the instruction after the jump. If they were enabled any sooner then no other interrupt service routine would be able to use R28 and R29, as they could potentially corrupt them before this service routine had completed.
- You should modify the bit set instruction shown as setting bit 11 to set the appropriate bit for the interrupt being serviced.
- If you modify this interrupt procedure to re-enable interrupts prior to the exit code, then you should change R30 to another register, as R30 is corrupted when a second interrupt occurs.

GPU only: 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.

For your information: it may interest you to know how the RISC processor enters an interrupt service routine. When an interrupt occurs, the following "hidden" instruction sequence is forced in the instruction stream at the end of the current atomic operation (one or more instructions):

```
subqt     #4,r31                      ; pre-decrement stack pointer
move      pc,r30                      ; address of interrupted code
store     r30,(r31)                   ; store return address
movei     #service_address,r30        ; pointer to ISR entry
jump      (r30)                       ; jump to ISR
nop
```

This code in not fetched from anywhere, but is directly injected into the instruction stream, so the PC read that you read while this is happening is that of the last instruction to be executed prior to the interrupt.

Atomic Operations

It is necessary for certain operations to be atomic, i.e. interrupts may not occur during these operations. Certain instruction types temporarily lock out interrupts while they complete their operation. These are:

- Immediate data moves, using the MOVEI instruction. Interrupts are locked out while the two words of immediate data are fetched.
- Matrix multiply operations, using the MMULT instruction. Interrupts are locked out until the operation has completed.
- Multiply and accumulate operations, using the IMULTN and IMACN instructions. The result register is not preserved by interrupts, and therefore any multiply/accumulate operation must consist of a sequence of IMULTN and IMACN instructions followed by a RESMAC instruction, with no intervening instructions. The IMULTN and IMACN instructions are always atomic with the succeeding instruction. See the section below on multiply / accumulate instructions.
- Jump instructions are always atomic with the instruction which succeeds them.

Sharing Hardware

The Jaguar hardware supports parallel processing, which is both a blessing and a curse. It offers much greater processing power when used well, and much greater scope for disaster when used badly.

There is nothing new about parallel processing, you might consider an interrupt and its underlying code to be parallel processes. However the Midsummer hardware supports four CPUs which can all execute simultaneously, and which can all support interrupts from a variety of sources. Most of the problems this can introduce are at a software level and are beyond the scope of this document, however there are some important warnings when it comes to sharing the Jaguar hardware.

Most of the hardware in Jaguar cannot be shared between two processes without special steps to ensure that only one can use it at a time. This is usually implemented by a *semaphore*, by which one process flags another if it can use the hardware resource. The two processes might be running on separate processors, or might be running on the same one if at least one is an interrupt.

An example of a hardware resource that cannot be shared is the blitter, where it would be disaster if two processes tried to set up a set of blitter parameters at the same time. Within the J-RISC processors, the divide remainder and mode, the high long word register, and the matrix multiplier all fall into this area. Careful attention should be paid to this if you want to share a piece of hardware.

Program Control Flow

The J-RISC processor runs through memory executing instructions unless it encounters a jump instruction, an interrupt occurs, or it stops itself. The instruction stream is 16 bit words, which are fetched into a small pre-fetch queue which requests 32 bits per fetch.

Jump Instructions

Two types of jump are supported, relative and absolute. Jump relative takes a signed five or ten bit offset, which is treated as an offset in words, and is 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. Only the five bit offset relative jumps are conditional, then ten bit offset jump relative is unconditional.

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, using the MOVE PC,Rn instruction.

The J-RISC processor can cease operation by clearing the appropriate GO bit in the local control register (described below). It may then only be restarted by an external write to this register. The GO bit must only be cleared by the processor it controls, although any processor can set it (although it may be cleared externally for debug when in single-stepping mode).

When a jump instruction is executed, and the condition is met, the following sequence occurs:

- interrupts are temporarily masked
- the next instruction is allowed to flow through the pipe-line
- the pre-fetch queue is flushed
- the new address is loaded into the program counter
- instruction fetch start from the new address to reload the pipe-line
- the temporary interrupt mask is cleared

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 one after the other (next to each other in memory), the results are not predictable, and may not be relied on.
- Do not place a MOVE PC to register instruction immediately after a jump absolute or jump relative instruction, the value read can not be relied upon.
- Do not follow an IMACN or IMULTN instruction by anything other than an IMACN instruction or a RESMAC instruction (see below).
- Do not precede an MMULT instruction by a LOAD or STORE instruction.
- Do not follow a jump instruction with an indexed store or load instruction. This may cause interrupts to behave unpredictably.

Conditional Jumps

Conditional jumps encode from a five bit flag field. This gives useful jumps as follows (unused codes are reserved for future modifications).

Field	Decode	Name	Code	Jump on Condition
00000	1	true	T	jump always
00001	/Z	not equal	NE	zero flag is clear
00010	Z	equal	EQ	zero flag is set
00011	C + Z	low or same	LS*	carry flag is set or the zero flag is set
00100	/C	carry clear	CC	carry flag is clear
00101	/C • /Z	high	HI	carry flag is clear and zero flag is clear
00110	/C • Z		??	carry flag is clear and zero flag is set
00111	V	overflow set	VS*	overflow flag is set
01000	C	carry set	CS	carry flag is set
01001	C • /Z		??	carry flag is set and zero flag is clear
01010	C • Z		??	carry flag is set and zero flag is set
01011	/V	overflow clear	VC*	overflow flag is clear
01100	N•V + /N•/V	greater or equal	GE*	overflow and negative flags are the same
01101	N•/V + /N•V	less than	LT*	overflow and negative flags differ
01110	N•V•/Z + /N•/V•/Z	greater than	GT*	not zero, and overflow and negative flags are the same
01111	Z + N•/V + /N•V	less or equal	LE*	zero, or overflow and negative flags differ
10000	debug 1	self interrupt	DSI*	causes the GPU to interrupt itself on interrupt 0, the jump is not taken
10001	debug 2	interrupt	DCI*	causes a CPU interrupt, the jump is not taken
10010	debug 3	single step	DSS*	enters single step mode, the jump is not taken
10011	debug 4	single step and interrupt	DSSI*	causes a CPU interrupt, enters single step mode, the jump is not taken
10100	/N	plus	PL	negative flag is clear
10101	/N • /Z		??	negative flag is clear and zero flag is clear
10110	/N • Z		??	negative flag is clear and zero flag is set
10111	unused			
11000	N	minus	MI	negative flag is set
11001	N• /Z		??	negative flag is set and zero flag is clear
11010	N • Z		??	negative flag is set and zero flag is set
11011	unused			
11100	debug 5	halt	DHT*	halts the processor (GO cleared), the jump is not taken
11101	debug 6	halt and interrupt	DHTI*	causes a CPU interrupt, halts the processor (GO cleared), the jump is not taken
11110	unused			
11111	0	false	F	jump never

* These codes are new to Midsummer, and require the enhanced mode flag to be set.

Help for Debugging

The best approach is not to write bugs in the first place. If you are not perfect, then the six debug codes listed above in the conditional jump table allow you to both pause and stop the J-RISC processor, and also to interrupt it and interrupt the CPU. This should allow a debugger to be written both to run on the processor itself by interrupting itself, and to run on another processor, which will be more useful.

The J-RISC processor is also capable of single-stepping, as discussed below.

All these are special jump condition codes. They will work for JUMP or JR, and in all cases the jump is not taken. The enhanced flag must be set for any of these to work. Because of pipe-lining effects the instruction after the jump will usually get executed before the debug action is taken, e.g. the DSS code will enter single-step mode after the next instruction has been executed (if it was already present in the pre-fetch queue).

The six functions are as follows:

1. The processor interrupts itself on interrupt 0. This is effectively causing an exception, and may be considered analogous to the 68000 illegal instruction.
2. An external CPU interrupt is generated. Execution will continue, so this is mostly useful for flagging some condition.
3. The processor enters single-step mode, i.e. it suspends program execution, usually after the next instruction, and waits for the SINGLE_GO command to continue. See the discussion on single stepping. This is like a break-point.
4. This is the combination of functions 2 and 3, so that the processor suspends its execution into single-step mode and interrupts the external CPU to advise it. This is a useful combination.
5. The processor stops itself. This effectively aborts operation.
6. The processor stops itself and generates an external CPU interrupt. This might be useful for trapping the processor executing code that it should not be.

Single Step Operation

As an aid to the debugging of programs, you can set the J-RISC processor 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.

Self Modifying Code

Self-modifying code carries implicit dangers, both for pre-fetch queues and for caches. This is because the hardware may be maintaining two copies of the instructions, one in physical memory, and one in the pre-fetch queue or cache. It is **never** safe to modify code that may be already cached, but this only

applies to the RCPU. Many programmers believe that self-modifying code is inherently bad style, as it is hard to understand and easy to get wrong. But for those of you who must, here are the rules.

It is possible to write self-modifying code for un-cached memory, as the only issue that arises is that the instruction may already be fetched into the pre-fetch queue. The safest way to write self-modifying code (RCPU: in memory that is not cached), and one which will guarantee portability, is to flush the pre-fetch queue by performing a JUMP or JR before executing the modified code. This ensures that the instructions are fetched from RAM, and all is consistent.

The amount of data in the pre-fetch queue at any moment can vary, and its fetch timing can be disturbed if an external processor is accessing internal space, so it is highly dangerous to modify code a short distance ahead of the current execution point and then linearly execute it. It is necessary to have a distance of at least twelve instructions if you don't wish to use JUMP or JR.

```
        MOVEI   #target,r3
        STORE   r7,(r3)   ; modify the code
        NOP               ; 11 intermediate instructions
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
target: ADD     r0,r0     ; code to be modified
```

Multiply and Accumulate Instructions

The J-RISC processor 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 clock cycle. MAC instructions are not therefore like other instructions, in that a special instruction is needed to write back their result.

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

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

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

DSP only: 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 overflow bits, and they may be read in the multiply/accumulate high result bits register (described later). However, their intended use is with the SAT32S instruction, which 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 it becomes FF80000000 and if it is more than 007FFFFFFF it becomes 007FFFFFFF.

The SAT32S 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.

Matrix Multiplies

The J-RISC processor 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 clock cycle. 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 DCT operation cannot be performed efficiently using the multiply/accumulate mechanism described above because each matrix contains sixty-four operands so that either one of them would occupy all the registers!

The J-RISC processor therefore has a MMULT instruction, which initiates a sequence of between three and fifteen multiply and 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, with the lower element in the low bits. This allows all of an eight by eight matrix to be stored in the secondary register bank, and is the *raison d'être* 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 apart for row access, or the matrix width apart for column access.

Like interrupts, the matrix multiply 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 must not be preceded by a LOAD or STORE instruction.

Divide Unit

The divide unit performs unsigned division, taking as operands a thirty-two bit divisor and dividend, giving a thirty-two bit quotient and a thirty-two 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 clock cycle, so that the complete divide operation completes in sixteen clock cycles. 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 the pipe-line will stall until the divide unit has completed. Otherwise, the divide unit has no effect on instruction flow, as it runs in a completely separate ALU to all other arithmetic functions.

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. The reason for this is that the divider performs non-restoring division at the rate of two bits per clock cycle.

A simple binary division works thus:

```
        subtract the divisor from the dividend
        if the result is positive
                shift 1 into the quotient
        else
                shift 0 into the quotient
                add the divisor back to the dividend (the restore)
        shift the dividend left 1
        repeat
```

As you can see this operation can involve two add/subtracts because of the restore. Non-restoring division gets round this by omitting the restore, but performing an add instead of a subtract at the next iteration. This is because when you do the restore you add the divisor, then at the next iteration you are effectively subtracting half the divisor (because of the shift), the sum of which is the same as adding half the divisor.

When the divide unit completes, the remainder register contains the un-restored dividend value, so it may need the divisor added to it if it is negative, to give the true remainder..

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.

Register File

The J-RISC processor 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 internal address space is accessible to an external bus master at any time - external access having the highest priority on the local bus. This means, for example, that the Blitter might read data from the local RAM.

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

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

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

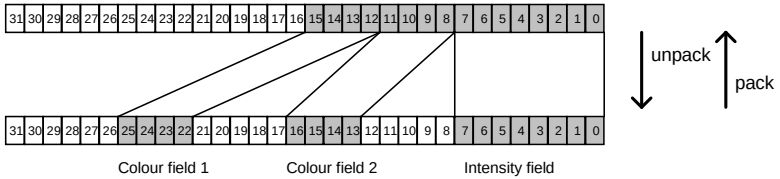
Pack and Unpack

GPU and RCPU only.

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. It can also be operated with 16 bit RGB pixels in a similar manner by setting the PACK_RGB control bit. The pack operation reverses this:

CRY Pixels

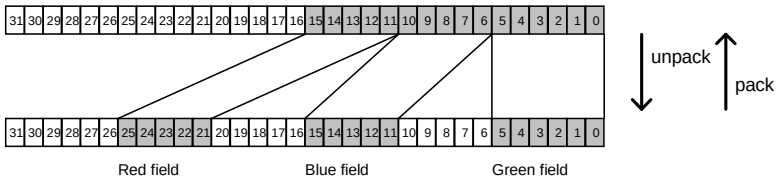
Register containing packed pixel



Register containing unpacked pixel

RGB Pixels

Register containing packed pixel



Register containing unpacked pixel

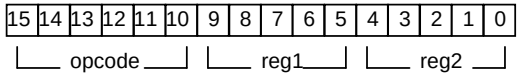
There are five unused bits above each field in an unpacked pixel, allowing up to 32 unpacked pixels to be added together. If a power of two unpacked pixel values are added, then a shift can be used to re-align them prior to packing the average value.

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.

Instruction Set

The J-RISC processor instructions are all sixteen bits, made up as follows:



- op code 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 with some instructions.

The instruction set is as follows, where the syntax is

<Op code name> <source>,<destination>

Note: The reg1 field of single operand instructions must always be set to zero for compatibility with manufacturing test modes and future enhancements.

ABS — Integer absolute value

Syntax	ABS	Rn
Processors	all variants	
Instruction No.	22	
Description	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.	
Flags	Z N C V	set if the result is zero cleared set if the operand was negative not defined
Encoding	`010110 00000 DDDDD` `DDDDD`	 Destination register number, 0-31
Register Usage	Cycle 1 Cycle 3	Destination register read Destination register write and flags are valid

ADD — Integer add

Syntax	ADD	Rn,Rn
Processors	all variants	
Instruction No.	0	
Description	32 bit unsigned or two's complement signed integer add, the result is the destination register contents added to the source register contents, and is written to the destination register.	
Flags	Z N C V	set if the result is zero set if the result is negative represents carry out of the adder set if signed arithmetic overflow has occurred

Encoding	`000000 SSSSS DDDDD`
	`SSSSS`
	`DDDDD`
		
		Source register number, 0-31
		Destination register number, 0-31

Register Usage	Cycle 1 Cycle 3	Source register read & Destination register read Destination register write and flags are valid

ADDC Add with carry

Syntax	ADDC Rn,Rn	
Processors	all variants	
Instruction No.	1	
Description	32 bit unsigned or two's complement integer add with carry in according to the previous state of the carry flag, otherwise like ADD.	
Flags	Z	set if the result is zero
	N	set if the result is negative
	C	represents carry out of the adder
	V	set if signed arithmetic overflow has occurred
Encoding	`000001 SSSSS DDDDD`	
	`SSSSS`	Source register number, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Source register read & Destination register read
	Cycle 3	Destination register write and flags are valid

ADDQ Add with quick data

Syntax	ADDQ n,Rn	
Processors	all variants	
Instruction No.	2	
Description	32 bit unsigned or two's complement integer add, where the source field is immediate data in the range 1-32, otherwise like ADD.	
Encoding	`000010 NNNNN DDDDD`	
	`NNNNN`	Immediate data, 1-32, where 32 encodes as 0
	`DDDDD`	Destination register number, 0-31
Flags	Z	set if the result is zero
	N	set if the result is negative
	C	represents carry out of the adder
	V	set if signed arithmetic overflow has occurred
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

ADDQMOD Add with quick data using modulo arithmetic

Syntax	ADDQMOD n,Rn	
Processors	**DSP**	
Instruction No.	63	
Description	32 bit unsigned or two's complement integer add, where the source field is immediate data in the range 1-32, otherwise like ADD, except that the result bits 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.	
Encoding	`111111 NNNNN DDDDD`	
	`NNNNN`	Immediate data, 1-32, where 32 encodes as 0
	`DDDDD`	Destination register number, 0-31
Flags	Z	set if the result is zero
	N	set if the result is negative
	C	represents carry out of the adder
	V	set if signed arithmetic overflow has occurred
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

ADDQT Add with quick data, transparent

Syntax	ADDQT n,Rn	
Processors	all variants	
Instruction No.	3	
Description	32 bit unsigned or two's complement integer add, like ADDQ except that it is transparent to the flags, which retain their previous values.	
Flags	ZNCV	unaffected
Encoding	`000011 NNNNN DDDDD`	
	`NNNNN`	Immediate data, 1-32, where 32 encodes as 0
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write

AND Logical AND

Syntax	AND Rn,Rn	
Processors	all variants	
Instruction No.	9	
Description	32 bit logical AND, the result is the Boolean AND of the source register contents and the destination register contents, and is written back to the destination register.	
Flags	Z	set if the result is zero
	N	set if the result is negative
	C	not defined
	V	not defined
Encoding	`001001 SSSSS DDDDD`	
	`SSSSS`	Source register number, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Source register read & Destination register read
	Cycle 3	Destination register write and flags are valid

BCLR Bit clear

Syntax	BCLR n,Rn	
Processors	all variants	
Instruction No.	15	
Description	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 destination register is now all zero
	N	set from bit 31 of the result
	C	not defined
	V	not defined

Encoding	`001111 NNNNN DDDDD`	
	`NNNNN`	Bit select for the operation, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

BSET Bit set

Syntax	BSET n,Rn	
Processors	all variants	
Instruction No.	14	
Description	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
	V	reflects the state of the set bit before it was modified
Encoding	`001110 NNNNN DDDDD`	
	`NNNNN`	Bit select for the operation, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

BTST Bit test

Syntax	BTST n,Rn	
Processors	all variants	
Instruction No.	13	
Description	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
	C	not defined
	V	not defined
Encoding	`001101 NNNNN DDDDD`	
	`NNNNN`	Bit select for the operation, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Flags are valid

CMP Compare

Syntax	CMP Rn,Rn	
Processors	all variants	
Instruction No.	30	
Description	32 bit compare, the source register contents are subtracted from the destination register contents 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
	V	set if arithmetic overflow was generated by the subtract
Encoding	`011110 SSSSS DDDDD`	
	`SSSSS`	Source register number, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Source register read & Destination register read
	Cycle 3	Flags are valid

CMPQ Compare with quick data

Syntax	CMPQ n,Rn	
Processors	all variants	
Instruction No.	31	
Description	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
	V	set if arithmetic overflow was generated by the subtract
Encoding	`011111 NNNNN DDDDD`	
	`NNNNN`	Immediate data, -16 to +15, two's complement value
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Flags are valid

DIV Unsigned divide

Syntax	DIV Rn,Rn	
Processors	all variants	
Instruction No.	21	
Description	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	ZNCV	unaffected
Encoding	`010101 SSSSS DDDDD`	
	`SSSSS`	Source register number, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Source register read & Destination register read
	Cycle 18	Destination register write

IMACN Signed integer multiply/accumulate, no write-back

Syntax	IMACN Rn,Rn
Processors	all variants
Instruction No.	20

Description	16 bit signed integer multiply and accumulate, like IMULT, except that the 32 bit product is added to the result of the previous arithmetic operation, and the result is not written back to the destination register. Intended to be used after IMULTN to give a multiply/accumulate group.
Flags	ZNCV unaffected
Register Usage	Cycle 1 Source register read & Destination register read
Encoding	`010100 SSSSS DDDDD` `SSSSS` Source register number, 0-31 `DDDDD` Destination register number, 0-31
Notes	Refer to the section on Multiply and Accumulate instructions

IMULT Signed integer multiply

Syntax	IMULT Rn,Rn
Processors	all variants
Instruction No.	17
Description	16 bit signed integer multiply, the 32 bit result is the signed integer product of the bottom 16 bits of each of the source and destination registers, and is written back to the destination register.
Flags	Z set if the result is zero N set if the result is negative C not defined V not defined
Encoding	`010001 SSSSS DDDDD` `SSSSS` Source register number, 0-31 `DDDDD` Destination register number, 0-31
Register Usage	Cycle 1 Source register read & Destination register read Cycle 3 Destination register write and flags are valid

IMULTN Signed integer multiply, no write-back

Syntax	IMULTN Rn,Rn
Processors	all variants
Instruction No.	18
Description	Like IMULT, but result is not written back to destination register. Intended to be used as the first of a multiply/accumulate group, as there are potential speed advantages in not writing back the result.
Flags	Z set if the result is zero N set if the result is negative C not defined V not defined
Encoding	`010010 SSSSS DDDDD` `SSSSS` Source register number, 0-31 `DDDDD` Destination register number, 0-31
Register Usage	Cycle 1 Source register read & Destination register read
Notes	Refer to the section on Multiply and Accumulate instructions

JR Jump relative

Syntax	JR cc,n
Processors	all variants
Instruction No.	53
Description	Relative jump to the location given by the sum of the address of the next instruction and the immediate data in the source field, which is signed and therefore in the range +15 to -16 words. The condition codes encode as described above under conditional jumps.
Flags	ZNCV unaffected
Encoding	`110101 NNNNN CCCCC` `NNNNN` Jump offset in words, -16 to 15, two's complement value `CCCCC` Condition code, described earlier
Register Usage	Cycle 1 Flags must be valid

JRE Extended range jump relative

Syntax	JRE n
Processors	all variants
Instruction No.	57
Description	Unconditional relative jump to the location given by the sum of the address of the next instruction and the immediate data in the combined source and destination fields, which is signed and therefore in the range +511 to -512 words. An offset of 0 is decoded as NOP, however this op-code will always be treated as NOP unless the enhanced bit is set (see NOP). The assembler should accept JR for this instruction and encode it appropriately.
Flags	ZNCV unaffected
Encoding	`111001 NNNNNNNNNN` `NNNNNNNNNN` Jump offset in words, -512 to 511, two's complement value
Register Usage	none

JUMP Jump absolute

Syntax	JUMP cc,(Rn)
Processors	all variants
Instruction No.	52
Description	Jump to location pointed to by the source register, destination field is the condition code, where the bits encode as described above under conditional jumps.
Flags	ZNCV unaffected
Encoding	`110100 SSSSS CCCCC` `SSSSS` Source register number, 0-31 `CCCCC` Condition code, described earlier
Register Usage	Cycle 1 Source register read and flags must be valid

LOAD Load long

Syntax	LOAD (Rn),Rn
Processors	all variants
Instruction No.	41
Description	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	ZNCV	unaffected	
Encoding	`101001 SSSSS DDDDD`		
	`SSSSS`	Source register number, 0-31	
	`DDDDD`	Destination register number, 0-31	
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

LOAD Load long, with indexed address

Syntax	LOAD	(R14+n),Rn
	LOAD	(R15+n),Rn
Processors	all variants	
Instruction No.	43, 44	
Description	32 bit memory read, as LOAD, except that the address is given by the sum of either R14 or R15 and the immediate data in the source register field, in the range 1-32. The offset is in long words, not in bytes, therefore a divide by four should be used on any label arithmetic to give the offset. This is slower than normal LOAD operations due to the two clock cycle overhead of computing the address.	
Flags	ZNCV	unaffected
Encoding	`101011 NNNNN DDDDD`	
	`101100 NNNNN DDDDD`	
	`NNNNN`	Address offset in long words, 1-32, where 32 encodes as 0
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	R14 or R15 register read
	Cycle n	Destination register write internal memory at cycle 3 or 4 external memory subject to bus latency

LOAD Load long, from register with base offset address

Syntax	LOAD	(R14+Rn),Rn
	LOAD	(R15+Rn),Rn
Processors	all variants	
Instruction No.	58, 59	
Description	32 bit memory load from the byte address given by the sum of R14 and the source register (the address should be on a long-word boundary). Otherwise like instructions 43 and 44.	
Flags	ZNCV	unaffected
Encoding	`111010 SSSSS DDDDD`	
	`111011 SSSSS DDDDD`	
	`SSSSS`	Source register number, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	R14 or R15 register read & Source register read
	Cycle n	Destination register write internal memory at cycle 5 or 6 external memory subject to bus latency

LOADB Load byte

Syntax	LOADB (Rn),Rn	
Processors	all variants	
Instruction No.	39	
Description	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 and some local RAM (refer to the discussion of each J-RISC processor), all other internal memory will perform a 32 bit read.	
Flags	ZNCV	unaffected
Encoding	`100111 SSSSS DDDDD`	
	`SSSSS`	Source register number, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Source register read
	Cycle n	Destination register write internal memory at cycle 5 or 6 external memory subject to bus latency

LOADW Load word

Syntax	LOADW (Rn),Rn	
Processors	all variants	
Instruction No.	40	
Description	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 and some local RAM (refer to the discussion of each J-RISC processor), all other internal memory will perform a 32 bit read.	
Flags	ZNCV	unaffected
Encoding	`101000 SSSSS DDDDD`	
	`SSSSS`	Source register number, 0-31
	`DDDDD`	Destination register number, 0-31
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

LOADP Load phrase

Syntax	LOADP (Rn),Rn	
Processors	**GPU & RCPU**	
Instruction No.	42	
Description	64 bit memory read. The source register contains a 32 bit byte address, which must be phrase aligned. The destination register will have the low long-word loaded into it, the high long-word is available in the high-half register. This applies to external memory only, internal memory will perform a 32 bit read.	
Flags	ZNCV	unaffected
Encoding	`101010 SSSSS DDDDD`	
	`SSSSS`	Source register number, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Source register read
	Cycle n	Destination register write, external memory subject to bus latency

MIRROR Mirror operand

Syntax	MIRROR Rn
Processors	**DSP**

Instruction No.	48
Description	The register is mirrored bit-wise, i.e. bit 0 goes to bit 31, bit 1 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 V not defined
Encoding	`110000 00000 DDDDD` `DDDDD` Destination register number, 0-31
Register Usage	Cycle 1 Destination register read Cycle 3 Destination register write and flags are valid

MMULT Matrix multiply

Syntax	MMULT Rn,Rn
Processors	all variants
Instruction No.	54
Description	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 V set if arithmetic overflow occurred
Encoding	`110110 SSSSS DDDDD` `SSSSS` Source register number, 0-31 `DDDDD` Destination register number, 0-31
Register Usage	Refer to the discussion of multiply/accumulate

MOVE Move register to register

Syntax	MOVE Rn,Rn
Processors	all variants
Instruction No.	34
Description	32 bit register to register transfer.
Flags	ZNCV unaffected
Encoding	`100010 SSSSS DDDDD` `SSSSS` Source register number, 0-31 `DDDDD` Destination register number, 0-31
Register Usage	Cycle 1 Source register read Cycle 2 Destination register write

MOVE Move program count to register

Syntax	MOVE PC,Rn
Processors	all variants
Instruction No.	51
Description	Load the destination register with the address of the current instruction. The actual value read from the PC is modified to take into account the effects of pipe-lining and prefetch, to give the correct address. This is the only way for the GPU to read its own PC.
Flags	ZNCV unaffected
Encoding	`110011 00000 DDDDD` `DDDDD` Destination register number, 0-31
Register Usage	Cycle 2 Destination register write

MOVEFA Move from alternate register

Syntax	MOVEFA Rn,Rn
Processors	all variants
Instruction No.	37
Description	32 bit alternate register to register transfer, the source register lying in the other bank of 32 registers.
Flags	ZNCV unaffected
Encoding	`100101 SSSSS DDDDD` `SSSSS` Source register number, 0-31 `DDDDD` Destination register number, 0-31
Register Usage	Cycle 1 Source register read Cycle 2 Destination register write

MOVEI Move immediate

Syntax	MOVEI n,Rn
Processors	all variants
Instruction No.	38
Description	32 bit register load with next 32 bits of instruction stream. The first word in the instruction stream is the low word, the second the high word. This instruction always takes at least three clock cycles to complete, and is the sole exception to the 16 bit instruction size. Note that the operand word ordering is little-endian.
Flags	ZNCV unaffected

Encoding	`100110 00000 DDDDD`
	`NNNNNNNNNNNNNNNN`
	`NNNNNNNNNNNNNNNN`
	`DDDDD` Destination register number, 0-31
	`NNNNNNNNNNNNNNNN` Immediate data words as described above

Register Usage	Cycle 4 Destination register write

MOVEQ Move quick data

Syntax	MOVEQ n,Rn
Processors	all variants
Instruction No.	35
Description	32 bit register load with immediate value in the range 0-31.
Flags	ZNCV unaffected

Encoding	`100011 NNNNN DDDDD`	
	`NNNNN`	Immediate data, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 2	Destination register write

MOVETA Move to alternate register

Syntax	MOVETA Rn,Rn	
Processors	all variants	
Instruction No.	36	
Description	32 bit register to alternate register transfer, the destination register lying in the other bank of 32 registers.	
Flags	ZNCV	unaffected
Encoding	`100100 SSSSS DDDDD`	
	`SSSSS`	Source register number, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Source register read
	Cycle 2	Destination register write

MTOI Mantissa to integer

Syntax	MTOI Rn,Rn	
Processors	all variants	
Instruction No.	55	
Description	Extract the mantissa and sign from the IEEE 32 bit floating-point number in the source register, and create a signed integer in the destination. The most significant bit is bit 23, but it is sign extended.	
Flags	Z	set if the result is zero
	N	set if the result is negative
	C	not defined
	V	not defined
Encoding	`110111 SSSSS DDDDD`	
	`SSSSS`	Source register number, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Source register read
	Cycle 3	Destination register write and flags are valid

MULT Multiply

Syntax	MULT Rn,Rn	
Processors	all variants	
Instruction No.	16	
Description	16 bit unsigned integer multiply, the 32 bit result is the unsigned integer product of the bottom 16 bits of each of the source and destination registers, and is written back to the destination register.	
Flags	Z	set if the result is zero
	N	set if bit 31 of the result is one
	C	not defined
	V	not defined
Encoding	`010000 SSSSS DDDDD`	
	`SSSSS`	Source register number, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Source register read & Destination register read
	Cycle 3	Destination register write and flags are valid

NEG Negate

Syntax	NEG Rn	
Processors	all variants	
Instruction No.	8	
Description	32 bit two's complement 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
	V	not defined
Encoding	`001000 00000 DDDDD`	
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

NOP Null Operation

Syntax	NOP	
Processors	all variants	
Instruction No.	57	
Description	Do nothing — if the enhanced mode bit is set then this instruction can become extended jump relative (JRE, see above). As long as the source and destination fields are zero this is still decoded as NOP, and behaves exactly as before.	
Flags	ZNCV	unaffected
Encoding	`111001 0000000000`	
Register Usage	none	

NORMI Normalisation integer

Syntax	NORMI Rn,Rn	
Processors	all variants	
Instruction No.	56	
Description	Gives the floating point 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 as an IEEE 32 bit floating point value (the normalisation integer 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
	V	not defined

Encoding	`111000 SSSSS DDDDD`	
	SSSSS	Source register number, 0-31
	DDDDD	Destination register number, 0-31
Register Usage	Cycle 1	Source register read
	Cycle 3	Destination register write and flags are valid

NOT Logical NOT

Syntax	NOT	Rn
Processors	all variants	
Instruction No.	12	
Description	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
	V	not defined
Encoding	`001100 00000 DDDDD`	
	DDDDD	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

OR Logical OR

Syntax	OR	Rn,Rn
Processors	all variants	
Instruction No.	10	
Description	32 bit logical or operation, the result is the Boolean OR of the source register contents 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
	V	not defined
Encoding	`001010 SSSSS DDDDD`	
	SSSSS	Source register number, 0-31
	DDDDD	Destination register number, 0-31
Register Usage	Cycle 1	Source register read & Destination register read
	Cycle 3	Destination register write and flags are valid

PACK Pack 16 bit CRY or RGB Pixel

Syntax	PACK	Rn
Processors	**GPU and RCPU**	
Instruction No.	63	
Description	Takes an unpacked pixel value and packs it into a 16 bit CRY or RGB pixel. The three fields, with five bit gaps between them are mapped onto the low sixteen bits, and the top sixteen bits are set to zero. The reg1 field should be set to zero to differentiate this from UNPACK;	
Flags	ZNCV	unaffected
Encoding	`111111 00000 DDDDD`	
	DDDDD	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write
Notes	See the section on Pack and Unpack	

RESMAC Multiply/accumulate result write

Syntax	RESMAC	Rn
Processors	all variants	
Instruction No.	19	
Description	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.	
Flags	ZNCV	unaffected
Encoding	`010011 00000 DDDDD`	
	DDDDD	Destination register number, 0-31
Register Usage	Cycle 3	Destination register write
Notes	Refer to the section on Multiply and Accumulate instructions	

ROR Rotate right

Syntax	ROR	Rn,Rn
Processors	all variants	
Instruction No.	28	
Description	The value in the destination register is shifted right by the value in the source register modulo thirty-two. This is effectively the same as a thirty-two bit rotate right by the bottom five bits of the source register. You can use this instruction for rotate left by complementing the value in the source register.	
Flags	Z	set if the result is zero
	N	set if the result is negative
	C	represents bit 31 of the un-shifted data
	V	not defined
Encoding	`011100 SSSSS DDDDD`	
	SSSSS	Source register number, 0-31
	DDDDD	Destination register number, 0-31
Register Usage	Cycle 1	Source register read & Destination register read
	Cycle 3	Destination register write and flags are valid

RORQ Rotate right by immediate count

Syntax	RORQ	n,Rn
Processors	all variants	
Instruction No.	29	
Description	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
	V	not defined

Encoding	`011101 NNNNN DDDDD`	
	`NNNNN`	Shift count, 1-32, where 32 encodes as 0, and has much the same effect
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

SAT8 Saturate to eight bits

Syntax	SAT8 Rn	
Processors	**GPU & RCPU**	
Instruction No.	32	
Description	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 000000FFh (255) it is set to that. 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
	V	not defined
Encoding	`100000 00000 DDDDD`	
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

SAT16 Saturate to sixteen bits

Syntax	SAT16 Rn	
Processors	**GPU and RCPU**	
Instruction No.	33	
Description	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 0000FFFFh (65,535) it is set to that. This is useful for computed Z, et cetera, to counteract the effect of rounding errors.	
Flags	Z	set if the result is zero
	N	cleared
	C	not defined
	V	not defined
Encoding	`100001 00000 DDDDD`	
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

SAT16S Saturate to sixteen signed bits

Syntax	SAT16S Rn	
Processors	**DSP**	
Instruction No.	33	
Description	Saturate the 32 bit signed integer operand value to a 16 bit signed integer. If it is less than FFFF8000h it is set to that, if it is greater than 00007FFFh it is set to that. This is useful for computed audio sample values, and so on, to counteract the effect of rounding errors.	
Flags	Z	set if the result is zero
	N	set if the result is negative
	C	not defined
	V	not defined
Encoding	`100001 00000 DDDDD`	
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

SAT24 Saturate to twenty-four bits

Syntax	SAT24 Rn	
Processors	**GPU and RCPU**	
Instruction No.	62	
Description	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 00FFFFFFh (16,777,215) it is set to that. 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
	V	not defined
Encoding	`111110 00000 DDDDD`	
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

SAT32S Saturate RESMAC value to thirty-two bit signed

Syntax	SAT32S Rn	
Processors	**DSP**	
Instruction No.	42	
Description	Saturate the 40 bit signed integer operand value to an 32 bit signed integer. This uses the overflow bits from 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 then 7FFFFFFFh it saturates to that.	
Flags	Z	set if the result is zero
	N	set if the result is negative
	C	not defined
	V	not defined
Encoding	`101010 00000 DDDDD`	
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

SH Shift

Syntax	SH Rn,Rn	
Processors	all variants	
Instruction No.	23	

Description	This instruction performs a thirty-two bit shift either to the left or to the right as given by the shift value in the source register. A positive shift value causes a shift to the right, a negative shift value gives a shift to the left. Shift values greater than plus thirty-two or less than minus thirty-two give a result of zero as all the bits are shifted out. Zero is always shifted in, so you should use the SHA instruction if you require sign-extension of values shifted to the right.	
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
	V	not defined
Encoding	`010111 SSSSS DDDDD`	
	`SSSSS`	Source register number, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Source register read & Destination register read
	Cycle 3	Destination register write and flags are valid

SHA Shift arithmetic

Syntax	SHA	Rn,Rn
Processors	all variants	
Instruction No.	26	
Description	This instruction performs a thirty-two bit arithmetic shift either to the left or to the right as given by the shift value in the source register. An arithmetic shift means that a shift right is sign-extended, that is the value shifted in is the top bit of the value being shifted. A positive shift value causes a sign-extended shift to the right, a negative shift value gives a normal shift to the left. Shift values greater than plus thirty-two give either zero or minus one depending on the sign of the value being shifted, shift values of less than minus thirty-two give a result of zero. You should use the SH instruction if you do not require sign-extension of values.	
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
	V	not defined
Encoding	`011010 SSSSS DDDDD`	
	`SSSSS`	Source register number, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Source register read & Destination register read
	Cycle 3	Destination register write and flags are valid

SHARQ Shift Arithmetic Right, with Immediate Shift Count

Syntax	SHARQ	n,Rn
Processors	all variants	
Instruction No.	27	
Description	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
	V	not defined
Encoding	`011011 NNNNN DDDDD`	
	`NNNNN`	Shift count, 1-32, where 32 encodes as 0
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

SHLQ Shift left with immediate shift count

Syntax	SHLQ	n,Rn
Processors	all variants	
Instruction No.	24	
Description	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
	V	not defined
Encoding	`011000 NNNNN DDDDD`	
	`NNNNN`	Immediate data, 1-32, where 32 encodes as 0
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

SHRQ Shift right with immediate shift count

Syntax	SHRQ	n,Rn
Processors	all variants	
Instruction No.	25	
Description	As SHLQ 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
	V	not defined
Encoding	`011001 NNNNN DDDDD`	
	`NNNNN`	Immediate data, 0-31
	`DDDDD`	Destination register number, 0-31
Register Usage	Cycle 1	Destination register read
	Cycle 3	Destination register write and flags are valid

STORE Store long

Syntax	STORE	Rn,(Rn)
Processors	all variants	
Instruction No.	47	
Description	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. Note that for all store instructions, the notion of source and destination register fields is the reverse of that used for all other instructions.	
Flags	ZNCV	unaffected
Encoding	`101111 SSSSS DDDDD`	
	`SSSSS`	Source register number, 0-31
	`DDDDD`	Destination register number, 0-31

Register Usage	Cycle 1	Source register read & Destination register read

STORE Store long, with indexed address

Syntax	STORE Rn,(R14+n) STORE Rn,(R15+n)	
Processors	all variants	
Instruction No.	49, 50	
Description	32 bit memory write, write as STORE, with address generation in the same manner as the equivalent LOAD instructions. The destination register contains the data to be written.	
Flags	ZNCV	unaffected

Encoding	`110001 SSSSS DDDDD`
	`110010 SSSSS DDDDD`
	`NNNNN` Address offset in long words, 1-32, where 32 encodes as 0
	`DDDDD` Destination register number, 0-31	

Register Usage	Cycle 1 Cycle 2	R14 or R15 register read Source register read

STORE Store long, to register with base offset address

Syntax	STORE Rn,(R14+Rn) STORE Rn,(R15+Rn)	
Processors	all variants	
Instruction No.	60, 61	
Description	32 bit memory store to the byte address given by the sum of R14 and the destination register (the address should be on a long-word boundary). Otherwise like instructions 49 and 50. The destination register contains the data to be written.	
Flags	ZNCV	unaffected

Encoding	`111100 SSSSS DDDDD`
	`111101 SSSSS DDDDD`
	`SSSSS` Source register number, 0-31
	`DDDDD` Destination register number, 0-31	

Register Usage	Cycle 1 Cycle 2	R14 or R15 register read & Destination register read Source register read

STOREB Store byte

Syntax	STOREB Rn,(Rn)	
Processors	all variants	
Instruction No.	45	
Description	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 and some local RAM (refer to the discussion of each J-RISC processor), all other internal memory will perform a 32 bit write.	
Flags	ZNCV	unaffected
Encoding	`101101 SSSSS DDDDD` `SSSSS` Source register number, 0-31 `DDDDD` Destination register number, 0-31	
Register Usage	Cycle 1	Source register read & Destination register read

STOREP Store phrase

Syntax	STOREP Rn,(Rn)	
Processors	**GPU and RCPU**	
Instruction No.	48	
Description	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	ZNCV	unaffected
Encoding	`110000 SSSSS DDDDD` `SSSSS` Source register number, 0-31 `DDDDD` Destination register number, 0-31	
Register Usage	Cycle 1	Source register read & Destination register read

STOREW Store word

Syntax	STOREW Rn,(Rn)	
Processors	all variants	
Instruction No.	46	
Description	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 and some local RAM (refer to the discussion of each J-RISC processor), all other internal memory will perform a 32 bit write.	
Flags	ZNCV	unaffected
Encoding	`101110 SSSSS DDDDD` `SSSSS` Source register number, 0-31 `DDDDD` Destination register number, 0-31	
Register Usage	Cycle 1	Source register read & Destination register read

SUB Subtract

Syntax	SUB Rn,Rn	
Processors	all variants	
Instruction No.	4	
Description	32 bit unsigned or two's complement integer subtract, result is the source register contents subtracted from the destination register contents, and is written to the destination register. The carry flag represents borrow out of the subtract, and the zero flag is set if the result is zero.	
Flags	Z N C V	set if the result is zero set if the result is negative represents borrow out of the subtract set if signed arithmetic overflow occurred
Encoding	`000100 SSSSS DDDDD` `SSSSS` Source register number, 0-31 `DDDDD` Destination register number, 0-31	
Register Usage	Cycle 1 Cycle 3	Source register read & Destination register read Destination register write and flags are valid

SUBC Subtract with borrow

Syntax	SUBC Rn,Rn
Processors	all variants
Instruction No.	5
Description	32 bit unsigned or two's complement 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 V set if signed arithmetic overflow occurred
Encoding	`000101 SSSSS DDDDD` `SSSSS` Source register number, 0-31 `DDDDD` Destination register number, 0-31
Register Usage	Cycle 1 Source register read & Destination register read Cycle 3 Destination register write and flags are valid

SUBQ Subtract with immediate data

Syntax	SUBQ n,Rn
Processors	all variants
Instruction No.	6
Description	32 bit two's complement 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 V set if signed arithmetic overflow occurred
Encoding	`000110 NNNNN DDDDD` `NNNNN` Immediate data, 1-32, where 32 encodes as 0 `DDDDD` Destination register number, 0-31
Register Usage	Cycle 1 Destination register read Cycle 3 Destination register write and flags are valid

SUBQMOD Subtract with immediate data using modulo arithmetic

Syntax	SUBQMOD n,Rn
Processors	**DSP**
Instruction No.	32
Description	32 bit two's complement integer subtract, where the source field is immediate data in the range 1-32, otherwise like SUB, except that the result bits 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 V set if signed arithmetic overflow occurred
Encoding	`100000 NNNNN DDDDD` `NNNNN` Immediate data, 1-32, where 32 encodes as 0 `DDDDD` Destination register number, 0-31
Register Usage	Cycle 1 Destination register read Cycle 3 Destination register write and flags are valid

SUBQT Subtract with immediate data, transparent

Syntax	SUBQT n,Rn
Processors	all variants
Instruction No.	7
Description	32 bit unsigned or two's complement integer subtract, like SUBQ except that it is transparent to the flags, which retain their previous values.
Flags	ZNCV unaffected
Encoding	`000111 NNNNN DDDDD` `NNNNN` Immediate data, 1-32, where 32 encodes as 0 `DDDDD` Destination register number, 0-31
Register Usage	Cycle 1 Destination register read Cycle 3 Destination register write

UNPACK Unpack 16 bit CRY or RGB Pixel

Syntax	UNPACK Rn
Processors	**GPU and RCPU**
Instruction No.	63
Description	Takes a packed CRY or RGB 16 bit pixel value and unpacks it into a 32 bit integer. The three fields are spread out with five bit gaps between them, and all other bits are set to zero. The reg1 field should be set to one to differentiate this from PACK.
Flags	ZNCV unaffected
Encoding	`111111 00001 DDDDD` `DDDDD` Destination register number, 0-31
Register Usage	Cycle 1 Destination register read Cycle 3 Destination register write
Notes	See the section on Pack and Unpack

XOR Logical exclusive OR

Syntax	XOR Rn,Rn
Processors	all variants
Instruction No.	11
Description	32 bit logical exclusive or, the result is the Boolean XOR of the source register contents 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 V not defined

Encoding	`001011 SSSSS DDDDD`	
	SSSSS	Source register number, 0-31
	DDDDD	Destination register number, 0-31
Register Usage	Cycle 1	Source register read & Destination register read
	Cycle 3	Destination register write and flags are valid

Writing Fast J-RISC Programs

To get the most out of the J-RISC processor, it is important to avoid what are traditionally known as wait states, but in a RISC context are usually called **pipe-line stalls** or just stalls. These are when the instruction pipe-line has to pause until some system resource becomes available, and are generally to control the use of limited hardware resources; or to protect the programmer from some out-of-order effect, such as using a register before it is valid as the result of some previous operation.

The processor can execute one instruction per clock cycle in ideal circumstances, but it is very easy for code to be subject to so many stalls that it only achieves 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 here should help do that. A well written program can usually achieve an instruction throughput of around three-quarters of the peak figure.

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

The register bank is a source of stalls because it has only two read/write ports, so that two reads, a read and a write, or two writes can occur in any given clock cycle. If a result is being written at the same time as an instruction that requires two reads, then a stall will occur — unless the write register matches one of the two read registers, in which case the write occurs and the write data is provided as if the read was taking place. The instruction set list shows the register usage of all instructions.

Instructions dependant on the flags can also be subject to stalls, the flags are not valid until the clock cycle in which the result is written back, so that if a ADD instruction is followed by a JUMP then a one clock cycle stall will ensue, the JUMP executing in the clock cycle in which the result of the ADD is written back.

Pipe-line stalls are incurred when:

- an instruction reads a register containing the result of the previous instruction, one clock cycle of wait is incurred until the previous operation completes.
- an instruction uses the flags from the previous instruction, one clock cycle of wait is incurred until the previous operation completes.
- an ALU result, memory load value or divide result has to be written back and neither register operand of the instruction about to be executed matches, one clock cycle of wait is incurred to let the data be written.
- two values are to be written back at once, one clock cycle of wait is incurred (this is unusual).
- 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 reads a register which is awaiting data from an incomplete memory read, this will be no more than one clock cycle from internal memory, but can be several clock cycles from external memory.
- a load or store instruction is about to be executed and the memory interface has not completed the external bus cycle for the a previous external load or store (the wait logic cannot determine if the transfer is internal or external before the instruction executes, so all loads and stores are held up if there is an external transfer incomplete).
- after a store instruction with an indexed addressing mode (one clock cycle).
- after a JUMP or JR (three clock cycles if executing out of internal memory).
- if the next instruction has not been read, this will only occur when executing out of external memory.
- during a matrix multiply if the CPU accesses internal space.

The most common cause of pipe-line stalls is using a register which was altered by the previous instruction. For example consider this code fragment:

```
1       add     r3,r0                   ; add offset to X
2       shrq    1,r0                    ; apply scaling factor
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
```

Stalls will be incurred after instructions 1, 2, 4 and 5. If the code were laid out like this:

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

No stalls would occur. This is an example if *interleaving*, and this is a powerful technique for speeding up your code. It is well worth the performance enhancement - 6 clock cycles instead of 10 in this example - to ensure that your 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.

Graphics Processor - GPU

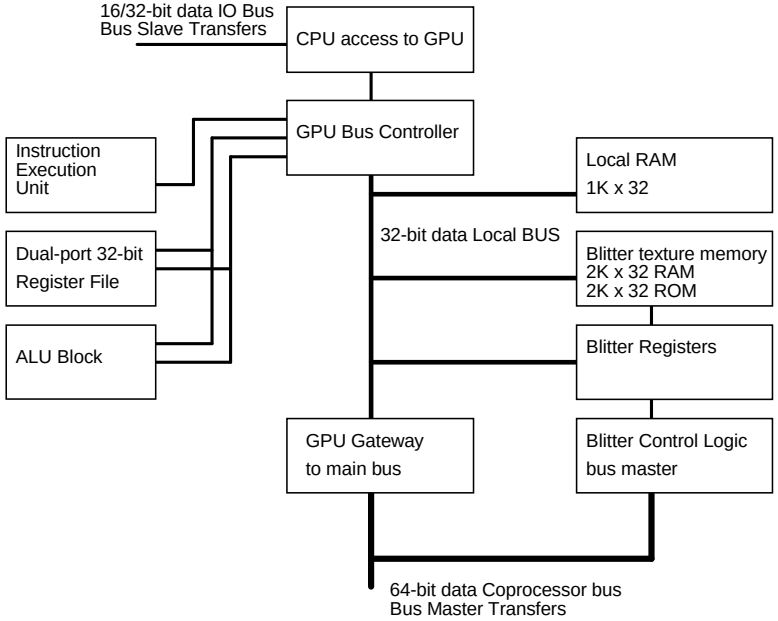
The Oberon Graphics Processor Subsystem contains one Jaguar RISC processor as described above (this one is known as the GPU — Graphics Processing Unit) and the blitter, whose control registers are in the GPU internal memory space. The GPU is a self-contained processing unit which runs in parallel with the rest of the system, but which is able to access the main system bus. External memory is controlled by a separate memory controller, which is not part the graphics processor 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 prioritised, i.e. ownership can be lost during a request (but not during a memory cycle). The graphics subsystem 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 external 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. All GPU transfers in the local space, to both memory and registers, can occur in parallel with activity on the main (external) bus, and are therefore efficient in terms of system use.

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



The three blocks at the lower right form the blitter. The control interface of the blitter is completely within the GPU space, but the blitter is an independent bus master to the GPU, i.e. it will start up and perform a blit completely independently from the GPU. The blitter texture memory is available to the GPU as 32 bit RAM (only long transfers are available), and may be used as general GPU RAM when not being used for texture mapping. If texture operation is restricted to one 1K x 32 bank of this RAM, then the GPU may access the other 1K x 32 bank without affecting blitter performance.

Memory Map

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

Addr.	Name	r/w	Description
F02000	GPU_REGS	RW	GPU registers, sixty-four 32 bit locations
F02100	GPU_FLAGS	RW	GPU flags
F02104	GPU_MTXC	W	GPU matrix control
F02108	GPU_MTXA	W	GPU matrix address
F0210C	GPU_BIGEND	W	GPU big / little endian control
F02110	GPU_PC	RW	GPU program counter
F02114	GPU_CTRL	RW	GPU operation control / status
F02118	GPU_HIDATA	RW	GPU bus interface high data
F0211C	GPU_REMAIN	R	GPU division remainder
F02120	GPU_DMACNT	W	GPU DMA transfer count
F02124	GPU_DMACTL	W	GPU DMA control register
F02124	GPU_DMASTAT	R	GPU DMA status
F02128	GPU_DMAEA	W	GPU DMA external address
F0212C	GPU_DMAIA	W	GPU DMA internal address
F02200	A1_BASE	W	Blitter A1 base
F02204	A1_FLAGS	W	Blitter A1 flags
F02208	A1_CLIP	W	Blitter A1 window size
F0220C	A1_PIXEL	RW	Blitter A1 pointer
F02210	A1_STEP	W	Blitter A1 step
F02214	A1_FSTEP	W	Blitter A1 step fraction
F02218	A1_FPIXEL	RW	Blitter A1 pointer fraction
F0221C	A1_INC	W	Blitter A1 pointer increment
F02220	A1_FINC	W	Blitter A1 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 pointer
F02234	A2_STEP	W	Blitter A2 step
F02238	BLIT_CMD	W	Blitter command
F0223C	BLIT_COUNT	W	Blitter loop counters
F02240	BLIT_SRCD	W	Blitter source data
F02248	BLIT_DSTD	W	Blitter destination data
F02250	BLIT_DSTZ	W	Blitter destination Z data
F02258	BLIT_SRCZ1	W	Blitter source Z data 1
F02260	BLIT_SRCZ2	W	Blitter source Z data 2
F02268	BLIT_PATD	W	Blitter pattern data
F02270	BLIT_IINC	W	Blitter intensity increment
F02274	BLIT_ZINC	W	Blitter Z increment
F02278	BLIT_STOP	W	Blitter collision stop control
F0227C	BLIT_I0	W	Blitter intensity register 0
F02280	BLIT_I1	W	Blitter intensity register 1
F02284	BLIT_I2	W	Blitter intensity register 2
F02288	BLIT_I3	W	Blitter intensity register 3
F0228C	BLIT_Z0	W	Blitter Z register 0
F02290	BLIT_Z1	W	Blitter Z register 1
F02294	BLIT_Z2	W	Blitter Z register 2
F02298	BLIT_Z3	W	Blitter Z register 3
F0229C	BLIT_FINNER	W	Fractional part of the inner counter and extended command
F022A0	BLIT_IDELTA	W	Inner counter initial value delta
F022A4	A1_XSD	W	A1 X step delta value
F022A8	A1_YSD	W	A1 Y step delta value
F022AC	BLIT_ISTEP	W	Intensity step value
F022B0	BLIT_ISD	W	Intensity step value delta
F022B4	BLIT_ZSTEP	W	Z step value
F022B8	BLIT_ZSD	W	Z step value delta.
F022BC	BLIT_X0	W	Texture X address pointer 0
F022C0	BLIT_X1	W	Texture X address pointer 1
F022C4	BLIT_X2	W	Texture X address pointer 2
F022C8	BLIT_X3	W	Texture X address pointer 3
F022CC	BLIT_Y0	W	Texture Y address pointer 0
F022D0	BLIT_Y1	W	Texture Y address pointer 1
F022D4	BLIT_Y2	W	Texture Y address pointer 2
F022D8	BLIT_Y3	W	Texture Y address pointer 3
F022DC	BLIT_XINC	W	Texture X inner loop increment
F022E0	BLIT_XSTEP	W	Texture X outer loop step
F022E4	BLIT_XSD	W	Texture X outer loop step delta
F022E8	BLIT_YINC	W	Texture Y inner loop increment
F022EC	BLIT_YSTEP	W	Texture Y outer loop step
F022F0	BLIT_YSD	W	Texture Y outer loop step delta
F022F4	BLIT_TBASE	W	Texture base address
F022F8	BLIT_IINCX	W	Alternate intensity increment register
F022FC	A1_MASK	W	A1 window address mask.
F02300	A2_CLIP	W	A2 clipping window size
F02304	A1_X	W	Alternate view of the A1 X pixel pointer and its fractional part
F02308	A1_Y	W	Alternate view of the A1 Y pixel pointer and its fractional part
F0230C	A2_X	W	Alternate view of A2 X pixel pointer
F02310	A2_Y	W	Alternate view of A2 Y pixel pointer
F02314	A1_XSTEP	W	Alternate view of the A1 X step pixel pointer and its fraction
F02318	A1_YSTEP	W	Alternate view of the A1 Y step pixel pointer and its fraction
F0231C	BLIT_COLOR	W	Background color and data path control
F02320	BLIT_TXTD	W	The texture data registers
F02400	BLIT_TCLUT	W	Blitter texture CLUT - 16 words packed into 8 longs
F03000	GPU_RAM	RW	GPU local program and data RAM base, 1024 x 32 bits

F04000	TXT_RAM	RW	Blitter texture RAM, 2048 x 32 bits
F06000	TXT_ROM	RW	Blitter texture ROM, 2048 x 32 bits

To the GPU all these addresses appear as 32 bit locations, and all transfers to them should be long transfers. The only exception to this is the block of 4K bytes of RAM at F03000 (but not the texture RAM), to which the GPU can perform byte and word transfers.

These locations may be accessed by all other processors for read or write as appropriate at the above addresses, via the GPU slave access port, where they appear to the system as 16 bit IO space memory. As they are all actually 32 bits, transfers **must** 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. They are not accessible to the GPU itself at the plus 8000 hex offset.

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.

GPU Flags Register F02100 Read/Write

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

Bit	Name	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	INT_ENA0-4	Interrupt enable bits for interrupts 0-4. The status of these bits is overridden by IMASK. Interrupts are allocated as follows: 4 Blitter 3 Object Processor 2 Timing generator 1 DSP interrupt, the interrupt output from Puck 0 CPU interrupt
9-13	INT_CLR0-4	Interrupt latch clear bits. These bits are used to clear the interrupt latches, which may be read from the status register. Writing a zero to any of these bits leaves it unchanged, and the read value is always zero.
14	REGPAGE	Switches from register bank 0 to register bank 1. This function is overridden by the IMASK flag, which forces register bank 0 to be used.
15	DMAEN	When DMAEN is set, GPU LOAD and STORE instructions perform external memory transfers at DMA priority, rather than GPU priority. This has no effect on program data fetches, which continue at GPU priority. This bit must **not** be changed while an external memory cycle is active. Note that these occur in the background, so be very careful about changing this flag dynamically, and do not modify it in an interrupt service routine.
16	OVERFLOW_FLAG	The ALU overflow flag, which is meaningful for two types of operation: either it means the last add or subtract operation was not representable for signed arithmetic, or it represents the state of the bit set or cleared by the last bit set or clear operation before the bit was set or cleared. Signed arithmetic overflow occurs when: • the sum of two positive numbers gives a negative result • the sum of two negative numbers gives a positive result • a negative number is subtracted from a positive number and gives a negative result • a positive number is subtracted from a negative number and gives a positive result *Note that this bit appears in bit 18 of the RCPU and DSP.*

WARNING - when you write a value to this register, it may not appear to have changed in the following two instructions, because of pipe-lining effects. If you are going to use the flags set by a STORE instruction, or are changing one of the other bits such as the register bank, then ensure that there are two NOPs after the STORE to this register.

Matrix Control Register F02104 Write only

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

Bit	Name	Description
0-3	MWIDTH	Matrix width, in the range 3 to 15
4	MADDW	When set, successive reads of the matrix held in memory are separated by the matrix width. When clear, reads are from consecutive locations.

Matrix Address Register F02108 Write only

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

Bit	Name	Description
2-11	MTXADDR	Matrix address.

Data Organisation Register F0210C Write only

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.

Bit	Name	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 organisation is big-endian. See the discussion elsewhere in this document.
2	BIG_INSTR	Normally, instructions are executed from a long-word in the order low word then high word. When this bit is set the execution ordering is reversed, i.e. high word then low word. However, move immediate data remains little-endian, i.e. the data must always be in the order low word then high word in the instruction stream.
3	BIG_TROUBLE	Under no circumstances set this bit. Data will be swapped in unlikely ways just when you least expect it. Let me tell you what I think of big-endian organisation (continued on page 396)...

GPU Program Counter F02110 Read/Write

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

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

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

GPU Control/Status Register F02114 Read/Write

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

Bit	Name	Description
0	GPUGO	This bit stops and starts the processor. Any processor may write to this register to start it, however only the processor controlled by this bit may clear it (unless single-stepping is enabled).
1	CPUINT	Writing a 1 to this bit allows the GPU to interrupt the CPU. There is no need for any acknowledge, and no need to clear the bit to zero. Writing a zero has no effect. A value of zero is always read.
2	GPUINT0	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	INT_LAT0-4	Interrupt latches. The status of these bits indicate which interrupt request latch is currently active, and the appropriate bit should be cleared by the interrupt service routine, using the INT_CLR bits in the flags register. Writing to these bits has no effect.
11	BUS_HOG	When the GPU is executing code out of external RAM it will normally give up the bus between program fetches, which should allow the CPU to continue to run at the same time. Setting this bit causes the GPU to attempt to hold on to the bus between program fetches, which improves its execution speed, at the expense of any lower priority device using the bus.
12-15	VERSION	These bits allow the GPU version code to be read. Current version codes are: 1 Pre-production test silicon (Jaguar One) 2 First production release (Jaguar One) 3 Pre-production test silicon (Midsummer) 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.
16	ENHANCED	The bit has to be set to enable some of the enhanced functionality of Oberon. The following functions are enabled when this bit is set: • additional condition codes are available to the JUMP and JR instructions • the JRE op-code is enabled, using NOP with non-zero register number fields. • The bug related to two consecutive divides is fixed
17	SCORE_WRITE	When this bit is set, score-board protection is enabled for register writes. This means that if a slow write to a register, such as external load or divide, is followed by a fast write to a register, such as register move, that the writes will be executed in the correct order. This bit should normally always be set, but may be cleared for strict compatibility with Jaguar One.
18	PACK_RGB	When this bit is set the pack and unpack instructions operate on 16 bit RGB data instead of CRY data.
19	RESET	Writing a one to this bits aborts all GPU operation instantly. Everything is cleared down to its power on state, execution halts, the GPU stops completely, and the processor sub-system must be re-initialised from scratch. This bit is very powerful. Writing a zero has no effect. **This bit must never be set in normal operation. It can have catastrophic side effects on other processors, and is only provided as a last resort for fatal situations.**

High Data Register F02118 Read/Write

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

Divide unit remainder F0211C Read only

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.

Divide unit Control F0211C Write only

Bit	Name	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.

GPU DMA Length Counter F02120 Write only

Writing to this counter sets up the length of the transfer and initiates it. The length is written in bytes but must be a whole number of phrases, and the allowable range of values is between 8 (one phrase) and 32760 / 7FF8 hex. Only the GPU is allowed to write to this register if the GPU is running.

GPU DMA Control F02124 Write only

This register controls various aspects of the DMA transfer. It is static.

Bit	Name	Description
0	DMA_OUT	Controls the direction of the DMA transfer. When set transfer is from internal to external memory.
1	DMA_DMA	When this bit is set the transfer occurs at the high GPU bus priority level (DMA level), when clear the transfer occurs at the normal GPU priority level.

GPU DMA External Address F02128 Write only

This register gives the phrase aligned address of external DMA data. It is a counter, and so must be written before each DMA transfer. It wraps within a 4 Mbyte window. **It must not be set to an address within the GPU internal space, this will not work reliably, and may cause unpredictable system crashes.**

GPU DMA Internal Address F0212C Write only

This register gives the long aligned address of internal DMA data, it is also a counter like the external address and so must be written to before each transfer. Valid addresses are in the range F02000 to F05FFC.

GPU DMA Status F02124 Read only

This read port allows the status of the DMA controller to be examined. This register is all zero when the DMA transfer is complete, and this condition should be polled for. Bits are assigned as follows:

Bit	Name	Description
0	OCYCLE	There is a cycle active on the external bus
1	OCYCLERD	The read transfer part of an external cycle is active.
2	DMA_PEND	There is a DMA transfer pending.
3-15	DMA_COUNT	These correspond to the same bits in the DMA length counter, which counts down as each phrase is transferred.
16-31	unused	Will be zero when the DMA is finished.

RISC Central Processor - RCPU

The RISC Central Processor is a Jaguar J-RISC processor intended to act as the CPU for the system, in other words it should be the highest level of program flow in Midsummer. It replaces the function of the 68000 in Jaguar One, and gives considerably more processing power to the role. It is truly the central processor of the system because of its power, the 68000 did not fill that function.

The system is still booted by the 68000, but once running it is intended that the RCPU takes over system management.

The RCPU is intended to be the processor of choice for running code compiled in C, and it should be possible to provide C libraries which largely hide the multiple processor nature of the system.

Cache Controller

Overview

The RCPU contains an instruction cache, which can dramatically improve the speed of execution and reduce the main bus usage of RCPU programs running from external memory. It works by automatically storing instructions in fast local RAM as they are executed, so that when they are executed again they can be fetched immediately from that RAM without waiting for, or slowing down, the external bus. This benefits programs which contain loops (most programs!), as the second and subsequent passes through the loop will be executed entirely from cache.

If the instruction cache controller is disabled the cache instead appears as two banks of local RAM (detailed in the Memory Map section below) which can be used for general program or data storage.

The RCPU also contains a simple mechanism to help with stack operations. This allows the top 512 bytes of data RAM to also be used as a rolling overlay for external RAM, so that the top of the stack can be held in internal memory without severely restricting the stack size. This is not strictly a cache, as it relies on software to copy data in and out of it, but it does allow significant enhancements in stack transfer performance in certain circumstances. This is described in greater detail in the “RCPU Stack Cache Base Pointer” section on page.8

What is a Cache?

A cache is a mechanism to store a copy of parts of slow external memory in fast local memory. It retains a copy of data that is read from external memory, so that if that data is read again then the local copy can be used, saving the time required to fetch it again from the slower memory. In the RCPU the cache stores only program instruction data. Programs are a good candidate for caching as they usually contain loops.

A cache controller contains two blocks of RAM, called **tag RAM** and **data RAM**. The data RAM contains the data that is being cached. In the RCPU these are instructions. The tag RAM identifies where in external memory the data held in the data RAM comes from. To keep tag RAM small, the data RAM is divided into **lines**, where each line has one tag entry. A line holds up to 16 instructions (that is 32 bytes or 4 phrases), and these always correspond to 32 consecutive bytes in external memory, on a 32 byte boundary.

Although it would be nice if any line could correspond to any area in memory, this would require the hardware to check all the tags to determine if there is a cache hit (a hit is when the data being fetched is in the cache). The simplest solution to this is a direct mapped cache, where any location in external memory can only be held in one cache location, so that any address only has to be compared against one tag to see if the correct data is cached. A direct mapped cache has a problem if a loop is jumping back and forth between two locations that correspond to the same cache line (this is known as thrashing), because each will keep over-writing the other, and the stored cache data is never re-used. Therefore the cache in the RCPU is **two-way set associative**. This means that any location in external memory can map onto one of two locations in the data RAM. This means that two tags have to be checked to see if there is a match. If either one of them matches, the cache returns the appropriate data to the instruction fetch unit. This is a reasonable overhead.

The next issue to consider is how the data is written into the cache. When a cache miss occurs, the cache unit will go and fetch an entire line from main memory. If one of the two locations on the data RAM where this data could be stored is empty, than the line is placed there. However, if both of them are filled, then the cache has a decision to make — what should it throw away? The usual answer is to throw away

the least recently filled line. This is normally a good solution, although it can clearly go wrong in the case where three locations that map to the same lines are being cycled through. The RCPU cache supports two algorithms: first in first out (FIFO), where the data that has been in the cache longest is discarded; and random, where the data that is discarded is chosen randomly. You may want to experiment to see which gives you better performance.

Cache Basics

Clearing the ENABLE_CACHE bit in the RCPU Cache Control register disables the cache, making the memory regions RCPU_PRAM and RCPU_TRAM available as regular 32-bit memory. Setting this bit allows these regions to be used by the cache controller, making them inaccessible to the programmer (reading from this region with return indeterminate values, writing will have no effect).

When enabled, the controller uses RCPU_PRAM to store instructions, and RCPU_TRAM to remember where they came from and if they are valid.

RCPU_TRAM will contain random data after reset, and must be cleared before the cache is enabled, so that the controller knows that RCPU_PRAM doesn't contain any valid instructions.

In fact, the cache must be cleared on the following occasions:

1) Before the cache is enabled
2) After modifying code that could be cached, e.g. after :
   a) Loading code from ROM or CDROM
   b) Moving code
   c) Self-modifying code

To clear the cache:

1) Clear the ENABLE_CACHE bit (making RCPU_TRAM accessible to you)
2) Fill the 64 longs in RCPU_TRAM with 0 (The DMA mechanism is the fastest way of doing this)

If this is the first clear after reset, the cache "ignore" range registers CACHE_ILWR, CACHE_IUPR must be initialised. Any instruction fetched from the region:

CACHE_ILWR <= address < CACHE_IUPR

will not be cached. To disable this region, just set CACHE_ILWR=CACHE_IUPR.

Now that the cache has been cleared, it can be enabled by setting the ENABLE_CACHE bit.

All subsequent RCPU instruction fetches will be cached (except those from RCPU_RAM, which doesn't need caching, and those within the 'ignore' area).

The CACHE_HIGH bit can be set at any time to increase the bus arbitration priority of cache fetches.

The next few sections explain in detail how the cache works. Although you can use the cache without this detailed understanding, it will help you achieve maximum performance.

RCPU_CACHECTRL F18130 RW

Bit	Name	Description
0	ENABLE	Clear to disable cache controller, allowing RCPU_PRAM and RCPU_TRAM to be used as general-purpose RAM. Set to enable cache controller, which uses these RAMs for storage. Cache controller's internal 'enable' state cannot change during a pending request or cache burst - this bit can be read to determine the internal state.
1	EN_IMMED	When cache is disabled, this has no effect. When cache is enabled, set to allow 'immediate' cache burst hits. (Explained under 'Cache Performance' below) Do NOT change this bit 'on the fly'. It will normally be set 'TRUE' at start-up then left unchanged.
2	EN_LINE	Similar to EN_IMMED, but enables 'line' cache burst hits.
3	HIPRI	When set, cache fetches are performed at DMA priority.
4	RANDOM	Determines line-replacement algorithm on cache-misses. Clear for FIFO, set for RANDOM.
5	BUS_HOLD	If this bit is set, then the 68000 can never have the bus at its normal priority. This will significantly enhance cache performance when the system is idle as the bus latency will be substantially reduced. Other bus masters will also benefit. However, this prevents the 68000 from running anything except interrupt service routines.
5-31	unused	Write zero.

RCPU_CACHEILO F18134 WO

RCPU_CACHEIHI F18138 WO

These registers define a region of memory as 'un-cached'. Within this region the burst mechanism still functions in order to improve program fetches, but nothing is written to the cache. This may be used to improve performance in some cases by preventing a 'one-off' piece of in-line code from replacing a much-executed loop in the cache. The region is defined as:

RCPU_CACHEILO <= burst_address < RCPU_CACHEIHI

The register-values must be cache-line-aligned, i.e. A4..A0 are zero, corresponding to the 32 bytes in a cache line. To disable the ignore mechanism, set RCPU_CACHEILO=RCPU_CACHEIHI.

Warning: Trying to execute code from the ignored region whilst 'EN_IMMED' and 'EN_LINE' are clear will cause the RCPU to hang.

Cache Organisation

The cache is divided into 'lines'. Each line is 32 bytes long (i.e. it can contain up to 16 instructions) and is always filled in a single burst. The lines are paired into 2-line 'sets', and there are 64 sets in total (the pairing is explained under 'Thrashing' below).

These sets map onto main memory as a 2K block, repeated throughout memory:

Addr	Set
000000	set0
000020	set1
000040	set2
...	...
0007E0	set63
000800	set0
000820	set1
000840	set2
...	...
000FE0	set63
001000	set0
001020	set1
001040	set2
...	...
	(and so on)

Thus each set maps onto many different memory locations, each 2K apart. When the RCPU executes an instruction from a location that is not in the cache (a 'cache miss'), the cache fills the entire set that maps onto that location. Each set also has a 'valid bit' and an 'address tag' which remembers which address that set was filled from, or indeed whether it has ever been filled at all. An example will help make this clear:

Your code jumps to location 0x000800, which happens to be in DRAM. The RCPU requests an instruction from this address, which maps to cache set0 (see above table). The cache checks the tag for set0 against the requested address. But the tag is either 'invalid' (showing that this set has never been filled), or shows that the set contains data from some other address (perhaps 0x000000). This is a cache miss, so the cache requests the external bus and bursts 4 phrases in from DRAM. It also updates the set0 tag to 'valid, 0x000800'.

Your code then continues to execute, fetching the instructions which formed the rest of the burst, and therefore can quickly be supplied from the cache. When location 0x000820 is reached, the same process will occur for set1.

Your code then branches back to 0x000800, which is now contained in set0. For as long as the processor stays within the loop we have just described, it will run entirely from cache.

The tags for each set are held in longs in RCPU_TRAM, organised as follows:

Bit	Name	Description
0-12	TAG0	Base address bits A23-A11 of the instructions stored in Line0
13-25	TAG1	Base address bits A23-A11 of the instructions stored in Line1
26	VALID0	True if Tag0 and Line0 are valid
27	VALID1	True if Tag1 and Line1 are valid

28	LRU	Indicates which line is 'least-recently-used' and will be replaced on next cache-miss at this set.
29-31	unused	Unused by cache.

Thrashing

The RCPU cache is properly defined as 'a 2-way set-associative instruction cache with LRU replacement'. This jargon is an attempt to minimise a nasty cache phenomenon called thrashing.

Imagine that you have a 3D-transform engine running on the RCPU from external DRAM, with much of the code written in 'C'. You clear and enable the cache then jump to the code...

The loop of code processes each point in turn, calling a subroutine to do the actual transform. Normally, turning on the cache will have a huge beneficial effect on this code. After the first time through the loop, both the loop and the subroutine will be in the cache, and subsequent iterations will execute at full speed.

However, if the loop and the subroutine are (some multiple of) 2K apart, they will map onto the same cache set(s). Thus when the loop calls the subroutine, the subroutine replaces it in the cache. When the subroutine then returns to the loop, the loop replaces it in the cache. The cache is thrashing, and isn't helping at all.

This is not quite true, because the cache is very efficient at reading instructions (because it bursts phrases at top speed without relinquishing the bus), so its still better than nothing at all. However, this is obviously an undesirable situation. Especially in 'C' it is hard to keep track of the exact memory-position of routines, and therefore hard to avoid this problem in software.

Hyper-observant readers will have noticed that the cache has 64 sets of 32 bytes each, mapping over a 2K byte area, but when disabled the cache RAM provides 4K bytes of storage. This factor of two discrepancy is because each set consists of two lines.

The two lines both map onto the same 32-byte region (aliased every 2K throughout memory). Whenever a cache miss happens at that set, the cache fills whichever line which was least-recently-used, leaving the other line alone. The 'LRU' bit field in the tag tracks this. In the example above, the 'main loop' miss would fill one line, and the 'subroutine' miss would fill the other. Execution would then continue entirely from cache.

This '2-way set-associativity' means that 3 or more frequently-visited locations must map onto the same set before thrashing occurs. This is unlikely in the critical inner-loops of most algorithms.

Cache Performance

In general, the cache tends to accentuate the characteristics of the RCPU prefetcher and RISC pipeline, i.e. contiguous, in-line code will execute fast, but jumps hurt performance.

Speed

Thrashing (see above) has the single biggest impact on cache-performance, wasting both bus bandwidth and slowing-down the RCPU. Large inner-loops (which exceed 4Kbytes in length) or multiple subroutine calls in inner-loops are the most likely causes.

The cache controller has two mechanisms which optimise processor performance on cache misses:

1. When burst-filling a line, the burst starts from the actual (phrase) address of the wanted instruction, wrapping-around as necessary to complete the set. For example, an instruction-fetch from 0x00001C will fetch phrases in the order:

0x00001C
0x000000
0x000008
0x000010

This speeds execution when the processor jumps into the middle of a cache line, as the processor can continue execution as soon as the first phrase is fetched.

2. During a line-fill, the cache (including the current line) is unavailable to the processor. Burst-fills from ROM especially can take a considerable time to complete, during which the prefetch queue can empty, stalling the processor. To avoid wasting these cycles, a 'Line FIFO' in the cache controller containing 2 longs effectively extends the prefetch queue during bursting. As longs are burst into the cache, they are passed straight to the prefetcher if it is not full, otherwise they are buffered into the FIFO.

Thus as the prefetch queue empties, it can receive the next long from one of three sources:

Immediate hit:	Straight from external memory as it is read-in during a burst.
Line hit:	From FIFO during a burst. Prefetcher was full but has now emptied.
Cache hit:	From cache. Long is already in cache, and no burst is in progress.

Space

If possible, routines should be aligned on line-start (4-phrase) boundaries. If a routine starts at the end of a cache-line (e.g. at address 0x00001E) then all the previous bytes in the line may be wasted, despite having taken time to read in and cache space to store. Compiler-writers might like to provide a command-line flag which forces line-alignment of subroutines (at the cost of increased code-size).

Routines which end at the start of a cache-line (e.g. at address 0x000022) are similarly wasteful, as the whole of the rest of the cache-line may be wasted.

Cache-efficiency 'super-sleuths' can examine the contents of the cache at any time by disabling and dumping the contents of RCPU_TRAM. The 'LRU' bits for each set indicate the line which will be replaced next.

RCPU Memory Map

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

Addr	Name	r/w	Description
F18000	RCPU_REGS	RW	RCPU registers, sixty-four 32 bit locations
F18100	RCPU_FLAGS	RW	RCPU flags
F18104	RCPU_MTXC	W	RCPU matrix control
F18108	RCPU_MTXA	W	RCPU matrix address
F1810C	RCPU_BIGEND	W	RCPU big / little endian control
F18110	RCPU_PC	RW	RCPU program counter
F18114	RCPU_CTRL	RW	RCPU operation control / status
F18118	RCPU_HIDATA	RW	RCPU bus interface high data
F1811C	RCPU_REMAIN	R	RCPU division remainder
F18120	RCPU_DMACNT	W	RCPU DMA transfer count
F18124	RCPU_DMACTL	W	RCPU DMA control register
F18124	RCPU_DMASTAT	R	RCPU DMA status
F18128	RCPU_DMAEA	W	RCPU DMA external address
F1812C	RCPU_DMAIA	W	RCPU DMA internal address
F1E000	RCPU_RAM	RW	RCPU local data RAM base, 256 x 32 bits
F1E800	RCPU_TRAM	RW	RCPU cache tag RAM base, 64 x 32 bits
F1F000	RCPU_PRAM	RW	RCPU cache data RAM base, 1024 x 32 bits

To the RCPU all these addresses appear as 32 bit locations, and all transfers to them should be long transfers. The sole exception is the block of 1K bytes of RAM at F1E000, to which the RCPU can perform byte and word transfer.

These locations may be accessed by all other processors for read or write as appropriate at the above addresses, via the RCPU slave access port, where they appear to the system as 16 bit IO space memory. As they are all actually 32 bits, transfers **must** 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. They are not accessible to the RCPU at the plus 8000 hex offset.

Interrupts

There are six interrupts sources within the RCPU. These are allocated as follows:

5	UART interrupt
4	Video interrupt
3	Object processor CPU interrupt
2	GPU to CPU interrupt
1	Puck interrupt
0	CPU interrupt

Interrupts 2 to 4 are the same interrupt signals connected to the 68000 interrupt controller in Oberon. They are separately available here, for local use or masking. The 68000 interrupt controller does not control these - if one of these interrupts is enabled at source, then all you have to do is to enable them within the RCPU to get the interrupt.

Internal Registers

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

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

RCPU Flags Register F18100 Read/Write

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

Bit	Name	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	INT_ENA0-4	Interrupt enable bits for interrupts 0-4. The status of these bits is overridden by IMASK. Interrupts are allocated as follows: 5 UART 4 Blitter 3 Object Processor 2 Timing generator 1 DSP interrupt, the interrupt output from Puck 0 CPU interrupt
9-13	INT_CLR0-4	Interrupt latch clear bits. These bits are used to clear the interrupt latches, which may be read from the status register. Writing a zero to any of these bits leaves it unchanged, and the read value is always zero.
14	REGPAGE	Switches from register bank 0 to register bank 1. This function is overridden by the IMASK flag, which forces register bank 0 to be used.
15	DMAEN	When DMAEN is set, RCPU LOAD and STORE instructions perform external memory transfers at DMA priority, rather than RCPU priority. This has no effect on program data fetches, which continue at RCPU priority. This bit must **not** be changed while an external memory cycle is active. Note that these occur in the background, so be very careful about changing this flag dynamically, and do not modify it in an interrupt service routine.
16	INT_ENA5	Interrupt enable bit for interrupt 5. Function as bits 4-8.
17	INT_CLR5	Interrupt latch clear bit for interrupt 5. Function as bits 9-13.
18	OVERFLOW_FLAG	The ALU overflow flag, which is meaningful for two types of operation: either it means the last add or subtract operation was not representable for signed arithmetic, or it represents the state of the bit set or cleared by the last bit set or clear operation before the bit was set or cleared. Signed arithmetic overflow occurs when: • the sum of two positive numbers gives a negative result • the sum of two negative numbers gives a positive result • a negative number is subtracted from a positive number and gives a negative result • a positive number is subtracted from a negative number and gives a positive result *Note that this bit appears in bit 16 of the GPU.*
19	EXT_MULT	Enable extended multiplier. Normally multiplies are 16x16 to give a 32-bit result. When this bit is set, multiplies are 16x32, where the destination is the 32-bit operand, giving a 48 bit result. This function does not work with matrix multiplies. The ENHANCED bit must be set for this function to work.
20	EXT_MHIGH	This bit selects the high 32 bits of the multiplier result when performing an extended multiply. The bottom 16 bits are ignored.
21	EXT_MSAT	This bit select the bottom 32 bits of the multiplier result, but saturates them as appropriate. If overflow has occurred into the top 16 bits, then the result is the largest (or smallest for signed negative underflow) representable integer. This works for both signed and unsigned multiplies. signed multiplies: result > 7FFFFFFF, result set to 7FFFFFFF result < 80000000, result set to 80000000 unsigned multiplies: result > FFFFFFFF, result set to FFFFFFFF If this bit is set, then EXT_MHIGH is ignored.

WARNING - when you write a value to this register, it may not appear to have changed in the following two instructions, because of pipe-lining effects. If you are going to use the flags set by a STORE instruction, or are changing one of the other bits such as the register bank, then ensure that there are two NOPs after the STORE to this register.

Matrix Control Register F18104 Write only

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

Bit	Name	Description
0-3	MWIDTH	Matrix width, in the range 3 to 15
4	MADDW	When set, successive reads of the matrix held in memory are separated by the matrix width. When clear, reads are from consecutive locations.

Matrix Address Register F18108 Write only

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

Bit	Name	Description
2-12	MTXADDR	Matrix address, in the range F1E000 - F1FFFC.

Data Organisation Register F1810C Write only

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

Bit	Name	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 organisation is big-endian. See the discussion elsewhere in this document.
2	BIG_INSTR	Normally, instructions are executed from a long-word in the order low word then high word. When this bit is set the execution ordering is reversed, i.e. high word then low word. However, move immediate data remains little-endian, i.e. the data must always be in the order low word then high word in the instruction stream.
3	BIG_TROUBLE	Under no circumstances set this bit. Data will be swapped in unlikely ways just when you least expect it. Let me tell you what I think of big-endian organisation (continued on page 396)...

RCPU Program Counter F18110 Read/Write

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

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

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

RCPU Control/Status Register F18114 Read/Write

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

Bit	Name	Description
0	RCPUGO	This bit stops and starts the processor. Any processor may write to this register to start it, however only the processor controlled by this bit may clear it (unless single-stepping is enabled).
1	CPUINT	Writing a 1 to this bit allows the RCPU 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	RCPUINT0	Writing a 1 to this bit causes a RCPU 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 RCPU 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 RCPU has actually stopped, and should be polled before issuing a further single step command. A one means the RCPU 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	INT_LAT0-4	Interrupt latches. The status of these bits indicate which interrupt request latch is currently active, and the appropriate bit should be cleared by the interrupt service routine, using the INT_CLR bits in the flags register. Writing to these bits has no effect.
11	BUS_HOG	When the RCPU is executing code out of external RAM it will normally give up the bus between program fetches, which should allow the CPU to continue to run at the same time. Setting this bit causes the RCPU to attempt to hold on to the bus between program fetches, which improves its execution speed, at the expense of any lower priority device using the bus.
12-15	VERSION	These bits allow the RCPU version code to be read. Current version codes are: 1 Pre-production test silicon (Jaguar One) 2 First production release (Jaguar One) 3 Pre-production test silicon (Midsummer) Future variants of the RCPU 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 RCPU.
16	INT_LAT5	Interrupt latch for interrupt 5. Has the same function for interrupt 5 as bits 6-10 have for interrupts 0-4.
17	ENHANCED	The bit has to be set to enable some of the enhanced functionality of the RCPU. The following functions are enabled when this bit is set: • additional condition codes are available to the JUMP and JR instructions • the JRE op-code is enabled, using NOP with non-zero register number fields. • The bug related to two consecutive divides is fixed • the software controlled reset function is enabled
18	SCORE_WRITE	When this bit is set, score-board protection is enabled for register writes. This means that if a slow write to a register, such as external load or divide, is followed by a fast write to a register, such as register move, that the writes will be executed in the correct order. This bit should normally always be set, but may be cleared for strict compatibility with Jaguar One.
19	PACK_RGB	When this bit is set the pack and unpack instructions operate on 16 bit RGB data instead of CRY data.
20	RESET	Writing a one to this bits aborts all RCPU operation instantly if the enhanced bit is set. Everything is cleared down to its power on state, execution halts, the RCPU stops completely, and the processor sub-system must be re-initialised from scratch. This bit is very powerful. Writing a zero has no effect. **This bit must never be set in normal operation. It can have catastrophic side effects on other processors, and is only provided as a last resort for fatal situations.**

High Data Register F18118 Read/Write

This 32 bit register provides the high part of RCPU 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.

Divide unit remainder F1811C Read only

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.

Divide unit Control F1811C Write only

Bit	Name	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.

RCPU DMA Length Counter F18120 Write only

Writing to this counter sets up the length of the transfer and initiates it. The length is written in bytes but must be a whole number of phrases, and the allowable range of values is between 8 (one phrase) and 32760 / 7FF8 hex. Only the RCPU is allowed to write to this register if the RCPU is running.

RCPU DMA Control F18124 Write only

This register controls various aspects of the DMA transfer. It is static.

Bit	Name	Description
0	DMA_OUT	Controls the direction of the DMA transfer. When set transfer is from internal to external memory.
1	DMA_DMA	When this bit is set the transfer occurs at the high RCPU bus priority level (DMA level), when clear the transfer occurs at the normal RCPU priority level.

RCPU DMA External Address F18128 Write only

This register gives the phrase aligned address of external DMA data. It is a counter, and so must be written before each DMA transfer. It wraps within a 4 Mbyte window. **It must not be set to an address within the RCPU internal space, this will not work reliably, and may cause unpredictable system crashes.**

RCPU DMA Internal Address F1812C Write only

This register gives the long aligned address of internal DMA data, it is also a counter like the external address and so must be written to before each transfer. Valid addresses are in the range F18000 to F05FFC.

RCPU DMA Status F18124 Read only

This read port allows the status of the DMA controller to be examined. This register is all zero when the DMA transfer is complete, and this condition should be polled for. Bits are assigned as follows:

Bit	Name	Description
0	OCYCLE	There is a cycle active on the external bus
1	OCYCLERD	The read transfer part of an external cycle is active.
2	DMA_PEND	There is a DMA transfer pending.
3-15	DMA_COUNT	These correspond to the same bits in the DMA length counter, which counts down as each phrase is transferred.
16-31	unused	Will be zero when the DMA is finished.

RCPU Extended UART Control F1813C Read/write

This register supplements the ASICTRL register at F10032, and that register must also be initialised before the UART is used. For a full discussion of the UART, refer to the section on it below.

Bit	Name	Description
0	ERROR	When read, this bit indicates that one of the error bits below is set. Writing a one to this bit clears all the error flags. Writing a zero has no effect.
1	BYTE_INT	When this bit is set, the RCPU is interrupted after each byte is received. When this bit is clear, it is interrupted when four bytes have been received.
2	RX_INT	When this bit is set, receiver interrupts are enabled. An interrupt is generated at the rate determined by BYTE_INT. The status of this bit is reflected by a read.
3	TX_INT	When this bit is set, transmitter interrupts are enabled. An interrupt is generated whenever the transmit buffer is empty. The TX_BYTE bit below controls whether this is after one or four bytes. The status of this bit is reflected by a read.
4	NOPAR	When this bit is set, the receiver no longer expects to receive a parity bit. This allows the standard 8-bit, no parity, one stop bit format to be received. It has no effect ion the transmitter, so to transmit this format you should ensure the transmitted parity bit corresponds to a stop bit. This bit also applies to the IO interface. The status of this bit is reflected by a read.
5	TX_BYTE	Set this bit to transmit single bytes. If this is set only the first byte is transmitted. The status of this bit is reflected by a read.
6	RCPU_TRANSMIT	Set this bit if the RCPU is to control the UART transmit interface. If this bit is clear, the normal IO interface controls transmit. The status of this bit is reflected by a read.
7	RCPU_RECEIVE	Set this bit if the RCPU is to control the UART receive interface. If this bit is clear, the normal IO interface controls receive. The status of this bit is reflected by a read.
16	OVERRUN_ERROR	This error flag indicates that the four byte receive buffer has overflowed and receiver data has been lost. This bit is read only.
17	FRAMING_ERROR	This error flag indicates that a framing error occurred on received data. The UART will cease operation until the error is cleared. This bit is read only.
18	PARITY_ERROR	This error flag indicates that received data has a parity error. The UART will cease operation until the error is cleared. This bit is read only.
19-21	BYTES_IN_BUF	This value indicates how many bytes are present in the UART receive data buffer. Valid values are 0-4. Even if the receiver is in byte mode (BYTE_INT set), further values will be added to the buffer until the long overflows. This value is read only.
22-24	BYTES_LAST_READ	This value indicates how many bytes were present the last time the receive data buffer was read. As it is not possible to read the receive data buffer and the BYTES_IN_BUF value atomically, the counter is latched whenever a read occurs and the value stored here.
25	RX_INT_FLAG	The current interrupt was caused by the receiver. This bit is read-only.
26	TX_INT_FLAG	The current interrupt was caused by the transmitter. This bit is read-only.

RCPU UART Data F18140 Read/write

This long location contains a long write-only transmit data buffer, and a long read-only receive data buffer. For a full discussion of the UART, refer to the section on it below. These buffers are big-endian, this means that the byte order of transmission or reception is as follows:

Bits	Order
24-31	first byte
16-23	second byte
8-15	third byte
0-7	fourth byte

If the interface is being operated in byte mode, then the byte should be read from or written to bits 0-7. However, note that if read overflow occurs (which is not flagged as an error in any case until the buffer contains four bytes), then the bytes will be shifted up in the long buffer as they are received. This means that a byte mode RCPU UART receiver actually has nearly four byte times to respond to the interrupt, a truly massive latency were it to ever occur!

RCPU Stack Cache Base Pointer F18144 RW

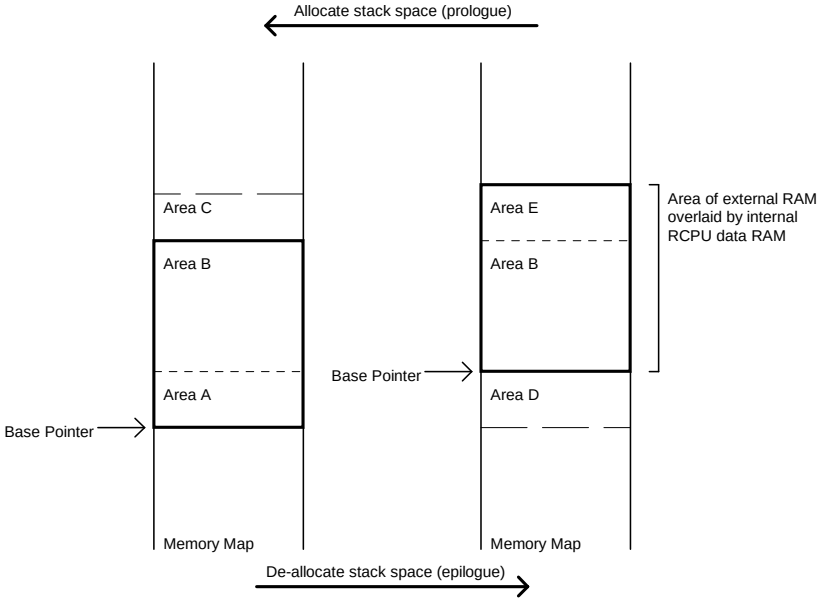
This register points to the long aligned address where the stack cache base lies. This mechanism allows stack caching by mapping the high 512 bytes of the RCPU data RAM to a second location anywhere in memory. Accesses to this address then occur to local RAM, effectively caching the top 512 bytes of the stack. (The bottom 512 bytes of the data RAM contain the interrupt vectors and so all of RAM cannot be used for this).

This 512 byte mapping “rolls” rather than ”slides”, so that location F1E200 always maps to an address whose bottom 9 bits are 0, and so on.

The base pointer must never be set to greater then FFFE00, as the result could be catastrophic.

The base pointer may be on any long-word boundary, allowing considerable flexibility in how this mechanism is used. Compiler writers may choose to allocate more RAM than is needed at a given procedure call by shifting the base pointer to below the bottom of the stack, so that the allocation overhead does not always occur. It may also be necessary to build some hysteresis into the allocation / de-allocation mechanism so that it does not thrash.

When a procedure prologue needs to adjust the stack cache base, it should move the base pointer to its new location, then copy the number of bytes by which the base pointer has moved (Area A) to the correct location in external RAM which has just been uncovered by the cache (Area C). Procedure epilogue code should copy the amount of bytes by which the pointer is to be moved (Area C) from the location in external RAM to the area that is about to be uncovered (Area A), then move the base pointer. Area B is not affected by this operation and need not be moved.



Note that the area overlaid should only be used by the RCPU as other processors cannot see any modifications that have been made within the overlaid area.

Note also that if the DMA mechanism is to be used to copy this data, then it should be pointed at the copy of the RAM at location F1E200, as it cannot access the overlaid address from its internal address pointer.

Digital Sound Processor - DSP

Introduction

The DSP is part of the Puck chip in Jaguar, and is one of the Jaguar RISC processors, which are described above. It uses a very similar instruction set and programming model to the GPU and RCPU, but there are certain differences specific to its role as a sound processor. The DSP has full access to the system memory map as a bus master, and its internal memory may be accessed by the other bus masters within the Jaguar System.

The DSP performs two roles within Jaguar: its primary function is sound synthesis; and it may also be available for additional graphics processing.

Sound synthesis may be the playback of sampled sound 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 output hardware within its internal address space. It also contains hardware specifically for playing back PCM sound samples held in some private DSP memory.

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 system address space. 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.

Memory Map

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	TRI	A triangle wave
F1D200	SINE	A full wave SINE
F1D400	AMSINE	An amplitude modulated SINE wave
F1D600	SINE12W	A sine wave and its second order harmonic
F1D800	CHIRP16	A chirp - this is a sine wave increasing in frequency
F1DA00	NTRI	A triangle wave with noise superimposed
F1DC00	DELTA	A spike
F1DE00	NOISE	White noise

Arithmetic Functions

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 16 bit value, i.e. if it is less than $FFFF8000 it becomes $FFFF8000 and if it is greater than $00007FFF it becomes $00007FFF. SAT32S takes a signed 40 bit operand (see the section below entitled 'Extended Precision Multiply / Accumulates') and saturates it to a signed 32 bit value in a similar manner.

Interrupts

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

5	External interrupt 1
4	External interrupt 0
3	Timer interrupt 1
2	Timer interrupt 0
1	I^2S interface interrupt
0	CPU interrupt

The external interrupts are inputs from additional Jaguar hardware outside the Oberon & Puck system. The timer interrupts are from Puck'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.

Circular Buffer Management

As circular buffers are common in DSP algorithms, for sample-looping, FIFOs, and so on; there is hardware support for addressing circular buffers. These have to 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 2^n. 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.

DSP Flags Register F1A100 Read/Write

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

Bit	Name	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	INT_ENA0-4	Interrupt enable bits for interrupts 0-4. The status of these bits is overridden by IMASK.
9-13	INT_CLR0-4	Interrupt latch clear bits 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	When DMAEN is set, DSP LOAD and STORE instructions perform external memory transfers at DMA priority, rather than GPU priority. This has no effect on program data fetches, which continue at GPU priority. This bit must **not** be changed while an external memory cycle is active. Note that these occur in the background, so be very careful about changing this flag dynamically, and do not modify it in an interrupt service routine.
16	INT_ENA5	Interrupt enable bit for interrupt 5. Function as bits 4-8.
17	INT_CLR5	Interrupt latch clear bit for interrupt 5. Function as bits 9-13.
18	OVERFLOW_FLAG	The ALU overflow flag, which is meaningful for two types of operation: either it means the last add or subtract operation was not representable for signed arithmetic, or it represents the state of the bit set or cleared by the last bit set or clear operation before the bit was set or cleared. Signed arithmetic overflow occurs when: • the sum of two positive numbers gives a negative result • the sum of two negative numbers gives a positive result • a negative number is subtracted from a positive number and gives a negative result • a positive number is subtracted from a negative number and gives a positive result *Note that bit appears in bit 16 of the flags registers of the GPU.*

WARNING - when you write a value to this register, it may not appear to have changed in the following two instructions, because of pipe-lining effects. If you are going to use the flags set by a STORE instruction, or are changing one of the other bits such as the register bank, then ensure that there are two NOPs after the STORE to this register.

DSP Matrix Control Register F1A104 Write only

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

Bit	Name	Description
0-3	MWIDTH	Matrix width, in the range 3 to 15
4	MADDW	When set, successive reads of the matrix held in memory are separated by the matrix width. When clear, reads are from consecutive locations.

DSP Matrix Address Register F1A108 Write only

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

Bit	Name	Description
2-12	MTXADDR	Matrix address, in the range F1B000 - F1CFFC.

DSP Data Organisation Register F1A10C Write only

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

Bit	Name	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_INSTR	Normally, instructions are executed from a long-word in the order low word then high word. When this bit is set the execution ordering is reversed, i.e. high word then low word. However, move immediate data remains little-endian, i.e. the data must always be in the order low word then high word in the instruction stream.

DSP Program Counter F1A110 Read/Write

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 value will be corrupted, as at this point the pre-fetch queue is discarded.

DSP Control/Status Register F1A114 Read/Write

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

Bit	Name	Description
0	DSPGO	This bit stops and starts the DSP. The CPU or DSP may write to this register at any time, but only the DSP should be used to clear this bit (unless single-stepping is enabled).
1	CPUINT	Writing a 1 to this bit allows 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	DSPINT0	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 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	INT_LAT0-4	Interrupt latches for interrupts 0-4. The status of these bits indicate which interrupt request latch is currently active, and the appropriate bit should be cleared by the interrupt service routine, using the INT_CLR bits in the flags register. Writing to these bits has no effect.

11	BUS_HOG	When the DSP is executing code out of external RAM it will normally give up the bus between program fetches. This behaviour should allow the CPU to continue to run at the same time. Setting this bit causes the DSP to attempt to hold on to the bus between program fetches, which improves its execution speed, at the expense of any lower priority device using the bus.
12-15	VERSION	These bits allow the 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 superset of this DSP.
16	INT_LAT5	Interrupt latch for interrupt 5. Has the same function for interrupt 5 as bits 6-10 have for interrupts 0-4.
17	ENHANCED	The bit has to be set to enable some of the enhanced functionality of the DSP. The following functions are enabled when this bit is set: • additional condition codes are available to the JUMP and JR instructions • the JRE op-code is enabled, using NOP with non-zero register number fields. • The bug related to two consecutive divides is fixed • the software controlled reset function is enabled
18	SCORE_WRITE	When this bit is set, score-board protection is enabled for register writes. This means that if a slow write to a register, such as external load or divide, is followed by a fast write to a register, such as register move, that the writes will be executed in the correct order. This bit should normally always be set, but may be cleared for strict compatibility with Jaguar One.
19	unused	
20	RESET	Writing a one to this bits aborts all RCPU operation instantly if the enhanced bit is set. Everything is cleared down to its power on state, execution halts, the RCPU stops completely, and the processor sub-system must be re-initialised from scratch. This bit is very powerful. Writing a zero has no effect. **This bit must never be set in normal operation. It can have catastrophic side effects on other processors, and is only provided as a last resort for fatal situations.**
21	INT_POL	When this bit is set, the polarity of EINT0 and EINT1 to the DSP is reversed. This is an enhanced function, so the enhanced bit must also be set.

Modulo instruction mask F1A118 Write only

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 to 0. This allows addresses to be readily generated for circular buffers of size 2^n bytes, where n is between 0 and 31.

Divide unit remainder F1A11C Read only

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.

Divide unit Control F1A11C Write only

Bit	Name	Description
1	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.

Multiply & Accumulate High Result Bits F1A120 Read only

This 32 bit register allows the high eight bits of the accumulated result to be read. After a RESMAC instruction the result 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.

DSP DMA Length Counter F1A120 Write only

Writing to this counter sets up the length of the transfer and initiates it. The length is written in bytes but must be a whole number of phrases, and the allowable range of values is between 8 (one phrase) and 32760 / 7FF8 hex.. Only the DSP is allowed to write to this register if the DSP is running.

DSP DMA Control F1A124 Write only

This register controls various aspects of the DMA transfer. It is static.

Bit	Name	Description
0	DMA_OUT	Controls the direction of the DMA transfer. When set transfer is from internal to external memory.
1	DMA_DMA	When this bit is set the transfer occurs at the high DSP bus priority level (DMA level), when clear the transfer occurs at the normal DSP priority level.

DSP DMA External Address F1A128 Write only

This register gives the phrase aligned address of external DMA data. It is a counter, and so must be written before each DMA transfer. It wraps within a 4 Mbyte window. **It must not be set to an address within the DSP internal space, this will not work reliably, and may cause unpredictable system crashes.**

DSP DMA Internal Address F1A12C Write only

This register gives the long aligned address of internal DMA data, it is also a counter like the external address and so must be written to before each transfer. Valid addresses are in the range F1A000 to F1DFFC.

DSP DMA Status F1A124 Read only

This read port allows the status of the DMA controller to be examined. This register is all zero when the DMA transfer is complete, and this condition should be polled for. Bits are assigned as follows:

Bit	Name	Description
0	OCYCLE	There is a cycle active on the external bus
1	OCYCLERD	The read transfer part of an external cycle is active.
2	DMA_PEND	There is a DMA transfer pending.
3-15	DMA_COUNT	These correspond to the same bits in the DMA length counter, which counts down as each phrase is transferred.
16-31	unused	Will be zero when the DMA is finished.

Private Memory Interface and PCM Processor

The DSP has some external memory which no other processor can use. This memory is on a private bus so that the DSP can access it without using the main 64-bit bus. It is intended for storing PCM sound samples, so that a multiple voice sampled sound generator may be implemented without any main bus overhead. This private memory may be either DRAM or ROM (SRAM, EEPROM or FLASH are also possible in ROM mode), and may be either 4 or 8 bit. The exact type has not yet been selected, and the hardware is capable of supporting all these types.

You can access this private memory in one of two ways.

1. DSP load and store instructions may access it in the address range E00000h to EFFFFFh.
2. A simple PCM list processor can fetch sound sample data from it.

Because of this, the DSP cannot perform loads or stores to E00000 to EFFFFF on the main bus.

PCM List Processor

The PCM List Processor is a data transfer engine within the DSP whose function is to fetch sample table data from the DSP private memory, and place the sample data in DSP internal memory. It can read 8 or 16 bit samples, it can decompress 8-bit samples, and it can handle interpolation and looping.

The PCM list processor is a little like the object processor in that it reads a list of sample descriptions and uses them to fetch the sample data. It does not have any means of branching, unlike the object processor. Its list only needs modifying when a sample is to be started, stopped, or when a looping sample has to be modified. It is normally triggered at sample rate by an interrupt process.

Each time it is triggered, the PCM list processor executes a linear list of PCM sample descriptions in DSP local RAM, and fetches the sample data from the private memory. A special sample with a table size of minus one flags the end of the table. The PCM processor has the following features for playing samples

- variable rate sample play (pitch shifting)
- interpolation between successive samples
- sample looping or stop at the end of each sample table
- automatic advance of the sample pointers, so that the only software overhead is firing it off for each set of samples, and summing the resultant sample values
- support for eight and sixteen bit samples, and for eight bit μ-law compressed samples
- samples may be played forwards or backwards

A sample description has the following format:

Long	Name	Bits	Description
1	table size	0-19	subtracted from pointer when looping, if this all ones, then this sample description is the end of the table
2	table end address	0-19	used to detect end of sample
	μ-law flag	27	sample is 8-bit μ-law data
	backwards flag	28	the sample is being played backwards (negative rate) so this reverses the end of table test
	word size flag	29	sample is 16 bit words, not 8 bit bytes
	interpolate flag	30	interpolate between sample and next one
	loop flag	31	loop at the table end address, instead of stopping
3	rate, 20.12 bits	0-31	added to pointer after each fetch (19.13 bits for word samples)
4	pointer, 20.12 bits	0-31	pointer to sample data (19.13 bits for word samples)
5	read value	0-31	sign-extended output sample data

These sample structures should be linearly packed in DSP RAM on a long boundary.

Interpolation uses the most significant six bits of the sample pointer to linearly interpolate between the sample pointed at and the one above it in memory. It uses a 6 x 16 bit multiplier, so the sample is used at full precision, but there is only six bits of precision in the interpolation control value. This should be enough for most purposes.

Compressed Samples

Sixteen bit samples are the ideal way in which to store audio data. However, they are large and so there is sometimes a need to store them in a more compact form. Much audio data can be stored as eight bit samples. These can sound reasonable, but they are not so useful for audio data with a wide dynamic range. The PCM list processor therefore supports compressed samples. These use an eight bit value which is a non-linear representation of a sixteen bit sample. This gives a reasonable resolution both for quiet and loud sounds.

Compressed samples for the PCM list processor use μ-law compression to increase the dynamic range of 8-bit samples. μ-Law is widely used in digital audio.

The mu law function is:

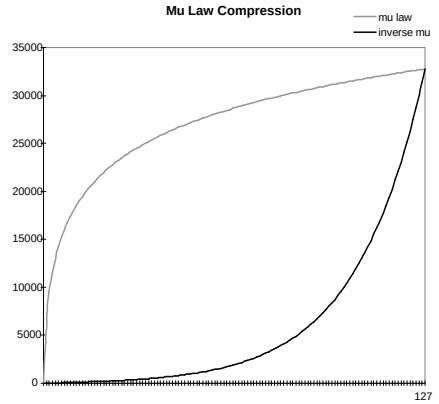
$$y = y\max\left(\frac{\left(\ln\left(1+\mu\left(\frac{|x|}{x\max}\right)\right)\right)}{\ln(1+\mu)}\right)$$

The inverse μ-law function is:

$$x = x\max\left(\frac{e^{\left(\left(\frac{y}{y\max}\right)\ln(1+\mu)\right)}-1}{\mu}\right)$$

where $y\max = 32767$, $\mu = 255$, $x\max = 127$

This gives a function close, but not identical to, an exponential function. Graphically, it appears thus:



The 8-bit sample data is treated as a sign and magnitude number, the sign is stored in the top bit, and the 7-bit magnitude component is used to index a look-up table, which gives a 13-bit de-compressed value. The sign is restored by complementing the output if required, giving a decompressed value in the range 32764 to -32764.

The actual μ-law expansion table is:

```
0000 0004 000C 0010 0018 001C 0024 002C 0034 003C 0044 004C 0058 0060 006C 0074
0080 008C 0098 00A4 00B0 00C0 00CC 00DC 00EC 00FC 010C 0120 0134 0144 015C 0170
0184 019C 01B4 01D0 01E8 0204 0220 0240 0260 0280 02A4 02C8 02EC 0314 033C 0368
0394 03C0 03F4 0424 045C 0490 04CC 0508 0548 0588 05D0 0618 0664 06B0 0704 0758
07B4 0810 0874 08D8 0944 09B4 0A28 0AA4 0B20 0BA8 0C30 0CC4 0D5C 0DF8 0EA0 0F4C
1000 10BC 1180 1250 1328 1408 14F0 15E8 16E8 17F0 1908 1A2C 1B5C 1C9C 1DE8 1F44
20B0 222C 23B8 2554 2704 28C8 2AA0 2C8C 2E90 30AC 32DC 3528 378C 3A0C 3CA8 3F64
4240 4538 4854 4B94 4EFC 5288 563C 5A1C 5E28 6260 66C8 6B64 7038 7540 7A80 7FFC
```

Note that bits 0 and 1 are set to zero on all these values.

DSP PCM List Pointer F1A130 Write only

This register holds the PCM control list pointer. This needs to be written each time the control list is to be executed, and the action of writing this pointer initiates list execution. This pointer must be an address in DSP internal RAM only, i.e. it must be in the range F1B000 - F1CFFF, and must be long aligned.

You can read the current list position in the PCM status port.

Bit	Name	Description
0-1	unused	
2-14	List Pointer	The valid part of the address.
15-31	unused	

DSP PCM Control Register / Status Port F1A134 Read/Write

This register defines the type of memory attached to the private memory interface, and allows certain status bits of the PCM control unit to be read. It should only be written by the machine initialisation software, and not by application code.

Bit	Name	Description
0	DRAM	If this bit is set, the attached memory is DRAM, which requires a multiplexed address, and row/column control signals. If this bit is clear, the attached memory is ROM or SRAM, and is addressed directly.
1	NYBBLE	If this bit is set, the private memory has a four bit data bus. If this bit is clear, the private memory has an eight bit data bus.

2-4	PTIM	If DRAM is set, then this gives the DRAM pre-charge time. The RAS control line will be high for at least 1 + PTIM clock cycles between any pair of row cycles. A value of 0 is treated as 8. If DRAM is clear, then this gives the ROM cycle time. Chip select is active for 3 + PTIM clock cycles (OE goes active 2 clock cycles after chip select). A value of 0 is treated as 8.
5-7	RTIM	If DRAM is set, then this gives the RAS active time of a refresh cycle, RAS goes low for RTIM clock cycles, CAS goes low one clock cycle before RAS, to give CAS-before-RAS refresh operation. A value of 0 is treated as 8.
8-11	REFRATE	This gives the rate of refresh cycles. These are performed at the rate of one refresh every 64 x (REFRATE + 1). A REFRATE of 0 disables refresh, and 0 should be set if DRAM is not set.
12	IDLE	Read-only; set when the PCM list processor is not active. This bit may be polled after a write to the list pointer to determine when the list has been processed.
13	NEG_LAT	The PMEMLAT output is normally an active high address latch (for latching the bottom eight bits of the address when DRAM is clear). If this bit is set, it becomes an active low signal.
13-15	unused	Write zero.
16-28	List Pointer	Read-only; the current value of the PCM list pointer may be read here, so that you can determine how much of the list has been executed.
29-31	unused	Write zero.

In the DSP, the Synchronous Serial Interface is mapped into the internal DSP space for higher efficiency when the DSP is controlling it. These are effectively 32 bit locations. They are described elsewhere in this document. In summary, they are:

SCLK	**Serial Clock Frequency**	**F1A150**	**WO**
SMODE	**Serial Mode**	**F1A154**	**WO**
LTXD	**Left transmit data**	**F1A148**	**WO**
RTXD	**Right transmit data**	**F1A14C**	**WO**
LRXD	**Left receive data**	**F1A148**	**RO**
RRXD	**Right receive data**	**F1A14C**	**RO**
SSTAT	**Serial Status**	**F1A150**	**RO**

Blitter

This section describes the Oberon Blitter.

What is the Blitter?

Blitter is an abbreviation for *bit block transfer*. It purpose is to process blocks of bits or pixels, by filling them in with a color or copying them from another block. 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 higher level graphics operations, whilst the Blitter performs the low-level repetitive pixel-by-pixel operations in parallel.

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

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

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

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

The *tour de force* of the Blitter is its ability to generate Gouraud shaded or textured 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 texture addresses, Z and intensity gradients, and the polygon vertices. The blitter will then draw a triangle in a single operation. 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 full account of how the Blitter works. These are useful for reference, but for an introduction to how to use the Blitter see the discussions further on, and in associated documentation and examples.

Blitter Register Set

The following is a list of all the registers in the blitter:

Addr	Name	Type	Status	Description
F02200	A1_BASE	WO	static	A1 base address. This is the lowest address of the rectangle of pixels which A1 points at.

F02204	A1_FLAGS	WO	static	A1 control flags. These determine various aspects of the pixel block pointed at by A1, such as pixel size, block width, et cetera.
F02208	A1_CLIP	WO	static	A1 clipping window size. The gives the X and Y size of a window for A1 to clip to. X and Y pointer values outside this window will not cause write cycles if this function is enabled.
F0220C	A1_PIXEL	RW❷	dynamic†	A1 pixel pointer. This points at the pixel where the blit starts. It is in pixel co-ordinates, and has sixteen bit X and Y parts. An alternate (and more logical) view of this register is provided by the A1_X and A1_Y registers.
F02210	A1_STEP	WO❷	dynamic†‡	A1 step integer part. The step value may be added to the pixel pointer after each pass through the inner loop. An alternate view of this register is provided by the A1_XSTEP and A1_YSTEP registers.
F02214	A1_FSTEP	WO❷	dynamic†‡	A1 step fractional part. The fractional parts of the register above. An alternate view of this register is provided by the A1_XSTEP and A1_YSTEP registers.
F02218	A1_FPIXEL	RW❷	dynamic†	A1 pixel pointer fractional part. This gives the fractional bits of the pixel pointer. An alternate view of this register is provided by the A1_X and A1_Y registers.
F0221C	A1_INC	WO	static	A1 increment integer part. The increment value may be added to the pixel pointer after each pixel is drawn.
F02220	A1_FINC	WO	static	A1 increment fractional part. This is the fractional part of the register above.
F02224	A2_BASE	WO	static	A2 base address. As A1.
F02228	A2_FLAGS	WO	static	A2 control flags. As A1.
F0222C	A2_MASK	WO	static	A2 window address mask. This is used to give a bit-wise mask of the pixel pointer. This causes the pointer to wrap within a pre-defined rectangle.
F02230	A2_PIXEL	WO	dynamic	A2 pixel pointer. As A1.
F02234	A2_STEP	WO	dynamic	A2 step integer part. As A1.
F02238	BLIT_CMD	WO❷	dynamic	Blitter command register. Control bits here control what operation the blitter performs, and a write to this register initiates blitter operation.
F02238	BLIT_STAT	RO	N/A	Blitter status register. Allows the blitter to be polled for completion and status.
F0223C	BLIT_CNT	WO❷	dynamic†	The blitter inner and outer loop counter values. These control the size of the blit operation.
F02240	BLIT_SRCD	WO	static unless SRCEN, SRCENX or GOURD	Source data, or computed intensity fractional parts. The data registers are all sixty-four bit locations.
F02248	BLIT_DSTD	WO	static unless DSTEN	Destination data.
F02250	BLIT_DSTZ	WO	static unless DSTENZ	Destination Z.
F02258	BLIT_SRCZ1	WO	static unless SRCENZ or GOURZ	Source Z1, or computed Z integer parts.
F02260	BLIT_SRCZ2	WO	static unless SRCENZ or GOURZ	Source Z2, or computed Z fractional parts.
F02268	BLIT_PATD	WO	static unless GOURD	Pattern data, or computed intensity integer parts.
F02270	BLIT_IINC	WO❷	static	Intensity increment
F02274	BLIT_ZINC	WO❷	static	Z increment
F02278	BLIT_STOP	WO	static	Collision stop control.
F0227C	BLIT_I0	WO	static unless GOURD §	Initial intensity 0. These four registers are alternative views of the pattern and source data registers, and each corresponds to a 16.16 bit initial intensity value. #
F02280	BLIT_I1	WO	static unless GOURD §	Initial intensity 1.
F02284	BLIT_I2	WO	static unless GOURD §	Initial intensity 2.
F02288	BLIT_I3	WO	static unless GOURD §	Initial intensity 3.
F0228C	BLIT_Z0	WO	static unless GOURZ §	Initial Z 0.These four registers are alternative views of the source Z registers, and each corresponds to a 16.16 bit initial Z value. #
F02290	BLIT_Z1	WO	static unless GOURZ §	Initial Z 1.
F02294	BLIT_Z2	WO	static unless GOURZ §	Initial Z 2.
F02298	BLIT_Z3	WO	static unless GOURZ §	Initial Z 3.
F0229C	BLIT_FINNER	WO❷	dynamic†	The low sixteen bits of this register corresponds to the fractional part of the inner counter, and the high bits are the extended command bits.
F022A0	BLIT_IDELTA	WO❷	dynamic†	This 16.16 bit register is the value that may be added to the inner counter initial value after each pass through the inner loop.
F022A4	A1_XSD	WO❷	static	This 16.16 bit register contains the A1 X step delta value.
F022A8	A1_YSD	WO❷	static	This 16.16 bit register contains the A1 Y step delta value.
F022AC	BLIT_ISTEP	WO❷	dynamic†	Intensity step value. This 16.16 bit value may be added to the intensity values after each pass through the inner loop.
F022B0	BLIT_ISD	WO❷	static	Intensity step value delta. This 16.16 bit value may be added to the intensity step value after each pass through the inner loop.
F022B4	BLIT_ZSTEP	WO❷	dynamic†	Z step value. This 16.16 bit value may be added to the Z values after each pass through the inner loop.
F022B8	BLIT_ZSD	WO❷	static	Z step value delta. This 16.16 bit value may be added to the Z step value after each pass through the inner loop.
F022BC	BLIT_X0	WO	dynamic §	Texture X address pointer 0. These are 16.16 bit values. #
F022C0	BLIT_X1	WO	dynamic §	Texture X address pointer 1
F022C4	BLIT_X2	WO	dynamic §	Texture X address pointer 2
F022C8	BLIT_X3	WO	dynamic §	Texture X address pointer 3
F022CC	BLIT_Y0	WO	dynamic §	Texture Y address pointer 0 #
F022D0	BLIT_Y1	WO	dynamic §	Texture Y address pointer 1
F022D4	BLIT_Y2	WO	dynamic §	Texture Y address pointer 2
F022D8	BLIT_Y3	WO	dynamic §	Texture Y address pointer 3
F022DC	BLIT_XINC	WO❷	static	Texture X inner loop increment. This is the amount added to the X pointer after each pixel is drawn in the inner loop.
F022E0	BLIT_XSTEP	WO❷	dynamic†	Texture X outer loop step
F022E4	BLIT_XSD	WO❷	static	Texture X outer loop step delta. This is added to the step value on each pass through the outer loop.
F022E8	BLIT_YINC	WO❷	static	Texture Y inner loop increment. This is the amount added to the Y pointer after each pixel is drawn in the inner loop.

F022EC	BLIT_YSTEP	WO❷	dynamic†	Texture Y outer loop step. When POLYGON is set this is the initial texture X pointer.
F022F0	BLIT_YSD	WO❷	static	Texture Y outer loop step delta. This is added to the step value on each pass through the outer loop.
F022F4	BLIT_TBASE	WO❷	static	Texture base address
F022F8	BLIT_IINCX	WO❷	static	Alternate intensity increment register, this is an 11.16 bit value which affects only the intensity field.
F022FC	A1_MASK	WO	static	A1 window address mask. This is used to give a bit-wise mask of the pixel pointer. This causes the pointer to wrap within a pre-defined rectangle.
F02300	A2_CLIP	WO	static	A2 clipping window size. The gives the X and Y size of a window for A2 to clip to. X and Y pointer values outside this window will not cause write cycles if this function is enabled.
F02304	A1_X	WO❷	dynamic†	This gives an alternate view of the X portions of the A1 pixel pointer and its fractional parts, and allows these to be written as a single 16.16 bit integer.
F02308	A1_Y	WO❷	dynamic†	This gives an alternate view of the Y portions of the A1 pixel pointer and its fractional parts, and allows these to be written as a single 16.16 bit integer.
F0230C	A2_X	WO	dynamic	The bottom 16 bits of this register give an alternate means if initialising the X portion of the A2 pixel pointer. (These are not the same as A1)
F02310	A2_Y	WO	dynamic	The bottom 16 bits of this register give an alternate means if initialising the Y portion of the A2 pixel pointer.
F02314	A1_XSTEP	WO❷	dynamic†‡	This gives an alternate view of the X portions of the A1 step value and its fractional parts, and allows these to be written as a single 16.16 bit integer.
F02318	A1_YSTEP	WO❷	dynamic†‡	This gives an alternate view of the Y portions of the A1 step value and its fractional parts, and allows these to be written as a single 16.16 bit integer.
F0231C	BLIT_COLOR	WO❷	static	This allows the CRY color fields of the pattern data to be updated as a single operation. Bits eight to fifteen of this register will update the color field of all four pixels in the pattern data register. Note that this is double-buffered unlike the pattern data itself. This register also specifies the static mixing colour for mixing with texture using Gouraud intensity to control the mix. When used for this the colour is specified in bits zero to fifteen. See below. Mixer control bits are specified here.
F02320	BLIT_TXTD	WO	static unless TEXTEN	The texture data registers may be written to allow some mixing effects without reading texture data. This is a sixty-four bit register.
F02400	BLIT_TCLUT	WO	static	Texture CLUT (16 words)
F04000	TXT_RAM	RW	static	Blitter texture RAM, 2048 x 32 bits
F06000	TXT_ROM	RW	non-volatile	Blitter texture ROM, 2048 x 32 bits

❷ These registers are *double-buffered*. This means that they may be written to while the blitter is still active performing the previous blit operation. The buffer which is written to is transferred into the main register when the next blitter command is written.

WO These registers are write-only. They may only be written to, their contents are not visible.

RO These registers are read-only. They may be read from, but not modified.

RW These registers may be written to and read from. Their contents may be modified as the result of blitter operation.

† Although these registers are dynamic, i.e. they are modified by the blitter, the double-buffer will restore them automatically to the previously written value, so they may be treated as static if you wish the same initial value to be re-used on the next blit.

‡ These registers are only dynamic if POLYGON is set.

§ These registers are not normally written to as part of a blit operation, as the blitter can initialise them automatically if DATINIT is set.

0 is aligned with bit 0 and so is the right-most for big-endian system (like the Jaguar console), and the left-most for a little endian system. You may therefore wish to consider the register at the highest address to be register 0 for a big-endian system.

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.

Windows

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

The address itself is generated from a window pointer. This has an X and Y value, and again is in pixels. The pointer may point to areas outside the window, and both pointers support 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 describe a raster scan over a rectangle. Along a scan line, the pointer may be updated either by one pixel or to the next phrase boundary, depending on how the Blitter is currently operating. Refer to the Data Path section for further details.

At the end of a scan line, the pointer is updated by a step value, which is the distance in X and Y to the start of the next scan line. This action of scan across the block, then step to the next start, is controlled by

the Blitter's inner and outer control loops, the inner loop traversing a scan line, and the outer loop adding the step value. Thus the inner loop length is the block width, and the outer loop length the block height.

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

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

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

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

Data Path

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

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 only pixels of eight bits or larger can be processed. The source and destination address pointers do not need to be aligned to the same point in a phrase, the Blitter will automatically align the source to the destination. If two source phrase must be read before a destination phrase can be written, then the SRCENX 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 to restore the destination data area when only parts of it are updated.

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

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

Write Data

Write data may come from any of the following sources:

- the pattern data register
- the logic function unit
- computed Gouraud shaded data
- texture unit data
- shaded texture data
- the data adder

The default is the LFU output. The ADDDSEL flag selects the data adder output, PATDSEL selects the pattern register output of the pattern data multiplexer. The PDSEL bits then determine if the texture data, pattern data or the output of the multiplicative mixing unit, which is used for shading texture data, is selected.

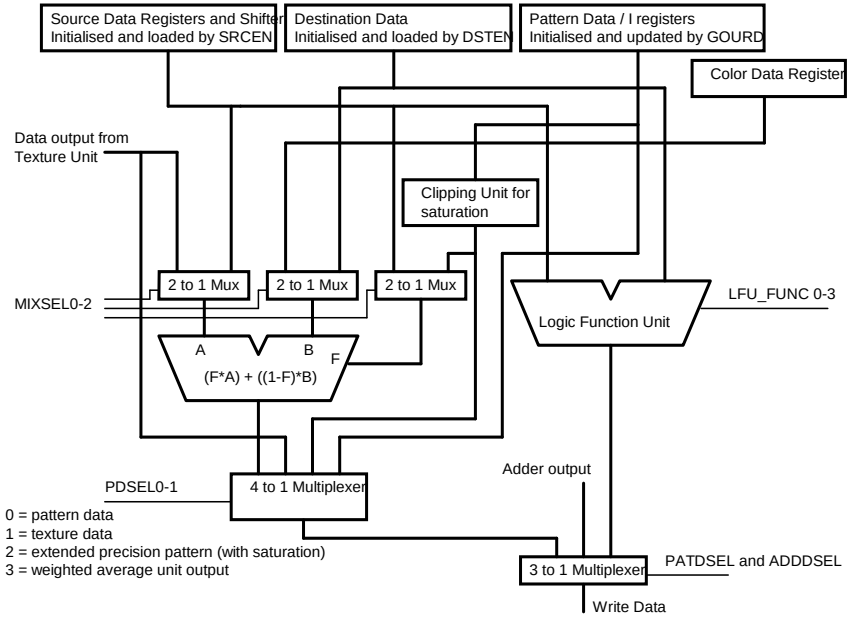
Write Z may come from

- 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.

The data paths which control the generation of write data are given in the simplified diagram below. This shows the precedence of the various control bits.



The clipping unit on the pattern data takes the extended precision intensity data fields and clips those to eight bits.

The multiplicative mixing unit performs the weighted average function shown. The A and B inputs, and the mixing control input F, may all be switched between different inputs to allow a variety of modes. These include:

1. Mix texture data with a background color using the computed intensity to control the mix
2. Do multiplicative Gouraud shading by mixing two colours, the texture data and the color register, according to the computed intensity. This allows shading in CRY or RGB16 modes, and offers a choice between shading to black or shading to white.
3. Translucent or anti-aliased edge texture mapping by mixing texture with the destination, with the source data providing a mix map (this technique is also known as alpha shading, and the source data is the alpha buffer).
4. Mixing two images from the source and destination fields, using the intensity to control the mix (the intensity is constant in this mode).
5. Computed translucency, mixing the texture data with the destination according to the computed intensity.

The adder output is the sum of the source and destination data, and is selected as the output data by setting ADDDSEL.

This diagram does not show the adders used for performing the Gouraud data and Z-buffer calculation, the Z-data path, or the output mask control..

Data Comparators

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

- The bit comparator. This is used for bit to pixel expansion, and selects a bit or group of bits from the source data register, using a counter which is cleared every time the inner loop is entered. The bit is then used to control whether a pixel is written at the current location.
- The Z comparator. This is used in 16 bit pixel mode to compare the 16 bit un-signed integer Z attribute of a pixel on the screen, 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 the 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 destination data to be written back. If this has been previously read then the data will be left unchanged, if not then a background colour can be used, stored in the destination data register
- The action of the Blitter can be stopped altogether. This may be used for collision detection, searching, etc.

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

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 clock cycle overhead when turning round from a read to a write transfer.

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

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

Controlling State Machines

These state machines are the “program” that the blitter runs. You can think of them as two nested `for` loops, the outer loop is generally passed though once for each scan line of the blit, the inner loop is passed through once for each pixel or each phrase of the blit.

The state diagrams may look hard to follow, and you probably need never examine them in any detail, but they are here to give an absolute reference to what the blitter does and in what order it does it. What is useful is the discussion of each state and what happens in it. The effect of some of the more useful control bits, such as POLYGON and DATINIT, is made a lot clearer by understanding how they affect these control loops.

The blitter is quite a complex piece of hardware, and it is hard to present exactly how it works in a concise manner, but this section should give some insight into what is going on.

Outer Loop

The Blitter outer loop control state machine is represented by this pseudo-code:

```
idle:             Blitter is idle, and will not perform any bus activity
if GO     if DATINIT goto init_if
          else      goto inner

init_if:          Initialise intensity fractions and texture X
goto      init_ii

init_ii:          Initialise intensity integers and texture Y
if GOURZ  goto init_zf
else      goto inner

init_zf:          Initialise Z fractions
goto init_zi

init_zi:          Initialise Z integers
goto inner

inner:            Run inner loop state machine (asserts step from its idle state)
if INDONE if OUTER0 goto idle
          else      if UPDA1F goto a1fupdate
                    else      if UPDA1  goto a1update
                              else      if GOURZ.POLYGON goto zfupdate
                                        else      if UPDA2  goto a2update
                                                  else      if DATINIT goto init_if
                                                            else      restart inner

a1fupdate:        Update A1 pointer fractions and more (see below)
goto a1update

a1update:         Update A1 pointer integers
if GOURZ.POLYGON goto zfupdate
else      if UPDA2  goto a2update
          else      if DATINIT goto init_if
                    else      restart inner

zfupdate:         Update computed Z step fractions
goto zupdate

zupdate:          Update computed Z step integers
if UPDA2  goto a2update
else      if DATINIT goto init_if
                    else      restart inner

a2update:         Update A2 pointer
if DATINIT goto init_if
else      restart inner

init_if:          Initialise 4 intensity fraction fields and 4 texture X pointers
goto init_ii

init_ii:          Initialise 4 intensity integer fields and 4 texture Y pointers
if GOURZ goto init_zf
else      restart inner

init_zf:          Initialise 4 Z fraction fields
goto init_zi

init_zi           Initialise 4 Z integer fields
restart inner
```

The outer loop state machine fires off the inner loop, and controls the updating process between passes through the inner loop. States have functions as follows:

idle	Blitter is off the bus, and no activity takes place.
inner	Inner loop is active, read and write cycles are performed
a1fupdate	A1 step fraction is added to A1 pointer fraction POLYGON true: A1 step delta X and Y fraction parts are added to the A1 step X and Y fraction parts (the value prior to this add is used for the step to pointer add). POLYGON true: inner count step fraction is added to the inner count fraction part POLYGON.GOURD true: the I fraction step is added to the computed intensity fraction parts † POLYGON.GOURD true: the I fraction step delta is added to the I fraction step
a1update	A1 step is added to A1 pointer, with carry from the fractional add POLYGON true: A1 step delta X and Y integer parts are added to the A1 step X and Y integer parts, with carry from the corresponding fractional part add (again, the value prior to this add is used for the step to pointer add). POLYGON true: inner count step is added to the inner count, with carry POLYGON.GOURD true: the I step is added to the computed intensities, with carry † POLYGON.GOURD true: the I step delta is added to the I step, with carry the texture X and Y step delta values are added to the X and Y step values.
zfupdate	the Z fraction step is added to the computed Z fraction parts † the Z fraction step delta is added to the Z fraction step
zupdate	the Z step is added to the computed Zs, with carry † the Z step delta is added to the Z step, with carry
a2update	A2 step is added to the A2 pointer
init_if	Initialise the fractional part of the computed intensity fields, from the increment and step registers. The texture X integer and fractional parts can also be initialised.
init_ii	Initialise the integer part of the computed intensity, and texture Y integer and fractional parts..
init_zf	Initialise the fractional part of the computed Z fields.
init_zi	Initialise the integer part of the computed Z fields.

† — these functions are irrelevant if the DATINIT function is enabled, which it will normally be.

All these states will complete in one clock cycle, with the exception of the idle state, which means the blitter is quiescent; and the inner state, which takes as long as is required to complete on strip of pixels. It is therefore possible for the blitter to spend a maximum of nine clock cycles of inactivity between passes through the inner loop.

Inner Loop

The blitter inner loop state machine is represented by this pseudo-code:

```
idle:               Inactive, blitter is idle or passing round outer loop
if STEP   if SRCENX goto sreadx
          else      if TXTEXT goto txtread
                    else      if SRCEN  goto sread
                              else      if DSTEN  goto dread
                                        else      if DSTENZ goto dzread
                                                  else      goto dwrite

sreadx:             Extra source data read
if STEP   if SRCENZ goto szreadx
          else      if TXTEXT goto txtread
                    else      if SRCEN  goto sread
                              else      if DSTEN  goto dread
                                        else      if DSTENZ goto dzread
                                                  else      goto dwrite

szreadx:            Extra source Z read
if STEP   if TXTEXT goto txtread
          else      goto sread

txtread:            Read external texture data
if STEP   if SRCEN  goto sread
          else      if DSTEN  goto dread
                    else      if DSTENZ goto dzread
                              else      goto dwrite

sread:              Source data read
if STEP   if SRCENZ goto szread
          else      if DSTEN  goto dread
                    else      if DSTENZ goto dzread
                              else      goto dwrite

szread:             Source Z read
if STEP   if DSTEN  goto dread
          else      if DSTENZ goto dzread
                    else      goto dwrite

dread:              Destination data read
if STEP   if DSTENZ goto dzread
          else      goto dwrite

dzread:             Destination Z read
if STEP   goto dwrite

dwrite:             Destination data write
if STEP   if DSTWRZ goto dzwrite
          else      if INNER0 goto idle
                    else      if TXTEXT goto txtread
                              else      if SRCEN  goto sread
                                        else      if DSTEN  goto dread
                                                  else      if DSTENZ goto dzread
                                                            else      goto dwrite

dzwrite:            Destination Z write
if STEP   if INNER0 goto idle
          else      if TXTEXT goto txtread
                    else      if SRCEN  goto sread
                              else      if DSTEN  goto dread
                                        else      if DSTENZ goto dzread
                                                  else      goto dwrite
```

States have functions as follows:

idle	Another state in the outer loop is active. No bus transfers are performed.
sreadx	Extra source data read at the start of an inner loop pass.
szreadx	Extra source Z read as the start of an inner loop pass.
txtread	Read texture data from external memory. This state is **only** used for external texture. TEXTEXT is the condition TEXTMODE=1.
sread	Source data read.
szread	Source Z read.
dread	Destination data read.
dzread	Destination Z read.
dwrite	Destination write. Every pass round the inner loop must go through this state..
dzwrite	Destination Z write.

The step signal is a composite decode that indicates that the inner loop may start, or that the underlying memory interface has completed and the blitter may advance to the next memory transfer.

Register Description

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

Address Registers

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

A1 Base Register F02200 Write only

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

Flags Register F02204 Write only

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

Bit	Name	Description
0-1	Pitch	The distance between successive phrases of pixel data in the window data structure. Gaps may be used to provide alternate pixel maps for double-buffering, for Z data, and for other control information. The distance between two successive phrases of pixels is given by two to the power of this value, with one special case; i.e. a pitch of 0 means pixel data phrases are contiguous, 1 means 1 phrase gaps, 2 means 3 phrase gaps; but 3 means 2 phrase gaps, which 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	Pixel size	The pixel size, where the actual pixel size is 2^n, n is the value stored here. Values 0-5 are allowed.
6-8	Z offset	This value gives the offset from a phrase of pixel data of its corresponding Z data in phrases. Values of 0 and 7 are not used.
9-14	Width	This width is distinct from the width in pixels stored in the window register, and is the width used for address generation. The width is a six bit floating point value in pixels, with a four bit unsigned exponent, and a three bit mantissa, whose top bit is implicit, and which has the point after the implicit top bit. This is similar to the IEEE single precision format without the sign bit. It must give a whole number of phrases in the current pixel size. For example, a screen width of 640 encodes as 1.01×2^9, where 1.01 is a binary number. This gives an exponent field of 9, i.e.1001, and a mantissa field of (1)01. This is stored thus: Bit 14 13 12 11 10 9 E3 E2 E1 E0 M1 M0 1 0 0 1 0 1
15	Mask	Enables Boolean AND masking of the A1 pointer by its window register.
16-17	X add ctrl.	These control the update of the X pointer on each pass round the inner loop. Values are: 0 Add phrase width and truncate to phrase boundary (sets phrase mode), note that you must only use this for eight bit or larger pixels 1 Add pixel size, effectively add one 2 Add zero 3 Add the increment
18	Y add ctrl.	This bit controls how the Y pointer is updated within the inner loop. It is overridden by the X control bits if they are in add increment mode. 4 Add zero 5 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.

21	Step load	When this bit is set, the step register is loaded directly into the pointer on each pass round the outer loop, instead of being added to it. This is used for polygon drawing mode. The UPDA1 flag must be set, the UPDA1F flag will usually be set too.

A1 Clipping Window Size F02208 Write only

This register contains the size in pixels, and may be used for clipping writes, so that if the pointer leaves the window bounds no write is performed. The width is an unsigned fifteen bit value in the low word, the height an unsigned fifteen bit value in the high word. The top 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 corner at the window origin, then the window base register should be modified to make it the top left corner of the clip rectangle.

In pixel mode clipping will occur when the A1 pointer goes outside the clipping window if A1 is being used as either the source or the destination pointer. In phrase mode, however, clipping will only work correctly when A1 is the destination pointer (DSTA2 is not set).

A1 Window Mask F022FC Write only

This register is used as the window size only in the sense that it may be used to AND mask the pointer register when the Mask flag is set. This causes the address to wrap within a rectangular area and may be used to give fill patterns.

A1 Window Pixel Pointer F0220C Read/Write

This register contains the X (low word) and Y (high word) pointers onto the window, and are the location where the next pixel will be written. They are sixteen bit signed values. 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 0-32767 and Y values in the range 0-4095 will produce valid addresses from the address generator, values outside this range are for clipping purposes only.

A1 Step Value F02210 Write only

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

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

If the step load bit is set in the A1 flags register, then this register is loaded into the pointer in the outer loop (when the UPDA1 bit is set) instead of being added to it.

A1 Step Fraction Value F02214 Write only

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

If the step load bit is set in the A1 flags register, then this register is loaded into the fractional parts of the pointer in the outer loop (when the UPDA1F bit is set) instead of being added to it.

A1 Window Pixel Pointer Fraction F02218 Read/Write

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

A1 Pixel Pointer Increment F0221C Write only

The increment is added to the pointer value within the inner loop when the address update is in add increment mode. This register contains the two 16 bit signed integer parts of the increment, the X part is in the low word, the Y part in the high word.

A1 Pixel Pointer Increment Fraction F02220 Write only

This is the fractional parts of the increment described above.

A1 X Step Delta F022A4 Write only

This register holds the 16.16 bit value which may be added to the A1 step X value on each pass through the outer loop if the POLYGON and UPDA1 and UPDA1F bits are set.

A1 Y Step Delta F022A8 Write only

This register holds the 16.16 bit value which may be added to the A1 step Y value on each pass through the outer loop if the POLYGON and UPDA1 and UPDA1F bits are set.

A2 Base Register F02224 Write only

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

A2 Flags Register F02228 Write only

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

Bits	Name	Description
0-1	Pitch	As A1.
2	unused	
3-5	Pixel size	As A1.
6-8	Z offset	As A1.
9-14	Width	As A1.
15	Mask	Enables Boolean AND masking of the A2 pointer by its window register.
16-17	X add ctrl.	These control the update of the X pointer on each pass round the inner loop. Values are: 0 Add phrase width (truncate to phrase boundary), note that you may only use this for eight bit or larger pixels 1 Add pixel size (effectively add one) 2 Add zero
18	Y add ctrl.	This bit controls how the Y pointer is updated within the inner loop. 3 Add zero 4 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.
21	Step load	When this bit is set, the step register is loaded directly into the pointer on each pass round the outer loop, instead of being added to it The UPDA2 flag must be set.
22	CLIP_A2	Enables clipping when the A2 pointer lies outside its window boundaries. This has the effect of inhibiting destination writes within the inner loop, but Blitter operation will continue. This is similar to the CLIP_A1 function in the command register.

A2 Clipping Window Size F02300 Write only

This register contains the size in pixels, and may be used for clipping writes, so that if the pointer leaves the window bounds no write is performed. The width is an unsigned fifteen bit value in the low word, the height an unsigned fifteen bit value in the high word. The top 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 corner at the window origin, then the window base register should be modified to make it the top left corner of the clip rectangle.

In pixel mode clipping will occur when the A2 pointer goes outside the clipping window if A2 is being used as either the source or the destination pointer. In phrase mode, however, clipping will only work correctly when A2 is the destination pointer (DSTA2 is set).

A2 Window Mask F0222C Write only

This register is used as the window size only in the sense that it may be used to AND mask the pointer register when the Mask flag is set. This causes the address to wrap within a rectangular area and may be used to give fill patterns.

A2 Window Pointer F02230 Read/Write

This register contains the X (low word) and Y (high word) pointers onto the window, and are the location where the next pixel will be written. They are sixteen bit signed values. 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 0-32767 and Y values in the range 0-4095 will produce valid addresses from the address generator, values outside this range are for clipping purposes only.

A2 Step Value F02234 Write only

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

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

Control Registers

Command Register F02238 Write only

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

Bit	Name	Description
Bits 0-5 enable corresponding memory cycles within the inner loop. Destination write cycles are always performed (subject to comparator control), but all other cycle types are optional.		
0	SRCEN	Enables a source data read as part of the inner loop operation.
1	SRCENZ	Enables a source Z read as part of the inner loop operation. This bit is ignored unless SRCEN is set.
2	SRCENX	Enables an "extra" source data read at the start of an inner loop operation. This is necessary where data has to be re-aligned, and may also sometimes be of use in bit-to-pixel expansion. If SRCENZ is set an extra Z read is also performed.
3	DSTEN	Enables a destination data read as part of inner loop operation. This must always be performed for pixels smaller than 8 bits, where part of the destination data write will need to restore the data that was previously there.
4	DSTENZ	Enables a destination Z read as part of inner loop operation.
5	DSTWRZ	Enables a destination Z write as part of inner loop operation.
6	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	NOGO	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 clock cycle overhead per update.		
8	UPDA1F	Add the fractional part of the A1 step value to the fractional part of the A1 pointer between inner loop operations in the outer loop.
9	UPDA1	Add the A1 step value to the A1 pointer between inner loop operations in the outer loop.
10	UPDA2	Add the A2 step value to the A2 pointer between inner loop operations in the outer loop.
11	DSTA2	Reverses the normal roles of the address registers from A1 as destination and A2 as source to A2 as destination and A1 as source.
12	GOURD	Enable Gouraud shaded data updates within inner loop, i.e. the intensity gradient fractional part, repeated four times, is added to the computed intensity fraction register (a.k.a. destination data), then the intensity gradient integer part is added with the carry from the previous add to the computed intensity value register (a.k.a. pattern data).
13	GOURZ	Enable polygon Z data updates within the inner loop, i.e. add Z fractions to the Z fraction register (source Z 2), then add with carry the Z integer part to the Z integers (source Z 1).
14	TOPBEN	Enable carry into the top byte of the intensity integers in Gouraud data updates (leave clear for CRY mode).
15	TOPNEN	Enable carry into the top nibble of the intensity integers in Gouraud data updates (leave clear for CRY mode).
Bits 16-17 select alternative write data – the default source is the Logic Function Unit, whose output is controlled by the LFUFUNC bits.		
16	PATDSEL	Select pattern data as the write data.
17	ADDDSEL	Selects the sum of 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 applies to 16 bit pixels. *This function is now largely obsolete and should not be generally used, the multiplicative data mixer and the texture unit offer more general ways of combining images.*
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	LFUFUNC	The bits control the data produced by the logic function unit. The output is the Boolean OR of the following minterms: bit 0 AND (NOT source) AND (NOT destination) bit 1 AND (NOT source) AND destination bit 2 AND source AND (NOT destination) bit 3 AND source AND destination for example, source data is selected by setting 1100 (destination terms cancel out), or the XOR of source and data is 0110.
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 when 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 when 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	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 data read from the source is modified, so source data should be selected using the LFU as the write data. This is particularly intended for performing flat shading on texture mapped surfaces. *This function is now obsolete and should NOT be used, use the sum of the texture data stream and the intensity/pattern values to flat shade or Gouraud shade textures.*

31	POLYGON	Enables blitter polygon drawing mode. The outer loop gains a variety of additional arithmetic operations to prepare the blitter for the next scan line of the polygon. See the discussion of polygon drawing below.

Status Register F02238 Read only

Bit	Name	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	PENDING	A double-buffered blitter command is pending and the previous blit is still in operation, so another blitter command cannot yet be written.
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.

Inner and Outer Counters Register F0223C Write only

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

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

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

Inner Count Fraction & Extended Control F0229C Write only

The low 16 bits of this register give the fractional part of the inner counter reload value. This is used in polygon mode, when the inner count can change by a non-integer amount on each scan line. The high sixteen bits are used to extend the blitter command set.

Bit	Name	Description
0-15	FINNER	The fractional part of the inner counter reload value.
16	DATINIT	Enables the initialisation of all four of the I and Z data fields from the I and Z step register and increment values. The step value is taken to be the value of the first pixel in the scan line, and the increment is added to it or subtracted from it to fill in all four 16 bit pixel fields in the intensity integer and fraction registers, and in the Z integer and fraction fields. If GOURZ is not set only the I field is initialised. This function can also apply to texture mapping.
17-18	TEXTMODE	Texture mapping control modes: 0 Disable texture mapping unit 1 Fetch textures from external memory (TEXTEXT) 2 Fetch textures from a single internal texture map 3 Fetch textures from a duplicated internal texture map (TEXTDBL) In modes 2 & 3 texture fetches are from internal memory even if the texture base address is not in internal memory.
19	INTERP	Enable the texture interpolation unit for anti-aliased textures.
20-22	TEXTXS0-2	Texture width, encoded as follows: 0 32 pixels 1 64 pixels 2 128 pixels 3 256 pixels 4 512 pixels 5 1024 pixels 6 2048 pixels
23-25	TEXTYS0-2	Texture height, encoded in the same way as the texture width.
26	TEXTNIB	Texture is packed as nibbles, not as words. It is expanded by a small blitter palette.
27	TEXTMIR	When this bit is set, textures are mirrored as they wrap. For example, if the texture width is 32 pixels, then when bit 5 of the X pointer is set, the texture X address is two's complemented.
28	TEXTRGB	This bit flips the interpolation unit and multiplicative data mixers between CRY and RGB16 modes. When this is set, the pixels are interpolated and mixed as RGB, if clear then they are CRY. This is used in conjunction with the INTERP bit when anti-aliasing textures.
29	EXT_INT	Extended precision intensity calculations. When this bit is set all intensity calculations are performed at 11.16 bit precision as signed numbers, and the saturation is performed as the pixel values are output. See notes below.
30-31	PDSEL0-1	These bits select what appears as pattern data in the data unit. The output of this is selected when the PATDSEL bit is enabled. Functions are as follows: 0 Pattern data 1 Texture data 2 Extended precision pattern data (use this if EXT_INT is set instead of 0) 3 Multiplicative mix of texture data and the color register (or destination data) with the saturated extended precision intensity controlling the mix.

Inner Counter Delta F022A0 Write only

This value is added to the inner counter reload value after each pass through the inner loop if POLYGON and A1UPDATE are both set. It is a 16.16 bit value. The inner counter reload value has a fractional part when this is enabled, the integer part is in the low word of the counters register and the fraction part is in the low word of the inner counter fraction and extended control register. This odd arrangement is for historical reasons.

To avoid ragged right hand edges on the polygon, the fractional part of the A1 X pointer is also added to the reload value before the counter itself is loaded. This only occurs when POLYGON is set.

Data Registers

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

Source Data Register F02240 Write only

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

Destination Data Register F02248 Write only

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.

Destination Z Register F02250 Write only

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

Source Z Register 1 F02258 Write only

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

Source Z Register 2 F02260 Write only

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

Pattern Data Register F02268 Write only

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

Intensity Increment F02270 Write only

This thirty-two 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.

Z Increment F02274 Write only

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

Collision control and Mode F02278 Write only

This registers allows the Blitter to be stopped when an inner loop write inhibit occurs. Blitter stop will occur in painting 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.
3	FIX_BUGS	This bit should be set to fix the following bugs, which are enabled by default for compatibility reasons: • The A1 ADDY control bit affected both address registers. It affects only A1 and the A2 one now has the required effect when fixed. • Incorrect masking of the phrase which corresponds to the right hand edge of the A1 clip window.
4	INT_DBUF	When this bit is set, the blitter interrupt will occur when the double buffer can accept another command. This means that after the first blitter command is written, an interrupt will be generated immediately, and the double-buffer may be filled, interrupts will then occur each time the double buffer is emptied. When this bit is clear, the blitter interrupt occurs when blitter operation has completed.

Intensity 0 F0227C Write only
Intensity 1 F02280 Write only
Intensity 2 F02284 Write only
Intensity 3 F02288 Write only

These four registers provide an alternate view of the computed 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.

Z 0 F0228C Write only
Z 1 F02290 Write only
Z 2 F02294 Write only
Z 3 F02298 Write only

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).

Intensity Step F022AC Write only

This register gives the step value for intensity. This is either added to the intensity values in the outer loop, or may be used, if DATINIT is set, to reload the four computed intensity values, suitable modified by the increment value. See the discussion on polygons below.

Intensity Step Delta F022B0 Write Only

This register is added to the intensity step value on each pass through the outer loop, the POLYGON, GOURD, UPDA1 and UPDA1F bits all have to be set for this to operate correctly.

Z Step F022B4 Write only

This register gives the step value for Z. This is either added to the Z values in the outer loop, or may be used, if DATINIT is set, to reload the four computed Z values, suitable modified by the Z increment value. See the discussion on polygons below.

Z Step Delta F022B8 Write Only

This register is added to the Z step value on each pass through the outer loop, the POLYGON, GOURZ, UPDA1 and UPDA1F bits all have to be set for this to operate correctly.

Color Data and Data Path Control F0231C Write Only

This double buffered register allows the CRY color fields of the pattern data to be updated as a single operation. Bits eight to fifteen of this register will update the color field of all four pixels in the pattern data register. Note that this is double-buffered unlike the pattern data itself.
This register also specifies the static mixing colour for mixing with texture using Gouraud intensity to control the mix. When used for this the colour is specified in bits zero to fifteen, and can be either CRY or RGB16.

The higher bits of this register control the blitter data path, as follows.

Bit	Name	Description
0-7	COLOR0-7	Specifies the low 8 bits of the background color value
8-15	COLOR8-15	Specifies the top 8 bits of the background color value, and the provide a double-buffered means of initialising the CRY color fields of the pattern data registers for Gouraud shading.
16	MIXSEL0	Controls the input to operand A of the multiplicative data mixer. When this bits is clear texture data is selected, when set the source data is selected.
17	MIXSEL1	Controls the input to operand B of the multiplicative data mixer. When this bit is clear the background color is selected (repeated four times over the phrase), when this bit is set the destination data is selected.
18	MIXSEL2	Control the input to the mix control input F of the multiplicative data mixer. When this bit is clear the saturated extended precision intensity fields are selected, when it is set the bottom byte of each word of the source data is used as the mix control.

Texture Unit Control Registers

Texture X address pointer 0-3	**F022BC-C8**	**Write Only**
Texture Y address pointer 0-3	**F022CC-D8**	**Write Only**

These four register pairs correspond to the four X and Y pointers required to simultaneously fetch four texture map source pixels when generating a phrase of destination data. If phrase mode is not set, the only pointer 0 is used, unless the texture is being anti-aliased in which case all four are required to index the four corners of the square.

In texture mapped polygon generation, these registers will not normally be written to, as they are automatically initialised from the texture X and Y step registers. Set POLYGON and DATINIT.

These are 11.16 bit values, allowing textures up to 2048 pixels on a side. The address pointers wrap if they overflow or underflow.

Texture X increment	**F022DC**	**Write Only**
Texture Y increment	**F022E8**	**Write Only**

This register pair give the amount added to the texture X and Y pointers after each pass through the inner loop. They therefore control the rotation and scaling occurring within the inner loop. Remember that the rotation and scaling are applied to the source, so that making these values smaller will increase the magnification of the texture.

When the blitter is running in phrase mode, then the value written here should be **four times** the pixel increment, because each address pointer has to be advanced four pixels.

These are 11.16 bit values.

Texture X step	**F022E0**	**Write Only**
Texture Y step	**F022EC**	**Write Only**

When DATINIT is not set, this register pair give the step added the pointer in the outer loop.

However, normally it will be set, in which case the name step is a misnomer. When DATINIT is set this is the value used to initialise the texture address pointers at the start of each pass through the inner loop. The step delta is added to these registers after each pass through the inner loop, so that these values follow a line down the left hand edge of the polygon.

These are 11.16 bit values.

Texture X step delta	**F022E4**	**Write Only**
Texture Y step delta	**F022F0**	**Write Only**

These registers give the value added to the step registers on each pass round the outer loop, if UPDA1 is set. These registers should give the gradient, in texture space, of the “left” hand side of the polygon in destination terms.

Modes of Operation

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

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

Block Moves

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

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

The length of the block should be stored in the inner counter - the number represents the number of pixels, so the largest block that can be copied is 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 address flags registers.

Having set these, a command is stored in the command register, with the SRCEN bit set to enable source reads, and the LFUFUNC bits set to 1100 to select source data. 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 pixel terms, is given by the X pointer plus the width times the Y pointer; a multiply operation is necessary to compute the address. To avoid the need for a hardware multiplier in the Blitter address generator, the width is rather strangely encoded.

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

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

Value	Binary	Floating-point	Encoded
20	000000010100	1.01 x 2^4	0100 01
80	000001010000	1.01 x 2^6	0110 01
128	000010000000	1.00 x 2^7	0111 00
640	001010000000	1.01 x 2^9	1001 01
3584	111000000000	1.11 x 2^11	1011 11

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

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

Clipping may be performed by the A1 address generator, and simply prevents writes occurring at addresses outside the window boundaries, i.e. X or Y either negative or grater than the window size. The window size is programmed in the A1 window size registers. This is not much faster than writing the clipped pixels, so if a large number of pixels are to be clipped then it is worth performing the clipping at a higher level.

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 bit per pixel modes.

The bit selection counter is reset every time the inner loop is left, so bit packed data patterns may be up to 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 enables 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 the curved surface by computing the intensity at each vertex, using a vertex normal, and some suitable illumination model. The vertex intensity is then linearly interpolated across the polygon edges, and the edge intensities are linearly interpolated across the polygon scan lines.

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

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

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

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

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

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

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

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

Address registers are set up as follows:

```
A1_BASE          0x01600000        The window base address
A1_PITCH         1                 Pixel data and Z data alternate
A1_PSIZE         4                 16 bit pixels
A1_ZOFFS         1                 Z data is one phrase up from pixel data
A1_WIDTH         0x11              20-pixel window: 1.01 x 2^4 = 0100 01
A1_ADDC          0                 Add one phrase to address
A1_WIN_X         20                Window width
A1_WIN_Y         5                 Window height
A1_PTR_X         1                 First pixel at address 0,1
A1_PTR_Y         0
```

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

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

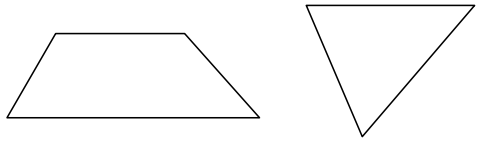
Control information is set up as follows:

```
Inner count      18                Strip width
Outer count      1                 Single pixel high strip
DSTEN            1                 Read destination data, to restore if necessary
DSTENZ           1                 Read destination Z, to compare with computed Z
DSTWRZ           1                 Write destination Z, restoring or replacing
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.

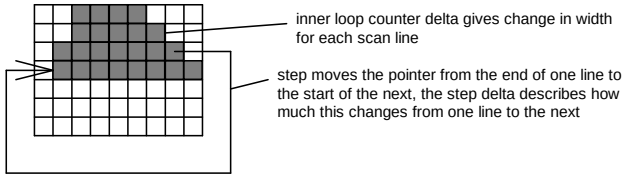
Polygon Drawing

Midsummer can draw polygons as a single blit. These polygons must be three or four sided, and can have only two sides which are not horizontal.



Polygon types that can be drawn in one blit

These polygons are drawn by allowing the inner loop counter to be modified by a delta value after each pass round the inner loop. The inner loop counter initial value and its inner loop count delta value are both 16.16 bit numbers. To re-initialise the pixel address pointer, the step value itself requires a step delta, which is added to it after each pass round the inner loop, thus:



To allow these polygons the be Gouraud shaded, the I and Z values require a step value, and that step value must have a step delta, in the same manner as the pointer. Texture source address X and Y values will also have a step and step delta value, as well as their increment values.

Gouraud Shading

This polygon blitting mode may also be applied to shaded polygons. However it is largely restricted to blitting triangles in this mode, because it is only for triangles that the intensity gradient does not change from one scan line to the next. This is also true for Z and texture address values. The constant gradient is not in fact true if perspective maths is involved, but is a good approximation. (See the discussion of perspective in texture mapping.)

Address Generation

Address generation for the polygon is handled by the A1 address generator. The address pointer is set up as normal to point at the first pixel of the polygon to be drawn. The inner counter and its fractional part are set up to give the width of the first row of pixels. The width of the second row of pixel should also be calculated, and the difference between this and the first loaded into the inner counter delta register.

A1 will normally be set to phrase mode for this operation, to give maximum speed. As this leaves the pointer on a phrase boundary at the end of a scan line rather than on a pixel boundary, the step values would not vary linearly as the polygon is drawn. Therefore the A1 step load bit in the flags register should be set. This causes the outer loop A1 updates to load the pointer directly from the step register (and its fractional part), rather than using the step register to modify the pointer. This means that the step register becomes the pointer initial value. The step delta registers then give the amount by which the step value should be modified each line. Normally, the X step will be set to the gradient of the left hand edge of the polygon, and the Y step set to one.

The UPDA1 and UPDA1F bits have to both be set for the step register function to operate correctly.

Data Handling

To allow the data for Gouraud shading and Z-buffering to be reset on each scan line, the I and Z values both now have step and step delta values. However, the data ALU values also suffer from the same problem as the address, i.e. scan lines end on a phrase boundary and not a pixel boundary, so the variation in the step values required to reset them to the start of the next scan line does not vary linearly.

To remove this problem, and to improve the programmer's model of the blitter generally, a new arithmetic unit in the blitter data section comes into play when the DATINIT bit is set. This causes an ALU to pre-calculate the offsets required for the four pixels that make up a phrase, taking the step value as the initial pixel value, and the increment as the offset. For big-endian operation the four possible initial conditions are:

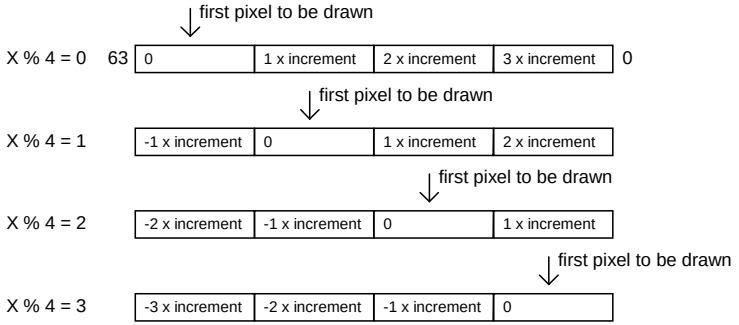


Table of initialization offsets for pixels.

The initialisation unit takes the I and Z increment values, divides them by four, then scales them and adds them to or subtracts them from the step values as shown. This is done for both the integer and fractional parts. The X value used for this is the A1 X pointer.

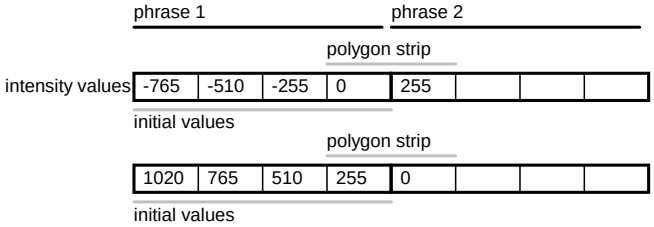
This function is not restricted to the blitter polygon drawing mode, and may also be used when polygons are being drawn strip by strip, as on Jaguar One, to save processing overhead.

Intensity and Colour Values

Midsummer implements extended precision intensity calculations. These should now be used in preference to the old Jaguar One 8.16 bit Jaguar values. Intensity is now treated as a signed 11.16 bit number, when the EXT_INT bit is set for extended precision intensity calculation. This mode must be used when using the polygon draw mode and Gouraud shading is being performed.

The intensity registers are extended to this precision, and there is an extended intensity increment register, which contains only the 11.16 bit intensity increment. Note that if the extended intensity increment register is used, the old intensity increment register should be initialised to zero once.

The extended precision intensity is not saturated when the addition of the increment is performed, the saturation is performed when the data is output. Saturation treats the 11 bit intensity value as a signed number, and if it is negative outputs zero, otherwise if it is greater than 255 it outputs 255. If overflow of the 11 bit value occurs then errors may occur, but this should not happen in normal operation. 11 bits was chosen as the precision as the most extreme case possible is drawing a two-pixel wide strip of a polygon, where the two pixels straddle a phrase boundary, as shown below:



The requirement in this situation is for the data initialiser to be able to set up appropriate intensity values on the left-most pixel value of the first phrase, so that when the intensity increment is added the correct value is present in the left-most pixel of the second phrase.

Note that there is not an equivalent extension to the range of the Z buffer values, and if your program has to deal with the sort of cases shown above for intensity for Z as well, then you will have to reduce the precision of the Z used to prevent overflow or underflow occurring in the initialisation process.

Polygon Control Flags

The main control flag for polygon drawing is the POLYGON control flag in the blitter command register. When this is set the following functions are enabled:

- The A1 step delta fractional parts are added to the A1 step fraction values in the A1 update fraction cycle between passes through the inner loop. This requires the UPDA1F flag to be set.
- The A1 step delta values are added to the A1 step values in the A1 update cycle between passes through the inner loop. This requires the UPDA1 flag to be set. If the UPDA1F flag is set carry occurs from the fraction to the integer parts as required.
- The inner counter reload value is modified by the inner counter delta value on each pass through the inner loop. This requires both the UPDA1 and UPDA1F flags to be set for correct operation. To avoid a potential rounding error, the fractional part of the A1 X pointer is added to the inner loop reload value before it is loaded into the inner counter itself.
- The I and Z values are modified in the outer loop. See the discussion of the outer loop states under “Controlling State Machines” above.

You will almost always want to set the DATINIT function so that the I and Z values, and the texture X and Y pointers, are initialised for each scan line from the step register, as discussed above.

The A1 step load function is also normally set for polygon operations, particularly in phrase mode. This means that the A1 step values define the start pixel for each scan line, and the A1 step delta values describe the gradient of the left hand side. Note that both the A1 step value and the A1 pointer should be initialised with the first pixel address at the start of the blit under these circumstances.

Texture Mapping

Texture Mapping involves mapping from a bit-map image onto a surface which has been rendered in 3D. It is used as a technique to make surfaces more “interesting” and realistic than just flat or Gouraud shaded surfaces. It can be used to simply make surfaces textured, rough or grubby; or it may be used to map a realistic appearance, e.g. brickwork on a wall, or rivets on a ship.

The texture must be mapped onto the surface so that it always appears in the same position on an object as the object moves in 3D. This requires rotation, scaling, skewing and perspective transformation. The first three functions transform a linear traverse across the target (i.e. one polygon scan line) into a linear traverse across the source image. Unfortunately, the perspective transform does not, and would require a divide at each pixel, so is therefore not feasible within the blitter. The distortions due to not performing the perspective transform properly are not particularly objectionable, and can be reduced by sub-dividing a polygon into smaller pieces.

Memory Considerations

Texture mapping on the original Jaguar system was fastest when done with both the source and destination in DRAM. However, as it had to be done pixel by pixel, it used the DRAM in pretty well the worst way imaginable. Because the source and destination lay in different DRAM banks, a row address overhead was performed for each read and write.

Midsummer is capable of texture mapping from internal memory. A texture generation unit computes four X and Y pairs for a phrase, in much the same way as it currently computes I and Z values. This can fetch values from internal memory in parallel with external bus activity. If the texture is duplicated in internal memory, then it can perform two fetches per clock cycle, so that the blitter can do texture-mapped write cycles at the maximum bus rate, i.e. a phrase of texture mapped pixels every two clock cycles.

However, the internal memory resource is severely limited in size. The blitter can therefore also perform these texture fetches out of external DRAM. Again this can now be performed a phrase at a time, but as the four source pixels are not contiguous, four reads are required for each write cycle. However, these will not necessarily all incur the row address overhead, because if they all lie in the same DRAM page, then only the first read will incur the overhead. At the time this document was written, all current systems have 2048 byte pages, so that if the whole texture is this small then four arbitrary pixels will always lie in the same page. However, even if the source texture is larger than this, it is likely that most of the time the four pixels will be sufficiently local to each other to lie in the same page.

Let us examine how long texture mapping takes. On Jaguar One, to texture map four 16 bit pixels took:

4 x (row change + source read + read to write delay + row change + destination write)

which is 4 x (3 + 2 + 1 + 3 + 2) = 44 clock cycles

On Midsummer, texture mapping four 16 bit pixels when all four source pixels lie in the same DRAM bank, which will the typical case, takes:

row change + (4 x source read) + read to write delay + row change + destination write

which is 3 + (4 x 2) + 1 + 3 + 2 = 17 clock cycles

The very worst case is if every fetch causes a row change — this will be exceptional, and this takes:

4 x (row change + source read) + read to write delay + row change + destination write

which is 4 x (3 + 2) + 1 + 3 + 2 = 29 clock cycles

Texture mapping from internal RAM will be four clock cycles per destination write of four pixels, with the texture duplicated in internal RAM it takes two clock cycles per four pixels.

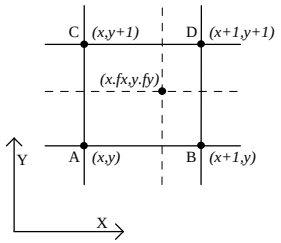
Clearly texture mapping from internal texture memory is very attractive, but the restrictions on internal memory size, will mean that it will not always be possible. You may wish to consider blitting each texture into internal memory before they use it. Certainly this will be worthwhile if the texture is used several times over.

Note that the blitter will perform one texture source data read for each destination pixel. If the texture is being significantly expanded then there will be a significant inefficiency here as the same source pixel is repeatedly re-read. This overhead will be far less significant for on-chip textures, and it may well prove worth-while to transfer some textures into internal RAM before using them to avoid this overhead.

Anti-Aliased Texture Mapping

Aliasing is a major quality problem on texture mapped surfaces. When they are expanded, the squares that make up the source pixels become very obvious. When the texture is compressed, bright spots can twinkle as the texture is moved or scaled, depending on whether or not they are sampled. The blitter implements a mode where anti-aliasing is performed over a two pixel by two pixel square. This helps a lot with the scaling up problem, and helps to some extent with the scaling down problem, although if the scaling down factor is more than about two, then some twinkling will still occur.

To anti-alias the texture, the blitter must read four source pixels, thus:



In this diagram, the point *(x.fx,y.fy)* is the target pixel address transformed back into source address space. The *x* and *y* parts are the integer part of the address, and the *fx* and *fy* parts are the fractional part of the address. Anti-aliasing is performed by taking a weighted average of the four surrounding pixels, designated A, B, C and D. The averaging function linearly interpolates between the corner values. The function is:

F(*x,fx,y.fy*) = (1-*fy*).((1-*fx*).A + *fx*.B) + *fy*.((1-*fx*).C + *fx*.D)

This function has to be performed on the intensity and C and R vectors of CRY pixels, and on each of R, G and B vectors for RGB pixels. The blitter only supports this function for CRY and RGB16 pixels.

The blitter anti-alias unit performs the two subtracts necessary to give the weighting factors, and then performs the six multiplies and three adds for each of the three pixel vectors. This adds an extra clock cycle to each texture generation transfer, and the unit produces one pixel at a time instead of phrase at a time as four source reads are required per pixel, instead of one.

Texture mapping one anti-aliased pixel when all four source pixels lie in the same DRAM bank takes:

row change + (4 x source read) + anti-alias + read to write delay + row change + destination write

which is 3 + (4 x 2) + 1 + 1 + 3 + 2 = 18 clock cycles per pixel

Texture mapping from internal RAM will be five clock cycles per pixel, with the texture duplicated in internal RAM it takes three clock cycles per pixel.

The INTERP bit has to be set in the extended command register to enable this function, and the TEXTRGB bit controls how the pixels are split up for the interpolation function. The destination address pointer should be set into 16 bit pixel mode, i.e. not in phrase mode.

Texture address unit 0 is the address of the bottom left pixel (A), and it is the fractional part of its address which is used to give the weighting factors. Texture address 1 is bottom right (B), 2 is top left (C), and 3 is top right (D). The data initialisation unit whose function is enabled by DATINIT knows about this functionality, and if both DATINIT and INTERP are set it will initialise address 0 to the step value, 1 to the step value with 1 added to the X address, and so on.

An interesting point: the interpolation unit could also be used to perform cross-fades between up to four pictures, by pointing the four texture X,Y pairs at each of the pictures, and using the fractional part of pointer 0 to control the relative levels. In this mode the pointers will have to be initialised manually and the DATINIT function cannot be used.

Texture Data

The texture map unit can fetch texture data from either internal or external memory. Texture addressing is more limited than general blitter addressing from the A1 and A2 address generators. Textures must be 2^n by 2^m pixels, for a restricted range of n and m. In addition, the texture must lie on a 2^n times 2^m boundary. Texture may be either 4 or 16 bits per pixel, and 4 bit-per-pixel textures are mapped to 16 bits by a colour look-up table within the blitter; so that all texture operations are performed at 16 bits per pixel.

Textures may be fetched from on-chip memory at the rate of one read per clock cycle, and this happens *at the same time* as external writes. If the texture data is duplicated in internal memory then two reads may occur per clock cycle, so that the blitter can generate texture data at the full write bus rate, i.e. one phrase per two clock cycles. If only one texture copy is available then this operation runs at half speed. There is no benefit in duplicating textures in external memory.

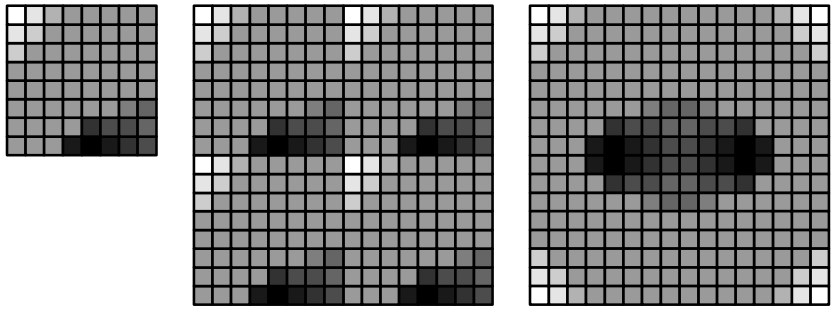
Internal texture RAM consist of two blocks of 1K x 32 RAM, into which pixels are packed. This RAM is shared with the GPU.

Textures can be from 32 to 2048 pixels in either dimension, either in 4 or 16 bits per pixel mode. Textures therefore range from 512 bytes, for a 32 x 32 4 bit texture to 8 Megabytes for a 2048 x 2048 16 bit texture. The latter may have limited applications! There is no restriction that textures have to be square.

Where double textures are being used in internal RAM, they have to be 4096 bytes apart. Double textures may only have a width and height of 32 or 64 pixels, they may be either 4 or 16 bits per pixel, but 64 x 64 16 bit per pixel textures may not be doubled because they are just too damn big.

Textures may be mirrored as they wrap. This function is enabled when the TEXTMIR bit is set in the extended command register. This means that textures will appear to tile even if they were not set up to do so, thus (for simplicity, a texture smaller than is possible in the blitter is shown here):

Texture Wrapping without TEXTMIR Wrapping with TEXTMIR



The blitter uses the texture pointer bit immediately above the most significant bit being used for address generation as the control bit for this function. For example, if the texture is 256 pixels wide, then if bit 8 of the texture X pointer is set the two's complement of the X pointer is used for address generation. This occurs for both X and Y pointers, will work for either internal or external textures, with 4 or 16 bit texture data, and will work for doubled internal textures too. The only restriction is that if either dimension of the texture is 2048 pixels, then the texture will not wrap in that dimension.

Note that the pixels at the mirrored edges are repeated.

Shading Textures

The data from the texture unit may be combined with the pattern data register to allow Gouraud shading to be combined with texture data. The extended intensity field is saturated, and used as a weighting factor to mix the texture data with the contents of the blitter color register. Full intensity corresponds to maximum weighting to the color register.

This allows the blitter a range of useful effects:

- darken or lighten textures by mixing with black or white
- adding fog and distance haze effects by mixing with grey or blue-grey
- color wash type effects, such as "red mist" by mixing with another colour

All these may be smoothly varied over a strip or polygon by setting up the Gouraud shade mechanism appropriately. The function uses multipliers to perform the mixing properly, it is

$$F(x) = (1-Iv).x + Iv.Cv$$

where Iv is the intensity value, x the input pixel value, an Cv the contents of the colour register. It can be used with either CRY or RGB pixels, by setting the TEXTRGB control bit appropriately.

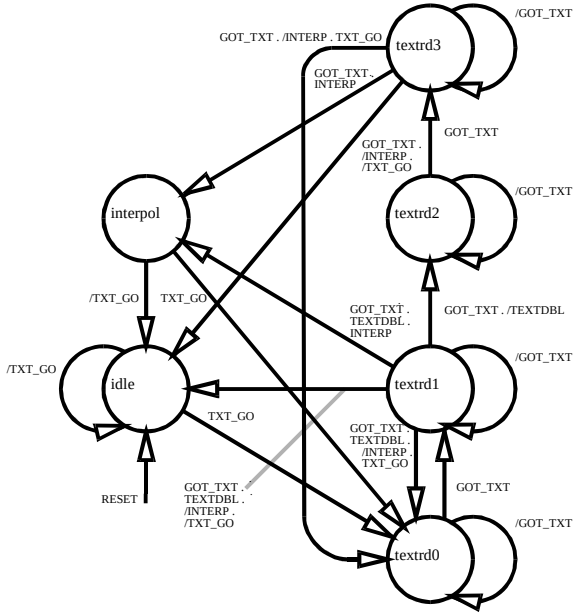
Two control bits, PDSEL0-1, control what is generated when PATDSEL is set. The output may be pattern data, as on Jaguar One, the texture data directly, the extended precision intensity pattern data with saturation applied, or the weighted mix data as described above.

The inputs to the multiplicative mixer may be selected from a range of sources. This discussed further above under the section on the Data Path, and allows a range of functions including:

- mixing texture data with a background color using the computed intensity to control the mix
- multiplicative Gouraud shading by mixing two colours, the texture data and the color register, according to the computed intensity. This allows shading in CRY or RGB16 modes, and offers a choice between shading to black or shading to white.
- translucent or anti-aliased edge texture mapping by mixing texture with the destination, with the source data providing a mix map.
- mixing two images from the source and destination fields, using the static intensity to control the mix.
- computed translucency, mixing the texture data with the destination according to the computed intensity.

Texture Read Operation

Texture operation is controlled by a simple state machine:



The states have the following functions:

idle	No action is performed
textrd0	Read from (X0,Y0), and from (X1,Y1) if TEXTDBL is true.
textrd1	Read from (X1,Y1) if TEXTDBL is false, or from (X2,Y2) and (X3,Y3) if TEXTDBL is true.
textrd2	Read from (X2,Y2).
textrd3	Read from (X3,Y3).
interpol	Interpolate between the four read pixel values, using the fractional parts of the (X0,Y0) pointer.

The texture read states will normally complete in one clock cycle when fetching from internal memory. The interpolation state will always complete in one clock cycle.

Built In Textures

Oberon has five textures built into ROM:

Name	First image	Second image
Metalica	F06000	F07000
Dirt	F06200	F07200
Clouds	F06400	F07400
Cement	F06600	F07600
Grass	F06800	F07800

The Cement, Clouds. Dirt and Metalica textures are 32 by 32 pixels. The Grass texture is 64 x 64. These are all four bit-per-pixel textures, so the appearance of them can be altered completely by modifying the blitter CLUT before texture mapping them. This will allow them to be re-used many times without looking the same each time.

The texture ROM is duplicated, so that TEXTMODE can be set to 3 when using them, for duplicated internal texture data. This can double the draw rate from them.

The names are not indicative of how the textures should be used, any of them could perform just about any of the functions described with a suitable palette of colours. You can get the source Targa files from Atari.

The CLUT values for these textures as originally encoded were:

```
metalica: 8869, 8776, 9764, 9788, 9797, 974E, A663, A77A,
          A678, 87A9, 977B, A78E, 87A1, 8789, 979A, C65A

dirt:     D567, D577, C553, B53B, E5C1, D68E, D58E, E5B1,
          E5A2, D6AF, D5DE, E6F3, E6DD, E6C8, E6E1, E673

clouds:   56BD, 56C6, 66D2, 66DE, 66D5, 66C6, 66CF, 46C6,
          56CE, 56C0, 46BE, 67DB

cement:   778C, 7787, 77A9, 7796, 7771, 777B, 775A, 7758,
          7765, 779C, 77A5, 764D, 77B8, 77AD, 77BD, 77CA

grass:    BA76, AA6D, BA8B, 9A66, A957, 8955, BAAE, 9947,
          B96F, BA9C, A978, BA9F, AA85, BA8D, 9A86, BAB2
```

These values are 16-bit CRY colors expressed in hex. (Clouds only uses 12 colors.)

Remember that you can use any CLUT you like with these textures, so you can darken or lighten them, change their hue, change their contrast, remove any color variation, use them in 16-bit RGB mode, or any other change you like. This means that you can use these textures without them looking the same as anyone else's use of them.

Puck

"First rehearse your song by rote,
To each word a warbling note.
Hand in hand with fairy grace
Will we sing and bless this place."

Act V, Scene 1

Puck is the companion chip to Oberon in the Jaguar games console, and is provides two J-RISC processors and some interface functions. These are:

- A J-RISC processor (DSP) principally intended for sound synthesis.
- A J-RISC processor (RCPU) intended to act as the main system processor.
- Frequency dividers for clock synthesis.
- Two programmable timers.
- Synchronous serial interface and baud rate generator (I^2S).
- Asynchronous serial interface and baud rate generator (ComLynx).
- Joystick interface decodes
- Six general purpose IO decodes

Puck occupies a 64K byte slot in 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.

Memory Controller

The memory controller in Puck allows the RISC processors to access 64 bit memory at the maximum bus rate. In order to do this with the best possible efficiency some of the Oberon functions are duplicated in Puck, to obtain the speed benefits of doing this control locally. The Puck memory controller registers **must** be programmed to match the MEMCON registers in Oberon at all times or failure will occur. The MEMCON1-2 and MEMCONP1-2 registers have the same respective bit positions, so the same data may be written to both.

PUCK_MEMC1 Puck Memory Configuration Register One F10040 WO

Bit 0	ROMHI	When set the two ROM decodes address the top 8M within the 16M window. When clear the ROM decodes address the bottom 8M. This document assumes throughout that ROMHI is set when discussing register addresses.
Bits 1,2	ROMWIDTH	Specifies the width of ROM: 0 8 bits 1 16 bits 2 32 bits 3 64 bits
Bits 3-15	unused	must be set to match MEMCON1.

All the above bits are undefined on reset and **must** be programmed to match MEMCON1.

PUCK_MEMC2 Puck Memory Configuration Register Two F10042 WO

Bits 0,1	unused	must be set to match MEMCON2.
Bits 2,3	DWIDTH0	Specifies the width of DRAM0 0 8 bits 1 16 bits 2 32 bits 3 64 bits
Bits 4,5	unused	must be set to match MEMCON2.

Bits 6,7	DWIDTH1	Specifies the width of DRAM1 0 8 bits 1 16 bits 2 32 bits 3 64 bits
Bits 8-11	unused	must be set to match MEMCON2.
Bit 12	BIGEND	Specifies that big-endian addressing should be used. This determines the address of a byte within a phrase and allows Jaguar to be used comfortably with Big-endian (Motorola) processors or with Little-endian (Intel) processors.
Bit 13-15	unused	must be set to match MEMCON2.

All the above bits are undefined on and **must** be programmed to match MEMCON2.

Frequency dividers

Puck is responsible for the synthesis of the system clocks. These are:

Chroma clock.	This is 4.43 MHz for PAL and 3.58 MHz for NTSC and should have a 50% duty cycle. The color encoder may require 4x this clock.
Video clock.	This is a multiple of the pixel clock (which is typically between 6 MHz and 12 MHz) and must be fixed relative to the chroma clock in order to avoid the "dot-crawl 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.
CPU clock	This is the clock for the 68000 CPU.

Clocks are generated by two free-running crystal oscillators, and by one voltage-controlled crystal oscillator (VCXO). The two free running crystals provide 26.6 Mhz, the same as Jaguar One; and a new frequency for Jaguar Two, probably in the range 30-35 Mhz. This is yet to be defined. The VCXO runs at four times the color sub-carrier, and gives the chroma clock.

!!! - more information required on the new clocking control.

CLK1 Processor clock divider F10010 WO

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 frequency 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

This register 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 on reset. The VCLKDIV output produces a pulse every N + 1 VCLK cycles.

CLK3 Chroma clock divider F10014 WO

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 the register. The CHRDIV output has a 50% duty cycle. This register is initialised to 3Fh (divide by 64) on reset.

The most significant bit of this register enables the chroma oscillator onto 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 multiple 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 onto the VCLK pin.

Programmable Timers

Puck 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 by M+1, where N and M are the values written to their associated registers. It is therefore possible to achieve frequency division in the range four to four billion.

The outputs of the second stages may be used to interrupt either of 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 a 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**
		F10036	**RO**
JPIT3	**Timer 2 Pre-scaler**	**F10004**	**WO**
		F1003A	**RO**

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, but this is really for chip test purposes as they can change while they are being read — they might be used by the DSP to measure short events with precision.

JPIT2	**Timer 1 Divider**	**F10002**	**WO**
		F10038	**RO**
JPIT4	**Timer 2 Divider**	**F10006**	**WO**
		F1003C	**RO**

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 of the DSP or the CPU. These interrupts are independently maskable.

Interrupts

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

- External — A rising edge on the EINT[0] input to Puck may cause an interrupt.
- DSP — The DSP 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.
- RCPU — The RCPU may generate an interrupt by writing to a port.

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 interrupt sources.

PUCK_INTC Interrupt Control Register F10020 WO

Bit	Name	Description
0	EXT_ENA	Enable external interrupt 0.
1	DSP_ENA	Enable interrupts from the DSP
2	TIM0_ENA	Enable Timer One interrupts (sample rate)
3	TIM1_ENA	Enable Timer Two interrupts (tempo)
4	ASI_ENA	Enable Asynchronous Serial Interface interrupts
5	SSI_ENA	Enable Synchronous Serial Interface interrupts
6	RCPU_ENA	Enable interrupts from the RCPU
8	EXT_CLR	Clear external interrupt 0.
9	DSP_CLR	Clear the DSP interrupt
10	TIM0_CLR	Clear Timer One interrupt
11	TIM1_CLR	Clear Timer Two interrupt
12	ASI_CLR	Clear the Asynchronous Serial Interface interrupts
13	SSI_CLR	Clear the Synchronous Serial Interface interrupts
14	RCPU_CLR	Clear the RCPU interrupt

PUCK_INTS Interrupt Status Register F10020 RO

This register allows a processor to determine which interrupt is pending.

Bit	Name	Description
0	EXT_INT	External interrupt 0
1	DSP_INT	DSP interrupt
2	TIM0_INT	Timer One interrupt
3	TIM1_INT	Timer Two interrupt
4	ASI_INT	Asynchronous Serial Interface interrupt
5	SSI_INT	Synchronous Serial Interface interrupt
6	RCPU_INT	RCPU interrupt

Synchronous Serial Interface

The synchronous serial interface in Puck is the interface to the audio digital to analogue converters, and is also the interface to the serial data stream from the CD-ROM drive. These two functions may be tied together, for instance when playing red book audio; or they may be quite separate. The data from the CD is also available from a memory mapped FIFO. This is described elsewhere (it is part of the Butch device in the Jaguar One CD-ROM).

The interface has two major components; a synchronous receiver/transmitter within the DSP memory area; and a CD DMA controller which is a stand-alone bus master.

The synchronous receiver/transmitter was also present in Jaguar One. It is the synchronous equivalent of a UART, and its main function is to transmit audio data to the DAC (digital-to-analogue converter). It can also receive the data from the CD. The CD DMA controller is new for Midsummer. It is described below on page 8.

A synchronous serial interface, as implemented here, consists of four wires: receive data, transmit data, serial clock and word strobe. The serial clock and word strobe are generated by the bus master of the interface, and define the bit rate and data framing on the two data lines. Puck can be either a master or a slave to the serial interface connected to the DAC and DSP connector, but can only be a slave to the CD-ROM (expansion bus). It is not possible to be a bus slave on the expansion bus (unlike Jaguar One).

The control of these lines is described in greater detail below. Puck has effectively two synchronous serial interfaces, so that data may be transferred from the CD and to the DAC simultaneously, without requiring both to use the same bit rate and word alignment.

The pins provided are as follows:

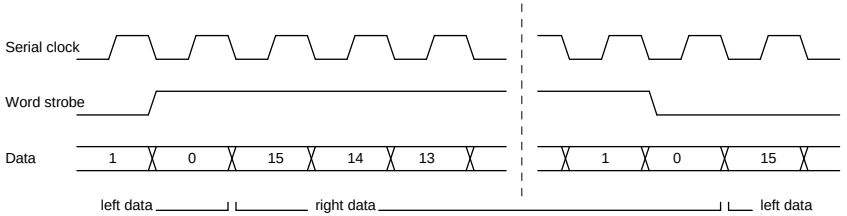
Name	Function	Description
TXD	output	transmit data to DAC and DSP connector
RXD	input	receive data from CD-ROM (expansion bus) or DSP connector
SCK	input/output	serial clock output to DAC; input from or output to DSP connector
WS	input/output	input from or output to DSP connector
QWS	output	word strobe output to DAC (muted version of WS)
SCK2	input	serial clock input from CD-ROM (expansion bus)
WS2	input	word strobe input from CD-ROM (expansion bus)

Diagrams below show how these are configured in operation.

Synchronous Serial Receiver / Transmitter

The interface can work in two modes. The first, called mode16, is compatible with I^2S and has a sixteen bit word length. The start of left and right words are marked by transitions in word strobe. Interrupts are generated on the rising edge of word strobe. The second mode, called mode32, allows longer packets of data to be communicated. In this mode a rising edge on word strobe synchronises the system which continues to receive/transmit 32 bit words. Interrupts are generated every 32 bits. Mode 32 is not used within the Jaguar console.

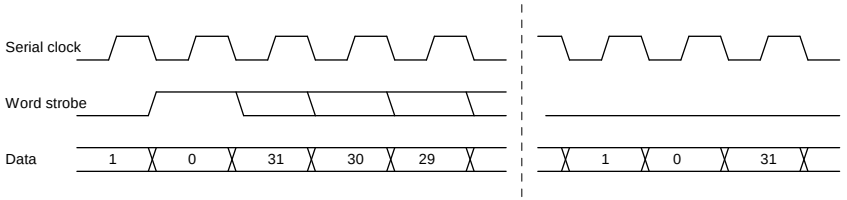
Mode16



Note

- The word strobe precedes the data by one bit.
- The word strobe and transmit data are clocked by the negative edge of the clock to provide the maximum set-up and hold time in the receiver/slave.
- Data and word strobe inputs are sampled on the rising edge of the clock.
- The data is sent transmitted MSB first. If the interval between word strobe transitions is greater than 16 bits the transmitter sends zeroes after the LSB and the receiver ignores them. If the interval is less than 16 bits the receiver sets the missing bits to zero.
- The diagram is the same whether the timing is generated internally or externally but Puck only produces word strobes 16 bits in length.

Mode32



Note

- Only the rising edge of the word strobe is significant
- Outputs change on the falling edge of the clock, and inputs are latched on the rising edge.
- 32 bit words continue to be received / transmitted until the next rising edge of word strobe.

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 these addresses. All transfers to them should be 32 bit, but the registers

themselves are only 16 bit. The addresses given are therefore a big-endian view of their position in the memory map.

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 0	INTERNAL	When set this bit enables the serial clock and word strobe outputs.
Bit 1	MODE	When set this bit selects MODE32.
Bit 2	WSEN	This bit enables the generation of word strobe pulses. When set PUCK produces a word strobe output which is alternately high for 16 clock cycles and low for 16 clock cycles. When cleared Puck will not generate further high pulses. This can be used by software to generate one word strobe at the start of a packet of long-words in MODE32.
Bit 3	RISING	Enables interrupts on the rising edge of word strobe.
Bit 4	FALLING	Enables interrupts on the falling edge of word strobe.
Bit 5	EVERYWORD	Enables interrupts on the MSB of every word transmitted or received.

LTXD Left transmit data F1A148 WO
RTXD Right transmit data F1A14C WO

These two sixteen bit registers hold data to be transmitted.

In MODE16 the right data is transferred to a shift register following the rising edge of word strobe and the left data is transferred following the falling edge of word strobe.

In MODE32 the left data (most significant) is transferred first after the rising edge of word strobe (and every 32 clocks later), the right data is transferred 16 clocks after the left data.

In either mode the registers may only be updated when the previous contents have been transferred to the shift register.

LRXD Left receive data F1A148 RO
RRXD Right receive data F1A14C RO

These two sixteen bit registers hold received data.

In M0DE16 the right data is transferred from the shift register to the register following the falling edge of word strobe and the left data is transferred following the rising edge.

In M0DE32 the left data (most significant) is transferred from the receive shift register to the left register 16 clocks after the rising edge of word strobe (and every 32 clocks later). The right data is transferred 16 clocks after the left data.

SSTAT Serial Status F1A150 RO

Bit 0	WS	This bit reflects the state of the Word Strobe pin in order for software to determine which data is being received. Do not use this signal for reading input data. Read the interrupt control register instead.
Bit 1	Left	In MODE32 it is not necessary for the Word Strobe to be toggled every 16 bits. An internal counter keeps track and this bit may be used as an alternative to WS to determine which word is currently being transmitted or received.

CD DMA Controller

The CD DMA controller is new for Midsummer. It is intended to allow the CD to be used as a data storage medium with a minimal system overhead for data transfer.

Overview

The original Jaguar 1 I^2S interface provided simple 32-bit I/O registers, requiring considerable processor overhead both to receive data from the CD and to transmit audio to the DAC. Midsummer improves this reception of data from the CD-ROM.

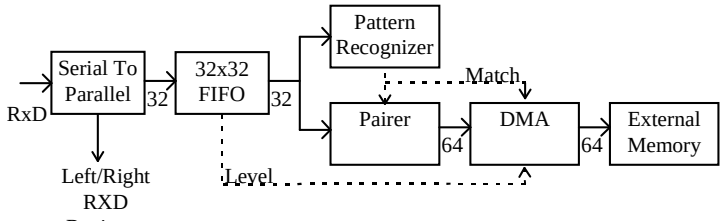
Two extra pins (SCK2 and WS2) allow the receive and transmit functions to operate completely independently, and the CD DMA controller can receive double-speed CD data, recognise partition markers then fill a circular buffer, all without processor overhead, using less than 0.4% of the main bus bandwidth.

The controller functions in a similar way to the pattern-matching and data-transfer parts of the CD-BIOS cd_initm/cd_read calls, and indeed these calls can be enhanced to use this mechanism. Management of the CD mechanism, including error-detection, remains a software function.

How it works

A 32-deep FIFO accumulates longs as they arrive from the I^2S receiver. A pattern-recogniser searches these longs for a string of matching values (CD_PAT determines the value and PAT_LEN the number of longs). Once the pattern is seen, the recogniser stops taking longs from the FIFO, which starts to fill up. Once the FIFO has filled to a programmable 'high-tide' mark, the DMA controller acquires the bus and burst-writes phrases (supplied by the Pairer) to memory until the FIFO is empty. The FIFO then starts to fill-up again and the process repeats.

The first and last phrases written are masked internally if necessary, to allow long-alignment of the start and end points.



CD_CTRL CD DMA Control Register F10080 WO

Cleared on reset.

Bit	Name	Description
0-4	HIGH_TIDE	Write a value between 3 and 30. When running, the CD DMA will empty the FIFO whenever it contains more than this many longs (i.e. LEVEL>HIGH_TIDE). Too-low a value will cause the DMA to acquire the bus often and inefficiently transfer only a few values. Too-high a value will cause the FIFO to overflow if the DMA cannot acquire the bus quickly (because a higher-priority bus-master is active).
5-10	PAT_LEN	Write a value between 0 and 63. The number of longs required in the partition-marker pattern. DMA will commence after this number of longs have been recognised. Set to zero to disable pattern-recognition.
11	CD_BIGPHR	Controls the 'endianness' with which DMA stores incoming longs in phrases. If true, the first long is stored in phrase bits [63:31]. Set true for Jaguar.

12	CD_BIGLNG	Controls the 'endianness' with which incoming left/right words from the I^2S interface are packed into long. If true, the first word is stored in long bits[31:16]. Set false for Jaguar.
13	HI_PRI	Causes CD DMA to request the bus with high priority
14	CD_LXFER	Determines which word in each received pair of words is considered the 'second', and thus the one that causes the pair to be transferred into the CD DMA controller. Clearing this bit means 'transfer on right', i.e. incoming words are in 'left,right' pairs (usually the case). Setting this bit means 'transfer on left', i.e. incoming words are in 'right,left' pairs.
15	unused	Write zero.

CD_STAT CD DMA Status F10080 RO

Bit	Name	Description
0-4	LEVEL	Number of longs in the FIFO. Zeroed by 'CD_RUN' command.
5	MATCH	If true, pattern-matching is complete and DMA is active. Cleared by 'CD_RUN' command unless PAT_LEN is zero.
6	OVERFLOW	Indicates that FIFO has overflowed since last 'CD_RUN' command. A higher-priority bus-master is hogging the bus for too long. HIGH_TIDE may need to be reduced.
7	FINISHED	DMA has reached the end address and stopped. Cleared by 'CD_RUN' command.
8-15	unused	Zero.

CD_FLOW F10084 WO

See the 'Data Flow' section below.

Cleared on reset.

Bit	Name	Description
0	CD_ENHANCED	When clear, bits 1,2,3 in this register are controlled by the INTERNAL bit in the SMODE register, ensuring compatibility with Jaguar1. Set to control these bits directly.
1	CD_I2SCK2EN	Determines source of internal SCK and WS. 0 for the internal clock generator, 1 for input from the CD on pins SCK2,WS2. If CD_ENHANCED is clear, this bit is !INTERNAL. Regardless of the state of this bit, the WS interrupts controlled by the SMODE register, and the state of WS read in the SSTAT register, refer to the transmit WS only.
2	CD_I2SRX2EN	Determines the source of LRXD,RRXD receive register timing. 0 for the internal SCK and WS 1 for input from the CD on pins SCK2,WS2. If CD_ENHANCED is clear, this bit is zero.
3	CD_I2SBYPASS	Determines the source of serial data out to the DAC on pin TXD. 0 for the LTXD,RTXD registers 1 for the CD on pin rxd. If CD_ENHANCED is clear, this bit is zero.
4	CD_I2SOE	Output-enable for SCK and WS pins. Normally set to 1 (by boot ROM) to drive the DAC. Must be clear to slave Puck to these pins.
5	CD_I2SWSDEL	If clear,WS is delayed by one SCK (DAC data is Philips format) If set, WS is not delayed by one SCK (DAC data is Sony format)
6	CD_I2SWSINV	If clear, WS is output high for left word, low for right (Philips format) If set, WS is output low for left word, high for right (Sony format)
7-15	unused	Write zero.

CD_ ACTN F10088 WO

Bit	Name	Description
0	CD_RUN	A write to this register with this bit set clears the DMA controller then enables it.
1	CD_STOP	A write to this register with this bit set disables the DMA controller
2-15	unused	Write zero.

CD_PATH F1008C RW

High word (bits 31..16) of pattern-recogniser long.

CD_PATL F1008E RW

Low word (bits 15..0) of pattern-recogniser long.

CD_STARTH F10090 RW

Bit	Description
0-7	High word (bits 23..16) of CD DMA controller start address. Once pattern is recognised, longs will be written to memory from this address.
8-15	Read/Write zero.

CD_STARTL F10092 RW

Bit	Description
0-1	Read/Write zero.
2-15	Low word (bits 15..2) of CD DMA controller start address (long-aligned).

CD_ENDH F10094 RW

Bit	Description
0-7	High word (bits 23..16) of CD DMA controller end address. DMA will continue to this address, then stop and the 'FINISH' bit will be asserted. To continue forever, set this address outside the range of the address mask.
8-15	Read/Write zero.

CD_ENDL F10096 RW

Bit	Description
0-1	Read/Write zero.
2-15	Low word (bits 15..2) of CD DMA controller end address (long-aligned).

CD_MASK F10098 RW

Bit	Description
0-4	Controls a mask applied to the DMA address, confining it within a circular buffer. This bit of the DMA current address is always held clear. e.g. If CD_MASK=8, bit 8 of the DMA current address will be held clear, so instead of incrementing from XXX0F8 to XXX100, address will wrap-around to XXX000. The circular-buffer size is 2^{CD_MASK} bytes, and aligned on a $2^{(CD\_MASK+1)}$ byte boundary. DMA transfers are phrase-aligned, so set this register to between 3 and 23. Set to 0 to disable.
5-15	Read/Write zero.

CD_CURH F1009C RO

Bit	Description
0-7	High word (bits 23..16) of CD DMA controller current transfer address. Note that this may change between reading this register and CD_CURL.
8-15	Zero.

CD_CURL F1009E RO

Bit	Description
0-2	Zero.
3-15	Low word (bits 15..3) of CD DMA controller current transfer address (phrase-aligned).

CD_FAKEH	F1009C	WO

Test only.

High word of 'fake' input long. See CD_FAKEL.

CD_FAKEL	F1009E	WO

Test only.

Low word of 'fake' input long. Writing to CD_FAKEH then CD_FAKEL inserts a long into the CD DMA controller as if it had arrived from the I2S data stream.

Using the CD DMA controller

To use the CD DMA controller, the following registers must be set-up:

1)Write a '1' to the CD_STOP bit to halt the controller (only necessary if the 'FINISHED' bit is false)

2)Set CD_CTRL as desired (PAT=0 to disable pattern-recognition).

3)Set CD_START to a long-aligned address.

For a linear buffer:

4)Write 0 to CD_MASK to disable it

5)Set CD_END to a long-aligned address.

For a circular buffer:

4)Write suitable value to CD_MASK (see example below).

5)For endless loop, set CD_END outside the buffer, else to a long-aligned address within.

6)If using the pattern recogniser, write desired long to CD_PAT.

7)Write to CD_ATN with CD_RUN bit set.

8)Start the CD mechanism.

As data arrives from the CD-ROM, it is searched for the partition marker - a pattern of PAT_LEN longs of value CD_PAT. Once this is found, subsequent longs are written to memory from CD_START. At any time, the current address that the DMA is writing-to can be monitored by reading CD_CUR. Once the end address is reached, the controller shuts-off and the 'FINISHED' bit becomes true.

Example

To read from a partition labelled with a pattern of sixteen '00000001's, into a 128K circular-buffer at the top of DRAM :

1)Write a '1' to the CD_STOP bit.

2)We'll set HIGH_TIDE to half-way through the FIFO (16), PAT_LEN to 16, CD_BIGEND true and low priority, so write $0A10 to CD_CTRL.

3)Write $001E0000 to CD_START.

4)We need to set the mask for a 128K circular buffer. This is 2^{17} so write 17 to CD_MASK.

5)Write 00FFFFFC to CD_END (outside mask, so will loop forever).

6)Write 00000001 to CD_PAT.

7)Write 0001 to CD_ACTN to enable the controller.

8)Start the CD mechanism.

Data Flow

Jaguar1 had a single I^2S bus of which it could either be master (driving SCK and WS) or slave (receiving SCK and WS). Both receive and transmit had to be clocked by the same source, so for example when slaving to a double-speed CD ROM, the DAC had to be driven at 88200Hz - somewhat excessive!

Midsummer separates the receive and transmit functions onto two separate I^2S busses, allowing data to be transmitted to the DAC at a different rate than it is received from CD and also allowing CD Audio disks to be played directly to the DAC without processor involvement.

If the CD_ENHANCED bit in CD_FLOW is left clear after reset, the data path is controlled automatically by the INTERNAL bit in the SMODE register, allowing Jaguar 1 compatibility. The only functionality lost compared to Jaguar1 is that peripherals on the Expansion bus can no longer be slaves. $0010 should be written to CD_FLOW by the Boot ROM to enable SCK,WS as outputs.

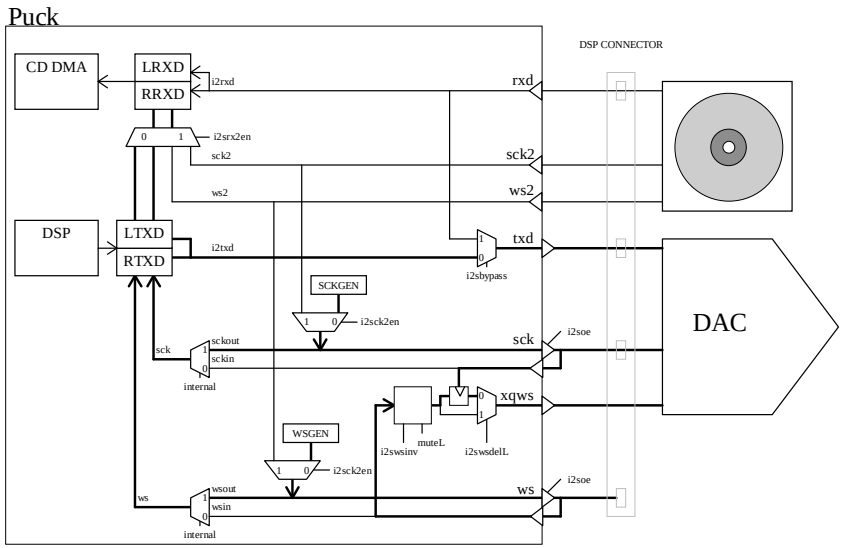
Frequently-used configurations are shown overleaf, with the corresponding CD_FLOW value (in these examples, the second nibble of CD_FLOW is assumed to be zero, although it could take other values).

"Audio Out Compatible" Mode

No CD-ROM connected (or it is disabled). DAC driven by DSP. Puck is bus-master.

SMODE $0005 (INTERNAL=1). Set SCLK for desired sample-rate.

CD_FLOW $0000 (COMPATIBLE)

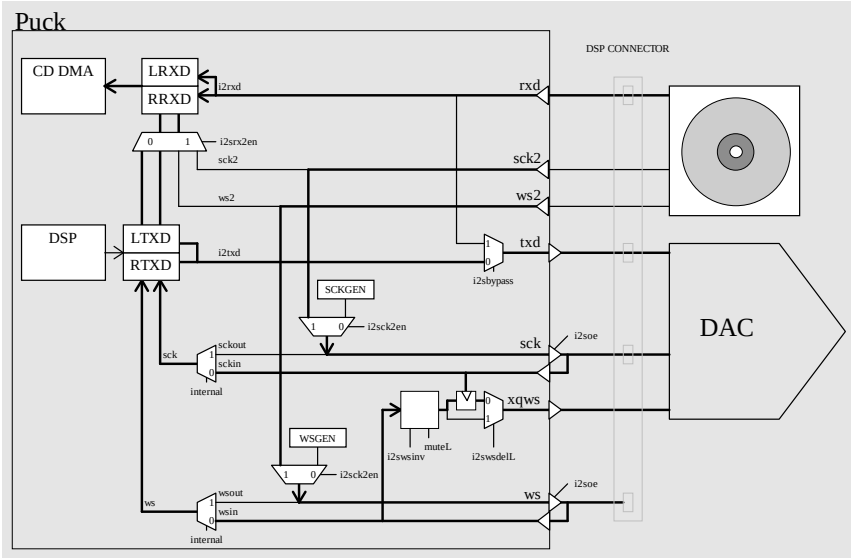


"CDROM In Audio Out Slaved Compatible" Mode

CDROM read via registers or CD DMA. DAC driven by DSP. CDROM is bus-master.

SMODE $0000 (INTERNAL=0)

CD_FLOW $0000 (COMPATIBLE)



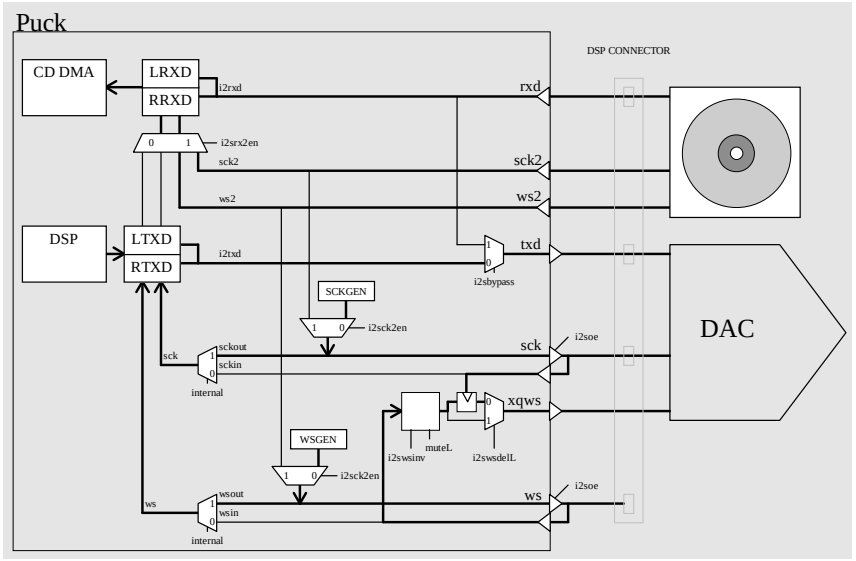
"CDROM In Audio Out Independent" Mode

CDROM read via registers or CD DMA (CD is master). DAC driven by DSP (Puck is master). Input and output are completely separate.

SMODE $0005 (INTERNAL=1). Set SCLK for desired sample-rate.

CD_FLOW $0005

3	2	1	0
CD_I2SBYPASS	CD_I2SRX2EN	CD_I2SCK2EN	CD_ENHANCED
0	1	0	1



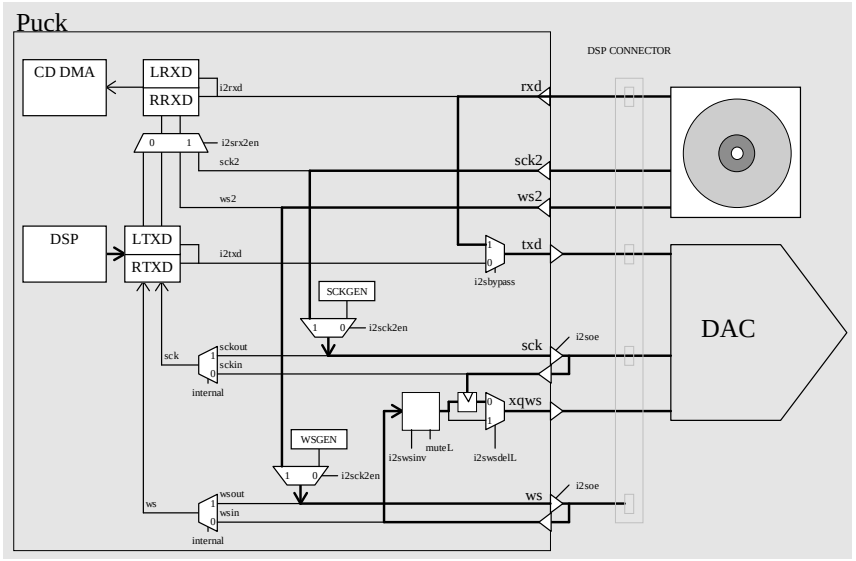
"CD Audio Bypass" Mode

Audio from CD plays straight-through to DAC without CPU intervention. CD is bus-master of both buses. Audio data can be read simultaneously via registers or CD DMA if desired.

SMODE $0005 (INTERNAL=1).

CD_FLOW $000F

3	2	1	0
CD_I2SBYPASS	CD_I2SRX2EN	CD_I2SCK2EN	CD_ENHANCED
1	1	1	1



Network UART

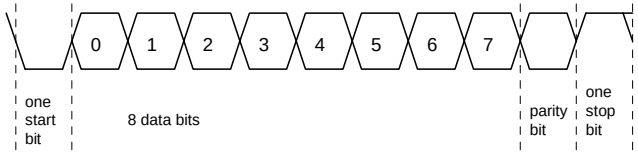
Puck contains a simple asynchronous serial UART intended as a serial network interface (like ComLynx) or as a serial communications port (RS232 or MIDI). The serial interface consists of two wires, UARTI, the receive data input and UARTO the transmit data output.

A prescaler register is used to allow a wide range of programmable baud rates. The highest baud rate possible is the system clock divided by thirty-two.

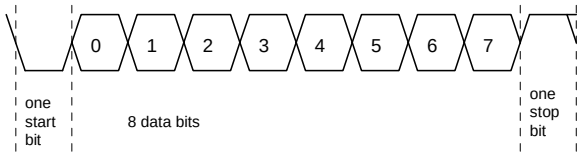
The data transmitter is double buffered, allowing a character to be written into the data register before the transmission of a previously written character is complete. The data receiver is also double buffered, a second character can be received on the UARTI pin before the previous character has been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

Data is transmitted and received in the formats shown below:

Normal transmit/receive format



Receive format when NOPAR is set:



The parity can be ODD, EVEN or none. The polarity of both the output and the input can be programmed to be active high or low. The polarity shown is active low. The transmitter can be programmed to transmit a stop bit in the parity position, and the receiver can be programmed into not expecting a stop bit at all, supporting the standard 8-bit, no parity, one stop bit format.

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

The UART is accessible either as part of the normal 16-bit IO interface, or within the RCPU internal space. If RCPU control of the UART is enabled, then the transmit and receive data buffers can be long words if required, reducing the overhead required for sending bursts of data.

ASICLK Asynchronous Serial Interface Clock F10034 R/W

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 ODD bit is transmitted in the parity bit time.
2	TXOPOL	Transmitter output polarity. Setting this bit to a one causes the UARTO output to be active low.
3	RXIPOL	Receiver input polarity. Writing a one to this bit makes the UARTI into an inverting input.
4	TINTEN	Enables transmitter interrupts. Note that the asynchronous serial interface bit in the Interrupt Control Register also needs to be set to enable interrupts.
5	RINTEN	Enables receiver interrupts. As for TINTEN the asynchronous serial interface bit in the Interrupt Control Register must also be set.
6	CLRERR	Clear Error. Writing a one to this bit clears any parity, framing or overrun error condition.
14	TXBRK	Transmit break. Setting this bit 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 ASICTRL 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	SERIN	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 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 register is written bits [0..7] are transmitted from the UARTO pin. Bits [8..15] are not used and should be written as zero.

RCPU Extended UART Control F1813C Read/write

This register supplements the ASICTRL register at F10032, and both registers must be initialised before the UART is used.

Bit	Name	Description

0	ERROR	When read, this bit indicates that one of the error bits below is set. Writing a one to this bit clears all the error flags. Writing a zero has no effect.
1	BYTE_INT	When this bit is set, the RCPU is interrupted after each byte is received. When this bit is clear, it is interrupted when four bytes have been received.
2	RX_INT	When this bit is set, receiver interrupts are enabled. An interrupt is generated at the rate determined by BYTE_INT. The status of this bit is reflected by a read.
3	TX_INT	When this bit is set, transmitter interrupts are enabled. An interrupt is generated whenever the transmit buffer is empty. The TX_BYTE bit below controls whether this is after one or four bytes. The status of this bit is reflected by a read.
4	NOPAR	When this bit is set, the receiver no longer expects to receive a parity bit. This allows the standard 8-bit, no parity, one stop bit format to be received. It has no effect ion the transmitter, so to transmit this format you should ensure the transmitted parity bit corresponds to a stop bit. This bit also applies to the IO interface. The status of this bit is reflected by a read.
5	TX_BYTE	Set this bit to transmit single bytes. If this is set only the first byte is transmitted. The status of this bit is reflected by a read.
6	RCPU_TRANSMIT	Set this bit if the RCPU is to control the UART transmit interface. If this bit is clear, the normal IO interface controls transmit. The status of this bit is reflected by a read.
7	RCPU_RECEIVE	Set this bit if the RCPU is to control the UART receive interface. If this bit is clear, the normal IO interface controls receive. The status of this bit is reflected by a read.
16	OVERRUN_ERROR	This error flag indicates that the four byte receive buffer has overflowed and receiver data has been lost. This bit is read only.
17	FRAMING_ERROR	This error flag indicates that a framing error occurred on received data. The UART will cease operation until the error is cleared. This bit is read only.
18	PARITY_ERROR	This error flag indicates that received data has a parity error. The UART will cease operation until the error is cleared. This bit is read only.
19-21	BYTES_IN_BUF	This value indicates how many bytes are present in the UART receive data buffer. Valid values are 0-4. Even if the receiver is in byte mode (BYTE_INT set), further values will be added to the buffer until the long overflows. This value is read only.
22-24	BYTES_LAST_READ	This value indicates how many bytes were present the last time the receive data buffer was read. As it is not possible to read the receive data buffer and the BYTES_IN_BUF value atomically, the counter is latched whenever a read occurs and the value stored here.
25	RX_INT_FLAG	The current interrupt was caused by the receiver. This bit is read-only.
26	TX_INT_FLAG	The current interrupt was caused by the transmitter. This bit is read-only.

RCPU UART Data F18140 Read/write

This long location contains a long write-only transmit data buffer, and a long read-only receive data buffer. For a full discussion of the UART, refer to the section on it below. These buffers are big-endian, this means that the byte order of transmission or reception is as follows.

Bits	Order
24-31	first byte
16-23	second byte
8-15	third byte
0-7	fourth byte

If the interface is being operated in byte mode, then the byte should be read from or written to bits 0-7. However, note that if read overflow occurs (which is not flagged as an error in any case until the buffer contains four bytes), then the bytes will be shifted up in the long buffer as they are received. This means that a byte mode RCPU UART receiver actually has nearly four byte times to respond to the interrupt, a truly massive latency were it to ever occur!

Joystick Interface

Puck has four outputs which together control four external TTL ICs to provide the joystick interface. There are two registers

JOY1 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 J0YL2 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 J0YL3 is the inverse of the value in bit 15.

J0Y2 Button register F14002 RW

When read the button input buffer is enabled and the data reflects the state of the four button inputs. Output J0YL1 is asserted (active low) during the read.

There are two joystick connectors each of which is a 15 pin high density 'D' socket. The pinouts are as follows:

PIN	J5	J6
1	JOY3	JOY4
2	JOY2	JOY5
3	JOY1	JOY6
4	JOY0	JOY7
5	NC	NC
6	BO/LP	B2
7	5 VDC	5 VDC
8	NC	NC
9	GND	GND
10	B1	B3
11	J0Y11	J0Y15
12	JOY10	JOY14
13	JOY9	JOY13
14	JOY8	JOY12
15	NC	NC

The JOYx signals correspond to bit x on the joystick port. All the joystick signals can be used as inputs. Signals JOY0 to J0Y7 can also be used as outputs. The direction of these signals is determined by bit15 of the joystick output port. If bit 15 is set JOY0 to JOY7 are outputs. All joystick signals are pulled up with resistors. Signals B0 to B3 are bits 0 to 3 on the button port. The LP signal is a light-gun input, a high level on this input transfers the current horizontal and vertical counts to the light-pen registers.

General Purpose IO Decodes

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

GPIO0	F14800-F14FFFh	CD-interface
GPIO1	F15000-F15FFFh	DMA ACK
GPIO2	F16000-F16FFFh	Cartridge
GPIO3	F17000-F177FFh	
GPIO4	F17800-F17BFFh	
GPIO5	F17C00-F17FFFh	Paddle Interface

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

Appendices

The COBWEB Development Board

All development systems currently being shipped (on August 1995), are Cobweb boards. You should read these notes before using it.

The Cobweb board is a prototype development board for Midsummer which has the Oberon β-test ASIC from Midsummer and the Jerry ASIC from Jaguar One. This system is intended to allow some software development to start before the availability of Puck. The Oberon β-test ASIC is not the final production version of Oberon, and is both slower and buggier then the production silicon.

Developers using this board should be aware of the following limitations and other issues:

1. The Oberon β-test ASIC only runs at 26.6 MHz, and even at this speed requires forced cooling.
2. Because Jerry is fitted, the Jaguar One DSP load and store limitations still apply (and all other Jerry/DSP bugs).
3. This is a development board - there are likely to be un-discovered bugs in it.
4. The development environment currently available is that of Jaguar One (but it does all seem to work!).
5. The video and audio quality may be poor

Because Puck is not present, the following features described in this document are not present:

1. The RCPU
2. The DSP enhancements over Jaguar One, including the PCM engine
3. The CD DMA channel

The GPU and Blitter enhancements, including texture mapping, are all present. Please refer to the bugs list below, or a more up-to-date version from Atari, for problems present in this Oberon β-test ASIC. These should not be present in the production version.

Data Organisation - Big and Little Endian

The Jaguar system is intended to be used in either a little-endian, e.g. Intel 80x86, or big-endian, e.g. 680x0, environment. The difference between these two systems is to do with the way in which bytes of a larger operand are stored in memory. There is potential for considerable confusion here, so this section attempts to explain the differences.

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

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

IO Bus Interface

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

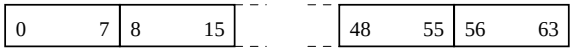
Co-Processor Bus Interface

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

Pixel Organisation

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

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



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

Consider an eight bit per pixel mode:

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

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

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

Oberon and Puck Bugs List

"If we shadows have offended,
Think but this, and all is mended,
That you have but slumber'd here
While these visions did appear.
And this weak and idle theme,
No more yielding but a dream,
Gentles, do not reprehend:
if you pardon, we will mend:"

Act V. Scene 1.

This document lists the known bugs in the Oberon and Puck devices. This is revision code 3 silicon.

Level	
3	This bug completely prevents some part of the ASIC from operating. Some functionality cannot be demonstrated, and further bugs could be obscured.
2	This bug can be fixed to some extent by a software or hardware work-around. The functionality may still be impaired but is demonstrable.
1	This bug can be fixed by a simple software or hardware work-around with no significant loss of functionality or performance.

The reference to hardware or software in bugs indicates who it affects.

Oberon Bugs

1 Carried Over 68K Bus Interface Bugs

These bugs we inadvertently carried over from Tom:

Level	1 *hardware*
Description	When the 68000 is slow to retract BGL and Oberon performs a very short bus cycle, it can see the trailing edge of BGL at the start of the next BRL operation and erroneously assume that it has the bus.
Work-round	Filter BGL through a flip-flop set on the falling edge of BGL, and cleared synchronously

by BGACKL.

Level 1 *hardware*

Description Oberon can retract BRL too quickly.

Work-round Stretch the trailing edge of BRL by one clock cycle.

2 GPU: DMA from the register file does not give the right data

Level 1 *software*

Description If the DMA engine is set to transfer the register file into external memory, it does not read the data correctly (this only works anyway if GO is clear).

Work-round Read the data by any other practical means.

3 GPU: DMA into the GPU RAM fails if the bus is lost

Level 2 *software*

Description When the DMA engine is transferring data into GPU RAM, and the bus is lost during the transfer, then values van be repeated within GPU RAM. This means that the data is no use.

Work-round Ensure that the DMA engine does not lose the bus during the transfer. This can be done by disabling refresh across the transfer, and ensuring that no higher priority bus master can use the bus - refer to the bus arbitration description on page Bus Arbitration5.

4 Blitter: Interpolated Pixel Math Errors

Level 2 *software*

Description When the blitter is anti-aliasing texture data, it appears that the interpolation math can cause the values to be one less than they should be under some circumstances. This results in a visible problem when CRY color values are reduced by one, even when mixing four pixels with the same color value.

Work-round Only certain textures show the problem, so either choose your textures carefully, or do not use the anti-aliasing.