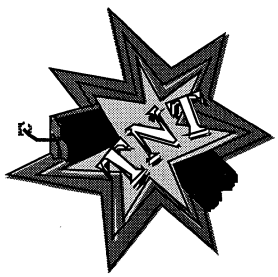
Rev. Level	ECN #	Approved	Date	Revision Description
A	34359	Steve Polzin	05-Jul-95	Initial Release

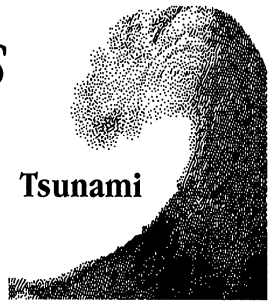
PowerMacintosh 8500/7500 ERS

Revision A

05-Jul-95



AV TECHNOLOGIES
Nitro
Tsunami



Steve Polzin: 862-6497

eWorld: Polzin

Rev. Level	ECN #	Approved	Date	Revision Description
A	34359	Steve Polzin	05-Jul-95	Initial Release

This document represents The PowerMacintosh 8500/7500 System Engineering Reference Specification.

TABLE 1 Revision History

Who	When	Rev	What
S. Polzin	25-Feb-93	0.0	Limited Release
S.Polzin	26-Feb-93	0.1	Bill's Updates on Video Modes
S.Polzin	04-Mar-93	0.2	Feedback from Brian, Bruce and Steve E.
S.Polzin	12-Mar-93	0.3	Updated Memory Map (stole some of Ron's MacRISC addressing stuff), Added Init Chapter, Feedback from Video Software review, feedback from Todd, added clock and some physical stuff, added Bill's video capabilities matrix. Release to "VG5" Reviewers.
S.Polzin et.al.	25-May-93	0.98	Final pre-world release. New PCI chapter.
S.Polzin et.al.	28-May-93	0.99	Kill the trees release. Logically complete. Some physical stuff may change at which point V1.0 will be released.
S.Polzin et.al.	16-Feb-94	1.0	Update to Unify TNT and Tsunami, stealing lots of stuff from Dale. Include real chip names, better physical stuff, accurate board level registers etc.
S.Polzin et.al.	21-Feb-94	1.01	64-bit PCI gone
S.Polzin et.al.	05-Jul-95	A	What we actually are shipping in preparation for Documentation release. No More VCI etc.. Eliminate direct Tsunami references

All versions following V1.01 will uses change bars to highlight changes.

Table Of Contents

Rev. Level	ECN #	Approved	Date	Revision Description
A	34359	Steve Polzin	05-Jul-95	Initial Release

CHAPTER 1

PowerMacintosh 8500/7500 System Overview

1.1 Introduction

The PowerMacintosh 8500/7500 family of PowerPC Macintosh computers is the second generation PowerPC desktop Macintosh family following PowerMacintosh 8100/7100/6100. PowerMacintosh 8500/7500 is the first PowerPC Macintosh that uses PCI as the new expansion bus and also conforms to the MacRISC internal HW/SW architecture. MacRISC is an Apple internal architectural definition that is the contract between hardware development and system software development. MacRISC defines the set of software visible state and behavior in the hardware.

The PowerMacintosh 8500/7500 family of products includes three initial models that share the same core chip-set:

- PowerMacintosh 7500 is a 3-spot Desktop machine in a new product design called "Outrigger". PowerMacintosh 7500 integrates advanced AVTECHNOLOGIES features in a compact form factor.
- PowerMacintosh 8500 uses the same motherboard as PowerMacintosh 7500 but utilizes the Fridgidare product design used in the Quadra 800, 800AV and the Power Mac 8100/80.
- PowerMacintosh 9500 is a 6-slot deskside machine that uses a variant of the Fridgidare product design called Sub-Zero. PowerMacintosh 9500 does not integrate video in or out and is targeted at the publishing market.

In this document, PowerMacintosh 8500/7500 is used to refer to all three products; PowerMacintosh 8500, PowerMacintosh 7500, and PowerMacintosh 9500 unless otherwise stated. Table 2 shows the feature comparisons between the three initial PowerMacintosh 8500/7500 Products.

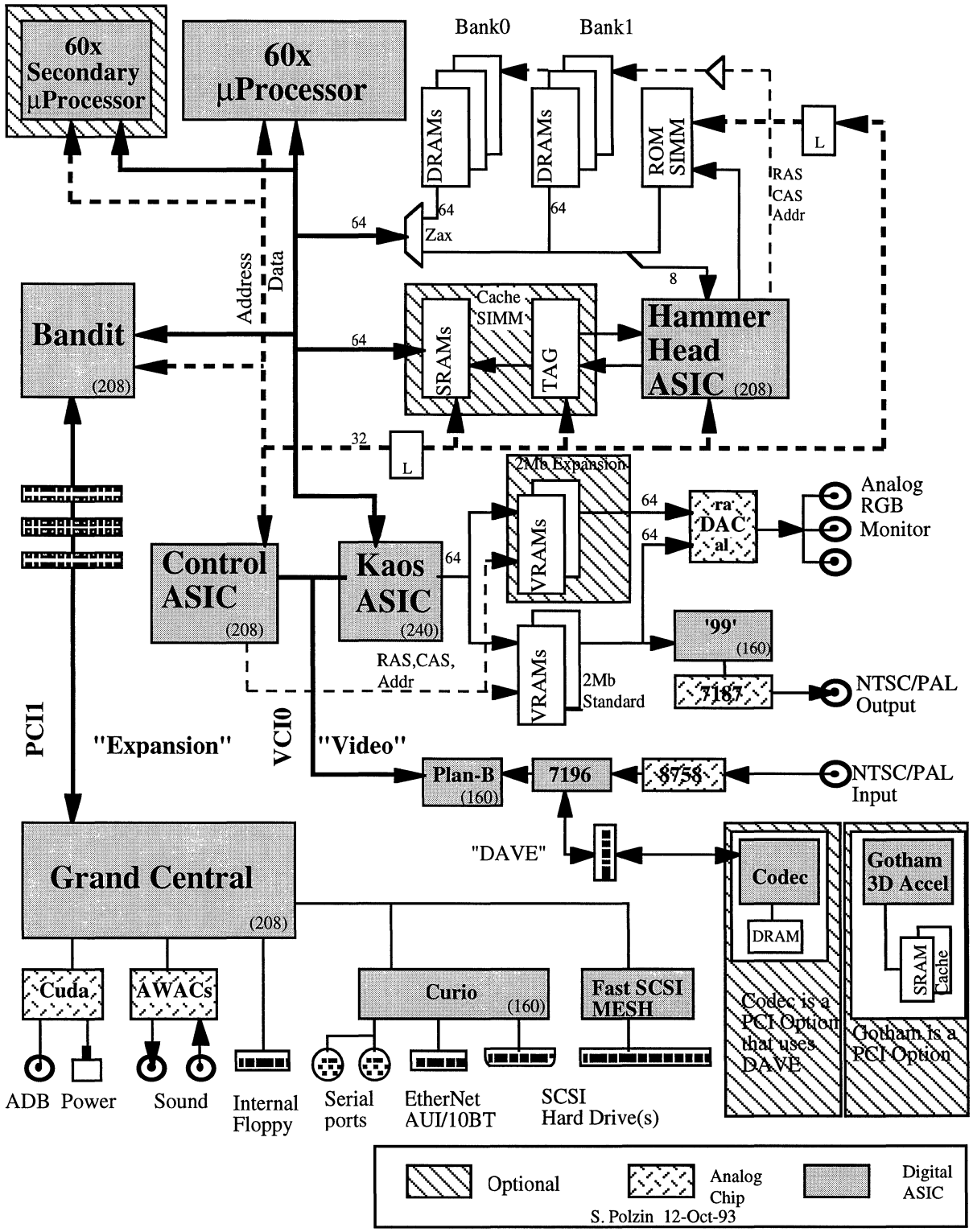
TABLE 2 PowerMacintosh 8500/7500 Family Feature Comparison

	PowerMacintosh 7500	PowerMacintosh 8500	PowerMacintosh 9500
Form Factor	Outrigger	Fridgidare	Sub-zero
Power Supply	150 Watts (DNA)	225 Watts (Crushed Ice)	225 Watts (Crushed Ice)
µ-Processor	100MHz 601	120MHz 604	120/132MHz 604
Base DRAM	8MB	16MB	16MB
DRAM Sockets	8	8	12
Cache	optional (256Kb-4MB)	256Kb (Standard) -4MB	512Kb (Standard) -4MB
Base VRAM	2MB	2MB	Surfer (2MB)
VRAM Sockets	4	4	Surfer (1 or 2MB)
Video In	Standard (24bpp)	Standard. (24bpp)	none
Sound	16 bit in/out 44.1KHz	16 bit in/out 44.1KHz	16 bit in/out 44.1KHz
Video Out	Stuffing option	Standard 24bpp w/convolution	none
Networking	AAUI Ethernet and 10-BaseT	AAUI Ethernet and 10-BaseT	AAUI Ethernet and 10-BaseT
Hard Drive	250-1GB	250-2GB	500-2GB
GeoPort™	2 ports	2 ports	2 ports
Expansion bays	1 5 1/4, 1 3 1/2	1 5 1/4, 2 3 1/2	1 5 1/4, 2 3 1/2
Video connector	DB-15	DB-15	DB-15
SCSI	Internal Fast, External	Internal Fast, External	Internal Fast, External
Slots	3 PCI,	3 PCI	6 PCI

Figure 1 shows the block diagram of the PowerMacintosh 8500/7500 products.

Rev. Level	ECN #	Approved	Date	Revision Description
A	34359	Steve Polzin	05-Jul-95	Initial Release

FIGURE 1 PowerMacintosh 8500/7500 Block Diagram



1.1.1 Related Documents

- PowerMacintosh 8500/7500 HammerHead ASIC ERS: James Kelly et.al.
- PowerMacintosh 8500/7500 Control/Kaos ASIC ERS: Eric Baden et.al.
- PowerMacintosh 8500/7500 Bandit ASIC ERS: Mike Regal et.al.
- Grand Central ASIC ERS: Kevin Christiansen et.al.
- PowerMacintosh 8500/7500 Plan-B ASIC ERS: Bruce Eckstein et.al.
- PowerMacintosh 8500/7500 Ninety9 ASIC ERS: Larry Thompson
- MacRISC Architecture Document: Ron Hochsprung ed.

1.2 Base Feature Set

The base feature set of the initial PowerMacintosh 8500/7500 product is:

- MacRISC Architecture Compliant
- 601, 603, 604 µProcessor support, field upgradable via card
- 50MHz to 150MHz µProcessor support, 33MHz to 50MHz system support
- Built-in framebuffer for 21" 24BPP Monitor
- VideoIn with Phillips Chip via DBDMA in Plan-B
- Dual Stream VideoOut with convolution for flicker-free graphics on interlaced monitors
- AV Connector panel for ease of connection (looks just like the back of a consumer VCR)
- PCI-based Local-Bus for video subsystem
- PCI V2.0 compliant expansion (3 slots in Outrigger, 3 slots in Fridge, 6 slots in SubZero)
- Grand Central DBDMA I/O: 16-Bit Stereo Sound in/out, ADB, SCSI, MFM or GCR Floppy, WorldPort, AAUI and 10Base-T Ethernet
- Second Fast SCSI channel for internal drives
- 8 to 1GB DRAM
- "Green Machine" Features (CPU Clock shutdown, Monitor shutdown etc.)
- Little Endian support

Optional Features of the initial PowerMacintosh 8500/7500 product are:

- 256K, 512K, 1M, or 4M WriteBack L2 cache
- Codec for VideoPhone or QuickTime Acceleration as a PCI option with DAVE access
- Gotham for 3-D (Escher) acceleration as a PCI option

1.2.1 Green Machine Features

The Environmental Protection Agency has published guidelines for certification of computers as "Green". This certification is called "Energy Star" and requires the computer to provide a mechanism to reduce its power consumption (as measured at the line cord) to less that 30W. Monitors are considered separate components and have a similar 30W mode requirement. Currently most low-end Macs conform to this requirement during normal operation. The

initial plan for "Energy Star" certification for mid to high end machines is to simply add an init that shuts down the machine using the soft-power feature of existing power supplies if the machine has been idle for some user-defined elapsed time AND the machine is running the Finder.

PowerMacintosh 8500/7500 is using a more sophisticated scheme to get to the 30W "Energy Star" certification. Hard power numbers are not yet available for PowerMacintosh 8500/7500. The features we hope to enable via PowerMacintosh 8500/7500 specific software (system extension and/or control-panel most-likely) are:

1. 603 and 604 Nap mode: 604 goes from ~12Watts to approximately less than 1W
2. SCSI SpinDown: All SCSI drives used in PowerMacintosh 8500/7500 support the spindown command. This saves the spindle motor energy. Apple is working with drive manufacturers to support an "electronics-off" command to save R/W head power etc. Similarly CDROMs can be placed in a low-power mode.
3. As a result of the µprocessor nap mode, the memory system (both DRAM and VRAM) is not accessed thereby reducing power.
4. To put the monitor in "Energy Star" mode, software invokes the VESA HSYNC/VSYNC protocol to put the monitor into low-power mode.
5. While the monitor is off, the VRAM serial clock and DAC clocks are set to low frequency, thereby reducing power.

Note: 601 Based PowerMacintosh 8500/7500 Systems
The 601 does not implement nap mode and therefore PowerMacintosh 8500/7500 systems that use the 601 will use the soft-power shut-off scheme for Energy Star compliance.

These power savings need to be quantified as the design progresses. The goal is not to go to the expense of a full PowerBook-style power management scheme but rather, to take advantage of existing control features to reduce power.

1.2.2 Little Endian Support

PowerMacintosh 8500/7500 supports both big and little endian mode PowerPC µprocessors. The PCI expansion and Video subsystems are specified and designed using little-endian notation. The ARBus and memory system are specified and designed using big-endian notation. The translation between the naming conventions occurs in the Bandit and Control/Kaos ASICs as a function of address space and mode bits. Microprocessor access to PCI I/O space are endian selected as a function of a mode bit in the Bandit and Control/Kaos ASICs. Accesses by PCI masters to the rest of PowerMacintosh 8500/7500 are endian selected as a function of mode bits in the Bandit and Control/Kaos ASICs (see Section 4.2.3 on page 43).

1.2.3 Quadra 840/AV Video Feature Comparison

PowerMacintosh 8500/7500 improves the rich set of video in and out capabilities of the Quadra 840/AV while at the same time providing a unified frame buffer model to software. Table 3 lists the VideoIn and VideoOut features of the Quadra 840/AV and PowerMacintosh 8500/7500.

TABLE 3 Quadra 840/AV and PowerMacintosh 8500/7500 VideoIn and VideoOut Feature Comparison

Feature	Quadra 840/AV	PowerMacintosh 8500/7500	Comments
Inputs	Composite and S-Video	Composite and S-Video	Both use the Philips 8708/09 A/D converters and 7196-style multistandard decoder. Both support S-SVideo Cameras.
Input Standards	Interlaced PAL, NTSC, and SECAM	Interlaced PAL, NTSC, and SECAM	
Color Space Conversion	8BPP GrayScale, 16BPP ARGB, 16BPP YUV	8BPP Gray-scale, 16BPP ARGB, 16BPP YUV, 32BPP ARGB	
Window ReSize	Decimation	Decimation	
CODEC Port	DAVE	DAVE	
Clipping	Yes	Yes	Quadra 840/AV uses Alpha Plane Clipping, PowerMacintosh 8500/7500 uses a separate in-memory clip mask data-structure.
Outputs	Composite and S-Video	Composite and S-Video	
Output Standards	Interlaced PAL, NTSC	Interlaced PAL, NTSC	Quadra 840/AV uses Mickey. PowerMacintosh 8500/7500 uses Philips 7187 DENC.
Video Output Convolution	1 to 8BPP only	8, 16 and 32BPP	Quadra 840/AV supports 1-8bpp convolution with Video In disabled. PowerMacintosh 8500/7500 allows for separate convolution of video and graphics (see "Ninety9" ERS) in all pixel depths.
Dual Stream Video Out	No	Yes with Expansion VRAM installed	In Quadra 840/AV and PowerMacintosh 8500/7500 with base VRAM, graphics goes blank when using video out. In PowerMacintosh 8500/7500 with expansion VRAM installed, graphics stays "live". PowerMacintosh 8500/7500 supports Video Genlock mode.

1.3 Block Diagram Description

The block diagram shown in Figure 1 represents the initial PowerMacintosh 8500/7500 motherboard implementation. The core ASICs are developed to allow for product evolution.

1.3.1 Memory and Cache (HammerHead ASIC)

The PowerMacintosh 8500/7500 memory and cache system is controlled by the "HammerHead" ASIC. This ASIC contains the ARBus arbiter (system bus controller), 128-bit DRAM controller, ROM access controller, and write-back L2 cache controller. The ARBus arbiter controls five ARBus masters and five ARBus slaves. Arbitration is fixed with the Control/Kaos ASIC node receiving the highest priority (see HammerHead ERS for details).

HammerHead controls a 128-bit wide DRAM system to provide low-latency main memory access and improve overall system DRAM bandwidth. The DRAM controller contains configuration registers that allow differing system speeds and DRAM speeds to be supported. The HammerHead ASIC supports memory system sizes up to 2GBytes. At 50Mhz system bus speed with 60ns DRAMs, HammerHead supports a 6:1:2:1 access with 128-bit wide (interleaved) DRAMs.

DRAM size and configuration options are shown in Table 4. The initial PowerMacintosh 8500/7500 motherboard has eight industry standard 64-bit DRAM SIMM sockets and no on-board DRAMs. SIMMs are populated one at a time. When two SIMMs are found that are the same size, they are configured by software to be a single 128-bit bank (interleaved), increasing memory performance.

TABLE 4 PowerMacintosh 8500/7500 Memory Sizes Supported

Dram Density	SIMM Configuration	SIMM Capacity	Maximum Memory with 8 SIMMs (PowerMacintosh 8500/7500)	Maximum Memory with 12 SIMMs (PowerMacintosh 9500)
4 Meg	512K x 64	4MBytes	32MBytes	48MBytes
4 Meg	1M x 64	8MBytes	64MBytes	96MBytes
4 Meg	2M x 64	16MBytes	128MBytes	192MBytes
16Meg	1Mx64	8MBytes	64MBytes	96MBytes
16Meg	2Mx64	16MBytes	128MBytes	192MBytes
16Meg	4Mx64	32MBytes	256MBytes	384MBytes
64Meg	8Mx64	64MBytes	512MBytes	768MBytes
64Meg	16Mx64	128MBytes	1GBytes	1.5GBytes

HammerHead controls access to the system ROM for both reading and writing (FLASH). PowerMacintosh 8500/7500 uses exactly the same physical ROM SIMM as PowerMacintosh 81/71/6100. Hammerhead contains minimal ROM access timing programmability. Registers in HammerHead are accesses via a single byte of the ARBus (see Chapter 2 on page 18 for addressing).

The HammerHead L2 cache controller uses synchronous burst mode SRAMs that are used in PowerMacintosh 81/71/6100 and (via a metal mask option) Intel 486/Pentium machines. The initial 256K cache is constructed with four 32Kx16 parts. At 50MHz, the L2 cache provides a 2:1:1:1 access for hits. Write-back is used to further reduce the usage of the DRAMs by the µprocessor. The L2 cache controller does not implement a write-through mode and ignores the state of the processor WT (write-through) pin that is asserted for accesses to pages marked write-through in the TLB/BATs. As the L2 cache fronts memory (and not PCI memory space where all frame buffers are located), this behavior is consistent and coherent with the MacRISC defined memory access model (see Appendix B).

1.3.2 ARBus to PCI Bridge (Bandit ASIC)

Bandit, the PowerMacintosh 8500/7500 PCI bridge chip, provides an interface from the main system ARBus and the PCI. Bandit provides a simple bridge function, buffering and translating transactions from one bus to the other. An external PCI Arbiter (Gated Clocks ASIC) provides the following priority:

1. Grand Central (Highest)
2. Slots 1-3 (Round Robin)
3. Bandit Master (lowest)

The PCI and ARBus are operated asynchronously: PCI at 33MHz and ARBus up to 50Mhz. Bandit responds to addresses as defined in Chapter 2 on page 18. Bandit supports the full PCI bandwidth and also supports burst transfers in both directions (ARBus <-> PCI) up to 32 bytes in length (cache block size).

The ARBus is defined to be big-endian byte ordering while the PCI is defined to be little endian byte ordering. Bandit performs the appropriate byte "swapping" and address swizzling. Byte addressing is similarly numbered in conflicting order on either side of the bridge. Bandit supports two modes of byte swapping (See Section 4.2.3 on page 43).

1.3.3 I/O System (Grand Central ASIC)

The PowerMacintosh 8500/7500 I/O system consists of a central core of traditional Macintosh I/O and additional PCI options. The traditional Macintosh Core I/O services are controlled by the Grand Central ASIC. Grand Central includes core support for:

- Cuda via VIA interface (like PowerMacintosh 81/71/6100)
- AWACS stereo audio in/out
- SWIMIII floppy controller
- 16-bit bus to Curio and an 8-bit slow speed bus to RaDACal, Ninety9 etc.
- Second FAST SCSI controller (MESH)

Ethernet, SCC and SCSI interfaces are provided by Curio. All internal I/O core and Curio core access is controlled by a central DMA controller implemented in Grand Central. Grand Central uses the MacRISC compliant DBDMA architecture. Grand Central also contains the MacRISC compliant central system interrupt collection registers.

1.3.4 Expansion

PowerMacintosh 8500/7500 uses the industry standard PCI bus as its local and expansion buses. Three slots are provided for PCI cards. PCI is a 32-bit multiplexed address/data bus. See Chapter 4 on page 39.

1.3.5 Graphics and Video (Control/Kaos ASIC)

The PowerMacintosh 8500/7500 graphics and video system supports the baseline Quadra 840/AV feature set while presenting a unified frame buffer model to software. PowerMacintosh 8500/7500 implements a unified graphics/video framebuffer in lieu of the split frame buffer model of Quadra 840/AV at the request of software. The Control/Kaos ASIC controls the built-in frame buffer and provides a Bridge function to the Video-In ASIC, Plan-B.

1.3.5.1 Frame Buffer

The PowerMacintosh 8500/7500 Graphics and Video ASIC, Control/Kaos controls the main system framebuffer. This framebuffer is constructed with 2Mbit VRAMs configured with a 64-bit port to the ARBus and Plan-B. The backside serial chain of the VRAMs is configured as a 128-bit datapath to the RaDACal high-performance DAC. RaDACal supports the high frequency dot clock of 21" monitors and also provides HW support for an Apple HW cursor. 2MBytes of VRAM are soldered to the PowerMacintosh 8500/7500 motherboard providing 21" 16BPP monitor support. Expansion VRAM sockets allow VRAM expansion up to 4MBytes providing full 21" 32BPP monitor support. The expansion VRAM together with RaDACal also allow Control/Kaos to support a simple double-buffered framebuffer that provides tear-free updates from rendering engines (Gotham). See Table 5 for graphics, double buffering and video-out support details.

1.3.5.2 Video PCI Bridge

Control/Kaos provides an ARBus to PCI bridge function patterned after the Bandit implementation. The resulting PCI is used for Video In via the Plan-B ASIC. This Video PCI defined by Control/Kaos differs from the expansion PCI defined by Bandit as follows:

1. The Video PCI is synchronous with the ARBus (up to 50MHz operation)
2. The Video PCI supports 1 external node, Plan-B.

1.3.5.3 Video Input

PowerMacintosh 8500/7500 VideoIn uses the Philips A/D (8758) and decoder (7196) chips. The resulting 8-bit grayscale, 16 or 32BPP RGB, or YUV video data is placed in the appropriate pixmap via a MacRISC compliant DBDMA engine implemented in the Plan-B ASIC. VideoIn RGB data can be stored in the framebuffer (display) with a 1BPP clipmask or sent to off-screen pixmaps in main memory (DRAMs). Clipping into the frame buffer is accomplished using the DBDMA read channel in Plan-B. This provides video play-through mode with clipping of obscured regions and menus in addition to basic titling. When VideoIn data is aimed at a backbuffer pixmap in DRAM, software is expected to perform any clipping and blending with the QuickDraw CopyBits routine when it moves the pixmap to a visible region of the display. The alpha channel is preserved on VideoIn data streams to allow software blending. There is no hardware support for alpha blending.

1.3.5.4 Plan-B Video In DBDMA Behavior

Plan-B provides DMA service to the 7196 decoder/scalar chip. Two DBDMA channels are provided. The DBDMA write channel which takes data from the 7196 pixel FIFO, attaches a suitable (via the DBDMA channel program)

address and performs a PCI write operation. The DBDMA read channel fetches 1BPP clipmask information from main memory.

1.3.5.5 Second Stream Video Output

PowerMacintosh 8500/7500 VideoOut uses the Ninety9 convolver ASIC and Phillips 7187 DENC to create high-quality interlaced video output for an interlaced monitor or "print-to-video" applications. Ninety9 uses an internal line store and the high video bandwidth provided by the VRAM SAM ports to implement convolution. "Ninety9" performs the RGB to YUV color space conversion, allowing the convolution to be performed in the YUV space. The 7187 digital encoder takes YUV square pixels from "Ninety9" and encodes them into appropriate NTSC or PAL composite video. Table 5 lists the usage restrictions with large monitors, double buffering, and second stream video-out. Second stream video-out can be configured to mirror the main graphics frame buffer (dual writes) or can be configured to display a unique image (just like a separate frame buffer). This implementation of VideoOut improves on the Quadra 840/AV video out user experience: with expansion VRAM installed, the graphics monitor remains "live" while using interlaced video-out, no re-boot is necessary between mode changes.

TABLE 5 PowerMacintosh 8500/7500 Video Out Frame Buffer Usage Options

VRAM Size	Single Buffered Graphics Monitor	Double Buffered Graphics Monitor	Video Out
2MB On-Board	16" @ 24BPP or 21" @ 16BPP	None	None
2MB On-Board	None	None	16BPP Convolved NTSC/PAL Unique or 24BPP Convolved NTSC/PAL Unique
2MB On-Board + 2MB Expansion	21" @ 24BPP	None	None
2MB On-Board + 2MB Expansion	21" @ 16BPP	None	16BPP Convolved NTSC/PAL Mirror or Unique
2MB On-Board + 2MB Expansion	16" @ 24BPP	None	24BPP Convolved NTSC/PAL Mirror or Unique
2MB On-Board + 2MB Expansion	None	21" @ 16BPP or 16" @ 24BPP	

1.3.6 PowerMacintosh 8500/7500 ASIC Development Summary

Table 6 summarizes the ASICs and custom chips being developed for the initial PowerMacintosh 8500/7500 products:

TABLE 6 PowerMacintosh 8500/7500 ASIC Development Summary

ASIC	Vendor	Gate Count	Frequency	I/O Types	Package	Power?
HammerHead	TI	30K	33-50MHz	CMOS	208PQFP	1.1W
Zax	TI	4K	33-50MHz	CMOS	160PQFP	0.5W
Bandit	TI	23K	33-50MHz	CMOS,PCI	208PQFP	1.5W
Grand Central	TI	55K	33MHz	CMOS,PCI	208PQFP	1W
Control	VLSI	35K	33-50MHz	CMOS,PCI	208PQFP	1.5W
Kaos	VLSI	45K	33-50MHz	CMOS,PCI	240PQFP	1.5W
Plan-B	TI	35K	33-50MHz	CMOS,PCI	160PQFP	1.2W
"Ninety9"	VLSI	TBD	50/26Mhz	CMOS	160PQFP	1W
MESH	VLSI	12K	50MH	CMOS/SCSI	100PQFP	1W
Gated Clocks	TI	600	33MHz	CMOS	28PLCC	200mW

1.4 PowerMacintosh 8500/7500 Software Model

The programming model presented to software follows the MacRISC architecture including DBDMA. This section describes some of the implementation details.

1.4.1 DBDMA for Standard I/O

Grand Central contains 10 DBDMA channels to support the "Standard" PowerMacintosh 8500/7500 I/O devices. ADB is accessed via a standard VIA interface (just like PowerMacintosh 81/71/6100). These DBDMA control registers are located in the Grand Central PCI1 memory space defined in Table 7 on page 18. Grand Central Device registers are located in PCI memory space defined in Section 2.4.1.1 on page 21.

1.4.2 DBDMA for Optional I/O

Optional PCI I/O devices have dedicated DMA engines with separate channels. Firewire (IEEE 1396) and Gotham (3-D) also conform to the MacRISC DBDMA specification. It is unclear if add-on SCSI or networking PCI chips will follow the DBDMA model. These DMA control registers are located in the given PCI devices PCI1 memory space as defined in Table 9 on page 21.

1.4.3 DBDMA for Video In

Two DBDMA channels are implemented in Plan-B. One performs VideoIn from the 7196 (DBDMA Write Channel), The second provides ClipMask sourcing. All DBDMA channels can access the entirety of the MacRISC address space for destinations (DBDMA Write Channel) and sources (ClipMask source). Clipping of video data streams is only supported to the PowerMacintosh 8500/7500 on-board VRAM frame buffer defined by the Control/Kaos ASICs. These DBDMA control registers are located in the Plan-B PCI0 space as defined in Table 7 on page 18.

1.5 Options

The PowerMacintosh 8500/7500 core ASICs are designed to facilitate both card-level option bundling for the first generation PowerMacintosh 8500/7500 product as well as follow-on PowerMacintosh 8500/7500 motherboard variants.

1.5.1 Baseline PowerMacintosh 8500/7500 Options

The following options are expected to be available as after-market upgrades or distribution-site bundling via the PCI.

1.5.1.1 CODEC

PowerMacintosh 8500/7500 supports a superset of the Quadra 840/AV DAVE (Digital Audio Video Expansion) connector. This interface allows a CODEC to send YUV video data through the 7196 color-space conversion and window resizer for display (via DBDMA) on the main graphics screen. The CODEC would be mounted on a PCI card and would use the PCI to receive the compressed video data stream as well as using the PCI slot for z-axis space, cooling and power. The DAVE connector on PowerMacintosh 8500/7500 is implemented as an extension to the normal PCI connector on one PCI slot.

The choice of CODEC is a function of the application. Currently, MPEG compress/decompress CODECs are available as well as numerous telephony based VideoPhone formats. As the compression standards become clear, the correct CODEC choice will be made, at which time, a version of the PowerMacintosh 8500/7500 motherboard will be created that solders down the CODEC and associated logic.

1.5.1.2 3-D Acceleration

The Gotham Chip accelerates 3-D rendering software. Gotham is a PCI chip and will initially be mounted on a short PCI card with associated SRAM for geometry information caching. Gotham uses an on-board DBDMA engine to directly write 3-D pixels into the system pixmaps (backbuffers or the display). Gotham connects to PowerMacintosh 8500/7500 via the VCI (PCI0) for direct access to the frame buffer.

1.5.1.3 FireWire

IEEE 1396 Serial Bus can be implemented as a PCI option. FireWire has adopted the MacRISC compliant DBDMA scheme that is being implemented in the Pele chip.

1.5.2 PowerMacintosh 9500 Surfer Video Card

PowerMacintosh 9500 does not have any on-board video circuitry. Rather, the monitor is driven by a 3rd party card named Surfer. The 3rd party has not been selected yet.The feature set requirements for the Surfer Video Card are:

1. PCI 2.0 compliance
2. 24bpp support up to 1152x870 resolution monitors
3. DB-15 standard Apple monitor connector
4. High performance, Quickdraw acceleration

CHAPTER 2

PowerMacintosh 8500/7500 Memory Map

2.1 PowerMacintosh 8500/7500 Memory Map Overview

PowerMacintosh 8500/7500 adheres to the MacRISC architecture memory map. Separate address spaces are allocated for DRAM, ROM, system control registers, PCI Configuration Space, PCI I/O space, NuBus Card (via a bridge) space and PCI Expansion card space. There are at least two different DBDMA engines in PowerMacintosh 8500/7500 and while the DBDMA register definition is constant (as defined in the MacRISC DBDMA Architecture Document), the locations are device specific. The general view of the PowerMacintosh 8500/7500 memory map is shown in Table 7.

TABLE 7 PowerMacintosh 8500/7500 Memory Map

Start Address	End Address	Name	Comments
0x00000000	0x7FFFFFFF	Main Memory	2GB of main memory. Linearly accessed
0x80000000	0xBFFFFFFF	PCI Memory Space	1GB of PCI Memory Space. PCI cards are located here as well as all Control/Kaos Frame Buffer space.
0xC0000000	0xEFFFFFFF	PCI Memory for NuBus Super Slot Space	PCI Memory space used for NuBus Super Slots accessed via a PCI to NuBus Bridge. This space supports 3 Nubus Super Slots (C, D E). Addressing details are shown in section 2.6.
0xF0000000	0xF1FFFFFF	PCI0 I/O Space	PCI0 in PowerMacintosh 8500/7500 contains the Video SubSystem. PCI0 Addressing details are shown in section 2.4.

TABLE 7 PowerMacintosh 8500/7500 Memory Map

Start Address	End Address	Name	Comments
0xF2000000	0xF3FFFFFF	PCI1 I/O Space	PCI1 in PowerMacintosh 8500/7500 and PowerMacintosh 9500 contain the Expansion and I/O Sub-System. Addressing details are shown in section 2.4.
0xF4000000	0xF5FFFFFF	PCI2 I/O Space	PCI2 is not used in PowerMacintosh 8500/7500 but is used in PowerMacintosh 9500 as the second expansion PCI. Addressing details are shown in section 2.4.
0xF6000000	0xF7FFFFFF	PCI3 I/O Space	PCI3 is not used in PowerMacintosh 8500/7500.
0xF8000000	0xF8FFFFFF	System Control	System Control Registers are located here. System ID lives here. See section 2.3.
0xF9000000	0xFEFFFFFF	PCI Memory for Nubus Slot Space	PCI Memory Space used for normal Nubus Slots. This space supports 6 Nubus Slots (9,A,B,C,D,E). Addressing details are shown in section 2.6.
0xFF000000	0xFFFFFFFF	ROM Space	16 MB of ExecutableROM space, See section 2.2.2

2.2 Memory-Like Space

Any size operand up to 32-bytes may be used to access Memory-Like space. Memory-like space is defined as DRAM, ROM and PCI Memory space. Only DRAM and ROM space are kept co-herent by the PowerMacintosh 8500/7500 hardware (HammerHead and the 60x µprocessor). It is the responsibility of operating system software to maintain the cache coherence of the built-in Frame Buffer space and any PCI Memory Space.

2.2.1 Main Memory (DRAM)

The Address Map allocates 2 GB of space for "DRAM" memory. This space is responded to on the ARBus by the HammerHead ASIC. On both the expansion and video mode Bandit/Control/Kaos chips, memory transactions that appear on PCI0, PCI1, or PCI2 with addresses that fall in this space are be forwarded onto the ARBus for servicing by HammerHead.

2.2.2 ROM

The Address Map allocates 16 MB of space for "ROM". This space is responded to on the ARBus by the HammerHead ASIC. Memory transactions that appear on PCI0, PCI1, or PCI2 with addresses that fall in this space are for-

warded onto the ARBus for servicing by HammerHead. ROM is populated starting from the top of memory as shown in Table 8.

TABLE 8 ROM Population

ROM Size	Starting Address
4MB	0xFFC00000
8MB	0xFF800000
12MB	0xFF400000
16MB	0xFF000000

2.3 System Registers

ASICs that interface to the ARBus in PowerMacintosh 8500/7500 have control and status registers mapped to 0xF8000000. The PowerMacintosh 8500/7500 system ID register lives in this space in the HammerHead ASIC. System Configuration accesses are limited to a 4-byte operand size. Access to System Configuration space with operand sizes of other than 4-bytes results in undefined PowerMacintosh 8500/7500 system behavior.

Most devices contain internal registers which must be addressed, typically with low-order address bits. To allow simple systems to be built, where only the minimal number of byte lanes (usually, just one) need be "touched", the "stride" for registers is made 16 bytes. This handles the widest conceivable bus (at least, for the systems we are likely to build for the lifetime of MacRISC). It also allows a simple hex interpretation of register numbers (i.e., one nibble to the left of an address).

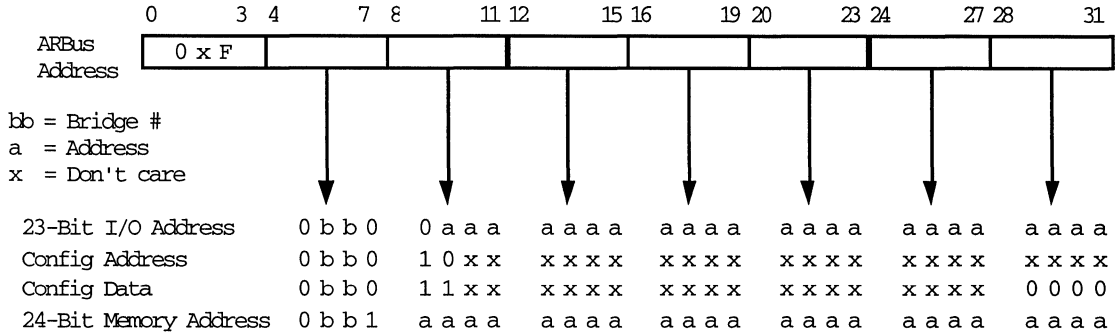
2.4 PCI I/O Space

PowerMacintosh 8500/7500 uses two separate PCI busses for video (PCI0) and expansion (PCI1) while PowerMacintosh 9500 uses two PCI busses for expansion (PCI1 and PCI2). Table 7 shows the major allocation of space for 4 ARBus-PCI bridges. This section shows how these ranges are further sub-allocated to allow the production of all possible PCI cycles from the processor. The upper 7 bits of an address are determined by the PCI Bridge number in the set of 0xF0, 0xF2, 0xF4, 0xF6. The next 3 bits are used to encode the desired PCI cycle type as shown in Figure 3.

2.4.1 Pass-Through Memory Cycles

A Pass-Through Memory Cycle is made when an ARBus address has A[7] == 0b1. In this case, the bridge passes the original ARBus address during the Address Phase. This effectively defines an 8 MB Pass-Through Memory region per ARBus-PCI bridge.

FIGURE 3 ARBus PCI Bridge Sub-Addressing



2.4.1.1 Grand Central Device Registers

Grand Central implements a large number of registers to control the various I/O devices. Additionally, to support backwards compatibility for VIA register offsets, 128KBytes of address space is consumed. To accommodate these requirements while not consuming large amounts of PCI memory space, Grand Central uses the pass-through memory space defined in Section 2.4.1 on page 20.

TABLE 9 PowerMacintosh 8500/7500 Grand Central Device Register Mapping

Start Address	End Address	Name	Comments
0xF3000000	0xF301FFFF	Grand Central Device Registers	Grand Central Device registers. Mapped to 0F3000000 to 0xF301FFFF on PCI1 for memory commands.

Note: Grand Central Base Address
Grand Central is fully PCI compliant and therefore can re-position its memory space response to anywhere in PCI memory space. The addresses defined in Table 9 are defined for convention only (defined by the current OpenFirmware and Expansion Manager code for the PowerMacintosh 8500/7500 ROM releases). Grand Central may be mapped anywhere in PCI Memory space.

2.4.2 I/O Cycles

PCI I/O Cycles are made when a PCI bridge is accessed with an address with A[7:9] == 0b000. An access to this address generates an I/O cycle on the PCI side of the bridge passing the low-order 23 bits from the ARBus address. Software should guarantee that the relevant, device specific upper bits (other than the bridge select bits) are zeros. Architecturally, the PCI spec defines PCI I/O cycles to have only the lower 16 bits of the address be significant. The extra bits are provided for architectural expansion and hardware simplicity.

ARBus read or write accesses by the CPU to the PCI I/O space defined in Table 7 are serviced by the appropriate Bandit/Control/Kaos chips and forwarded onto the appropriate PCI as I/O read/write commands. Bandit/Control/Kaos drives the a low-order 16-bits of the ARBus address onto the PCI during the command cycle to access up to 64K worth of PCI I/O space. Access to PCI I/O space must occur after each PCI node has been configurated via the

PCI configuration space. PowerMacintosh 8500/7500 system, behavior is undefined if PCI I/O space is accessed BEFORE the PCI nodes are properly configured.

2.4.3 Configuration Cycles

Configuration Cycles are generated in an indirect manner, similar to mechanism #1 suggested in the PCI specification for x86 class processors. This mechanism uses two registers (Config Address and Config Data), accessed using normal ARBus Cache-Inhibited, single-beat Reads and Writes using 32-bit operands. These registers are located in the bridges PCI I/O Space as shown in Table 10.

TABLE 10 PCI Bridge Config Address and Config Data Locations

ARBus Address	PCI Bridge	Register
0xF0800000	Control/Kaos PCI 0	PCI0 Config Address
0xF0C00000	Control/Kaos PCI 0	PCI0 Config Data
0xF2800000	Bandit PCI 1	PCI1 Config Address
0xF2C00000	Bandit PCI 1	PCI1 Config Data

The first register is the Config Address register. This register contains the address to be used during the Address Phase of a subsequent PCI Config cycle. Two types of config cycles are defined for the PCI. For config cycles that are destined for the PCI directly defined by Bandit, Type 0 Config cycles are generated. For Config cycles that are destined for a hierarchical bridge via a PCI to PCI bridge chip, Type 1 Config cycles are generated. In both cases, the same register in Bandit/Control/Kaos is used. Bandit/Control/Kaos drives the contents of the Config Address register directly on the PCI AD lines, un-modified.

The second register involved with Config Cycles is the Config Data register. This is actually a "pseudo-register" in the sense that it does not physically exist within the bridge. An access to this "register" causes a Config (or, Special) Cycle to be made on the PCI side, using the address contained within the Config Address register.

The type of Config cycle (Config Read / Config Write) depends upon the cycle accessing the Config Data "register". The data for a PCI Config Write Cycle is specified by the data being written to the Config Data "register". Data read from the during a PCI Config Read Cycle is the value returned for a read of the Config Data "register".

2.4.3.1 Type 0 Config Cycles

For Type 0 Config Cycles, where the device being accessed is on directly on the PCI bus side of the bridge. When the Config data register is accessed (read or Write), a corresponding PCI config cycle (read or write) is generated with the PCI wires asserted directly from the Config address register as defined in Figure 4

FIGURE 4 Type 0 Config Address Register Format

31:11	10:8	7:2	1	0
ID Sel	Function Number	Register Number	0	0

TABLE 11 Type 0 Config Address Register Description

Field	Bits	Type	Initial State	Description
IDSEL	31:11	R/W	Undef	Specifies which of PCI AD[31:11] is to be asserted during the config cycle, thereby asserting IDSEL# to the appropriate PCI device. Only one of [31:11] should be set in this field. If more than one bit is set when Config Data is accessed, PowerMacintosh 8500/7500 system operation is undefined. The mapping of IDESEL bits to devices is shown in Table 14 and Table 13.
Function Number	10:8	R/W	Undef	Specifies which of Config or Special cycles is to be generated when the Config Data register is accessed. If [10:8] = 0b0, PCI configuration cycles are generated. If [10:8] = 0b111, special cycles are generated with the PCI AD[31:0] wires driven with the contents of the Config Data register.
Register Number	7:2	R/W	Undef	Specifies which of 64 configuration registers in the specified PCI device is to be accessed.

2.4.3.2 Type 1 Config Cycles

Type 1 Config Cycles are used in hierarchical PCI configurations. In this case, the device to be accessed is not on the PCI bus side of the bridge, but, instead, is located across a PCI-PCI bridge which is on the local PCI bus. For Type 1 Config Cycles, the Config Address register Bus Number field is non-zero. When the Config data register is accessed (read or write), a corresponding PCI config cycle (read or write) is generated with the PCI wires asserted directly from the Config address register as shown in Figure 4.

Note: Type1 and Control/Kaos
Control/Kaos do NOT support Type 1 Config cycles in the VCI.

FIGURE 5 Type 1 Config Address Register Format

31:24	23:16	15:11	10:8	7:2	1	0
0	Bus Number	Device Number	Function Number	Register Number	0	1

TABLE 12 Type1 Config Address Register Description

Field	Bits	Type	Initial State	Description
Zeros	31:24	R/W	Undef	This field must be zero for Type 1 Config cycles.
Bus Number	23:16	R/W	Undef	Specifies which logical PCI bus in the hierarchy is the ultimate target of this configuration cycle. When Bus Number = 0b00, the PCI directly behind this bridge is specified and PowerMacintosh 8500/7500 system operation is undefined. When Bus Number != 0x0, another PCI is specified and a configuration cycle is generated with AD[31:2] asserted with exactly the contents of this register and AD[1:0] = 0b01.
Device Number	15:11	R/W	Undef	Specifies which of PCI AD[31:11] is to be asserted during the resulting Type 0 config cycle on the target bus, thereby asserting IDSEL# to the appropriate PCI device.
Function Number	10:8	R/W	Undef	Specifies which of Config or Special cycles is to be generated when the Config Data register is accessed. If [10:8] = 0b0, PCI configuration cycles are generated. If [10:8] = 0b111, special cycles are generated with the PCI AD[31:0] wires driven with the contents of the Config Data register.
Register Number	7:2	R/W	Undef	Specifies which of 64 configuration registers in the specified PCI device is to be accessed.

2.4.3.3 Special Cycles

Special Cycles are generated using a modification of the Config Cycles. When the Config Address register gets written with a value such that the Bus Number is zero, the Device Number is all 1's, the Function Number is all 1's, and the Register Number has a value of zero, then the bridge is primed to do a Special Cycle the next time the Config Data register is written. When the Config Data register is written, the bridge generates a Special Cycle encoding on the C/BE[3:0] wires during the address cycle, and drives the data from the write data to the Config Data register onto AD[31:0] during the first data cycle.

Interrupt Acknowledge cycles are required for some PCI chips that are used to the Intel 8259 interrupt controller or x86 processor environment. On PowerMacintosh 8500/7500, IACK cycles are generated by setting up the Config Address register as above to generate Special Cycles. In this case, a read from the Config Data Register after setting up the Config Address register forces a PCI IACK cycle to be generated with the data returned on the ARBus being the x86 interrupt "vector".

2.4.3.4 PCI Device Number Mapping

On each PCI in the PowerMacintosh 8500/7500 system, the PCI AD wires are connected to device and connector IDSEL# pins via 22Ω resistors. The mapping for PCI0 (Video PCI or VCI) is shown in Table 13, the mapping for PCI1 (Expansion) is shown in Table 14 and the mapping for the PowerMacintosh 9500 second expansion PCI is shown in Table 14. The Bandit/Control/Kaos bridge can be assigned either Device Number 0 or 1 as a function of the source of the PCI config cycle. If Bandit/Control/Kaos is the master, AD[11] is used. If Bandit/Control/Kaos is the target, AD[12] is used. This allows Bandit/Control/Kaos to be used in a Portable Docking configuration that has two Bandits on the same PCI bus (one in the Duo, the other in the Dock).

TABLE 13 PCI0 Device Number Mappings (PowerMacintosh 8500/7500 Only)

Device Number	PCI AD connection	VCI0 Device
0/1	AD[11] or [12]	Control/Kaos
2	AD[13]	Plan-B

TABLE 14 PCI1 Device Number Mappings (PowerMacintosh 8500/7500 and PowerMacintosh 9500)

Device Number	PCI AD connection	PCI1 Device
0/1	AD[11] or [12]	Bandit
2	AD[13]	Expansion Slot A
3	AD[14]	Expansion Slot B
4	AD[15]	Expansion Slot C
5	AD[16]	Grand Central

2.5 PCI Memory Space

ARBus read or write accesses by the CPU to the PCI Memory space defined in Table 7 are serviced by the appropriate Bandit/Control/Kaos chips and forwarded onto the appropriate PCI as Memory read/write commands. Bandit/Control/Kaos passes the ARBus Address on to the PCI address un-modified. The range of addresses responded to by Bandit/Control/Kaos from the ARBus is defined by the Address Mask register. This register defines sixteen coarse (256MB) and sixteen fine (16MB) regions of the 32-bit address space that Bandit/Control/Kaos responds-to/ignores-from the ARBus and PCI. Bandit/Control/Kaos implements a simple decode/anti-decode scheme whereby, if a given

address is decoded to be passed from the ARBus to the PCI, the same address is not be allowed to pass from the PCI to the ARBus. The coarse and fine grain decodes are defined in Figure 6.

FIGURE 6 Bandit/Control/Kaos Address Mask Register Definition

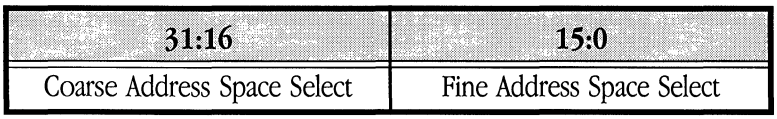


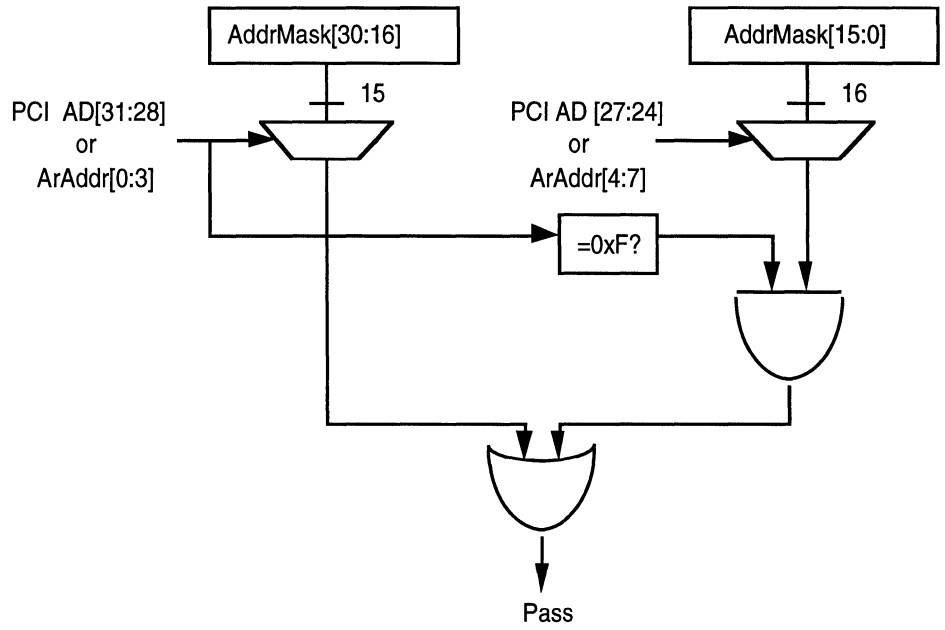
TABLE 15 Bandit/Control/Kaos Address Mask Register Definition

Field	Bits	Type	Initial State	Description
Coarse Address Select	31:16	R/W	Undef	This field defines what coarse (256MB) regions of memory are passed from the ARBus to the PCI. If a given bit in this field is set, Addresses on the ARBus with the matching bits ArAddr[0:3] are passed from the ARBus to the PCI while the same address values on the PCI, AD[31:28], are not passed to the ARBus. If a given bit in this field is not set, addresses on the ARBus with the matching bits [0:3] are not passed from the ARBus to the PCI while the same address values on the PCI, AD[31:28], are passed to the ARBus. Figure 7 illustrates this decode function.
Fine Address Select	15:0	R/W	Undef	This field defines what fine (16MB) regions of memory are passed from the ARBus to the PCI when ARBus address [0:3] = 0xF. If a given bit in this field is set, Addresses on the ARBus with the matching bits [4:7] are passed from the ARBus to the PCI while the same address values on the PCI are not passed to the ARBus. If a given bit in this field is not set, addresses on the ARBus with the matching bits [4:7] are not passed from the ARBus to the PCI while the same address values on the PCI are passed to the ARBus. Figure 7 illustrates this decode function.

Note: 0xFxxxxxxx Decoding
When Bandit BaseAddr0 [31] is set, the relevant bits in [15:0] must also be set to access any of the 0xFxxxxxxx space. If [15:0] == 0, the state of [31] is ignored.

It is the responsibility of initialization software to insure that the multiple copies of Bandit/Control/Kaos do NOT have overlapping PCI memory space register definitions. Bandit/Control/Kaos drive the full 32-bits of the ARBus address onto the PCI during the command cycle. It is the responsibility of initialization software to configure the relevant PCI device base address registers appropriately to match the address range enabled in the given Bandit/Con-

FIGURE 7 Bandit/Control/Kaos Decode Details



trol/Kaos chip. Access to PCI Memory space must occur after each PCI node has been configurated via the PCI configuration space. PowerMacintosh 8500/7500 system behavior is undefined if PCI Memory space is accessed BEFORE the PCI nodes are properly configured. PCI Memory space accesses are supported up to a 32-byte operand size.

2.5.1 PCI DMA Traffic

PCI Masters that initiate DMA transfers on either PCI0, PCI1 or PCI2 "see" the same address map as the ARBus (µprocessor) for memory-like addresses (DRAM, ROM, PCI Memory) and are accessed with PCI Memory read/write commands with a 1:1 PCI to ARBus address mapping. PCI masters that need to access I/O registers in another node on the same PCI simply generate I/O read/write PCI commands with the appropriate 16-bit address. PCI masters that need to access I/O registers in another node on the other PCI generate memory read/write PCI commands with the address corresponding the desired PCI node/register as defined in Table 7. Bandit/Control/Kaos passes these transactions onto the ARBus where they are serviced by the other Bandit/Control/Kaos and therefore the appropriate PCI node.

2.6 PCI Memory and NuBus

As part of a strategy of providing transition support for current users and developers, the Address Map allows a PCI-to-NuBus Bridge implemented as a PCI card that is cabled to an external NuBus chassis. All "NuBus" accesses are directed through this Bridge card to the NuBus chassis. Likewise, parent-board accesses from NuBus cards are made on their behalf by the Bridge card.

2.6.1 Normal NuBus Slots

PowerMacintosh 8500/7500 supports six "normal" NuBus slot address spaces: 9 through E. These are mapped into the traditional NuBus slot space at 0xF9aaaaaa to 0xFEaaaaaa. To access these slot addresses through a PCI to NuBus Bridge, the Bandit/Control/Kaos that defines the PCI that the PCI to Nubus Bridge resides on must have its Base Address Register 0 configured with at least the following bits set: [31] and [14:9].

2.6.2 NuBus Super Slots

PowerMacintosh 8500/7500 supports three NuBus Super Slot address spaces: C through E. These are mapped into the traditional NuBus SuperSlot space at 0xCaaaaaaa to 0xEaaaaaa. To access these slot addresses through a PCI to NuBus Bridge, the Bandit/Control/Kaos that defines the PCI that the PCI to Nubus Bridge resides on must have its Base Address Register 0 configured with at least the following bits set: [30:28].

Note: No Slot F

MacRISC ROM space exists in what would normally be NuBus Slot F. However, the Mac II model has always assumed that the maximum slot number is E; hence, there is no compatibility problem.

2.7 Non-ASIC Registers

A number of registers that are not implemented in ASICs are accessed via the General Bus (GBus) defined by the Grand Central ASIC and appear in the Grand Central Address space. Grand Central provides chip select outputs for up to seven external "devices" on the GBus. These devices are mapped as shown in Table 16.

TABLE 16 Gbus Device Mapping

GBus Device #	Address	Device	Reference
0	0xF3019000	Ethernet ROM	Section 2.7.1 on page 30
1	0xF301A000	Generic Board Register1	Section 2.7.2 on page 30
2	0xF301B000 to 0xF301B030	RaDACal Registers	Section 2.7.3 on page 31
3	0xF301C00 to 0xF301C30	Ninety9 Registers	Section 2.7.6 on page 34
4	0xF301D00	NVRAM Address Latch	Section 2.7.4 on page 32
5	0xF301E00	Generic Board Register2	Section 2.7.2 on page 30
6	0xF301F00	NVRAM	Section 2.7.4 on page 32

2.7.1 Ethernet ROM

The Ethernet ROM is assigned GBus device number 0. It is implemented as a single byte-wide device. Successive locations in the Ethernet ROM are located 16 bytes apart. The Ethernet ROM is found starting at address 0xF3019000.

2.7.2 PowerMacintosh 8500/7500Generic Board Register1

The Generic Board Register 1 is read only and accumulates misc. bits of information from the expansion bus, MotherBoard and system enclosure as shown in Figure 8.

FIGURE 8 Generic Board Register 1 Definition

15:13	12:11	10	9	8	7	6	5:4	3:2	1:0
Stuffing Config	BoxID	Ethernet 10BaseT Link	Mic Sense	EMMO	SCC Port B RTS	SCC Port A RTS	PCISlot C PRSNT	PCISlot B PRSNT	PCISlot A PRSNT

TABLE 17 Generic Board Register 1 Description

Field	Bits	Type	Initial State	Description
Stuffing Config	15:13	Read Only	As Defined	This field defines what optional devices are present on a given motherboard. Bit 15 -> Reserved, pulled high Bit 14 -> 0 = No Second SCSI. 1 = MESH Second SCSI Bit 13 -> 0 = VideoOut Stuffed. 1 = No VideoOut
Box ID	12:11	Read Only	As Defined	This field is directly driven from the Box ID pins on the motherboard. Software uses this field to put up the right picture for "About This Macintosh". This field is encoded as follows: 0b00 -> Econoline Box (PowerMacintosh 8500/7500) 0b01 -> Fridge Box (Nitro), SubZero Box (PowerMacintosh 9500) 0b1x -> Reserved
Ethernet 10Base-T Link	10	Read Only	As Defined	This bit is set whenever the Ethernet 10Base-T circuitry detects that the motherboard RJ-45 connector has been attached to a live 10Base-T hub.
Mic Sense	9	Read Only	As Defined	This bit is set when the mini stereo plug inserted into the motherboard audio-in jack is a microphone level source.
EMMO	8	Read Only	As defined	This bit is used by manufacturing for "Extended Mac Minus One" testing.

TABLE 17 Generic Board Register 1 Description

Field	Bits	Type	Initial State	Description
SCC Port B RTS	7	Read Only	As Defined	This bit reflects the state of the RTS wire from SCC port B. It is used by DMA LocalTalk Software.
SCC Port A RTS	6	Read Only	As Defined	This bit reflects the state of the RTS wire from SCC port A. It is used by DMA LocalTalk Software.
PRSNT	5:4 3:2 1:0	Read Only	As defined	This field is driven by PCI cards to specify how much power they intend to draw from the given slot. This field is encoded as follows: 0b11 No Board in Slot 0b01 Board present, 25W maximum power 0b10 Board present, 15W maximum power 0b00 Board present, 7.5W maximum power Software uses this field to verify that the maximum of 50W alloted to the three slots is not exceeded.

2.7.3 RaDACal

The RaDACal CLUT/DAC is programmed via an 8-Bit interface. All of its directly accessible registers are located 16 bytes apart. Most of the RaDACal registers are accessed through an address register (Similar to the 85C30 SCC). Only four ARBus and PCI addresses are used to access RaDACal. The internal Address register is used to index into the complete internal state of the part as shown in Table 18.

TABLE 18 RaDACal Register Addressing

PowerMacintosh 8500/7500 Address	Register	Description
0xF301B000	Address Register	This register is loaded with an 8-bit address to index into the Cursor and Color Palette RAMS as well as select the various control registers.
0xF301B010	Cursor Palette RAM	Using the values 0x0 to 0x7 in the address register, this location accesses the Cursor Palette RAM.

TABLE 18 RaDACal Register Addressing

PowerMacintosh 8500/7500 Address	Register	Description
0xF301B020	Control Registers	Using the decode (below), this location accesses the following registers: 0x10 -> Cursor Position [11:8] 0x11 -> Cursor Position [7:0] 0x20 -> Control Register 0x21 -> Double Buffer Control Register 0x22 -> Test Control Register 0x30 -> Red Test Register 0x31 -> Green Test Register 0x32 -> Blue Test Register
0xF301B030	Color Palette RAM	Using the values 0x00 to 0xFF in the address register, this location accesses the Color Palette RAM.

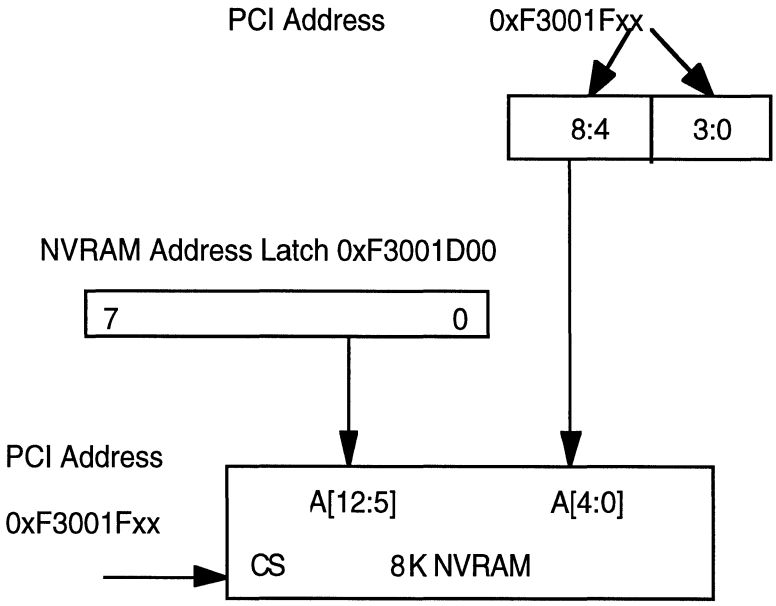
2.7.4 NVRAM Address Latch Register

MacRISC OpenFirmware booting requires a sizable array of non-volatile RAM to store boot parameters etc. The PowerMacintosh 8500/7500 family of machines support 8K bytes of non-volatile RAM that is accessed in the Grand Central GBus space. Addressing the entire 8K bytes requires an index register to be loaded for every set of 32 locations. Each location is a byte wide and is located on 16-byte boundaries. See Figure 9 and Figure 10. NVRAM is located in the same space in both PowerMacintosh 8500/7500 and PowerMacintosh 9500.

FIGURE 9 NVRAM Address Latch Register Definition

15:8	7:0
Reserved	NVRAMAddress[12:5]

FIGURE 10 NVRAM Addressing



2.7.5 PowerMacintosh 8500/7500Generic Board Register2

The Generic Board Register 2 is a logical extension of GBR 1. This register is read only and accumulates misc. bits of information from the expansion bus, MotherBoard and system enclosure as shown in Figure 8.

FIGURE 11 Generic Board Register 2 Definition

15:8	7	6	5:4	3:2	1:0
Reserved	Composite Video Out Sense	SVideo Out Sense	PCISlot E PRSNT	PCISlot D PRSNT	PCISlot C PRSNT

TABLE 19 Generic Board Register 2 Description

Field	Bits	Type	Initial State	Description
Composite Video Out Sense	7	Read Only	As defined	When this bit =0, it indicates that a suitable plug has been inserted in the composite Video output connector on the AV Connector panel.
SVideo Out Sense	6	Read Only	As defined	When this bit = 0, it indicates that a suitable plug has been inserted in the S-Video output connector on the AV Connector panel.
PRSNT	5:4 3:2 1:0	Read Only	As defined	This field is driven by PCI cards to specify how much power they intend to draw from the given slot. This field is encoded as follows: 0b11 No Board in Slot 0b01 Board present, 25W maximum power 0b10 Board present, 15W maximum power 0b00 Board present, 7.5W maximum power Software uses this field to verify that the maximum of 50W alloted to the three slots is not exceeded.

2.7.6 Ninety9 Registers

Ninety9 has an 8 bit programming port, including 8 bits for data, one read/write signal, a /CS signal (chip select), and two hardware address signals providing four addresses. Two of these addresses are dedicated to the CLUT, and the other two are dedicated to the cursor memory and control registers as show below. Please refer to the Ninety9 ERS for programming details.

- 0xF301D000 -> CLUT data port: data port for reads and writes of CLUT
- 0xF301D010 -> CLUT address port: pointer into the CLUT, auto post incrementing
- 0xF301D020 -> Cursor/register data port data port for reads and writes of cursor memory and registers.

• 0xF301D030 -> Cursor/register address port pointer to cursor memory locations and control registers, auto post incrementing for cursor locations, addresses 0-127; not auto post incrementing for address 128 and above

2.8 System Reset

The PowerPC Architecture defines the System Reset vector to be at 0xFFF00100. This vector naturally falls in the ROM space for all configurations of ROM that are at least 4MB is size. No hacks are required to alias this space anywhere.

2.9 Interrupts

All interrupts in PowerMacintosh 8500/7500 are presented to the 60x µprocessor though a central interrupt collection register located in the Grand Central ASIC. Figure 12 illustrates the general interrupt wire routing in PowerMacintosh 8500/7500. Table 20 on page 36 details the mapping of board level interrupts to Grand Central Interrupt registers.

FIGURE 12 PowerMacintosh 8500/7500 Interrupt Wiring

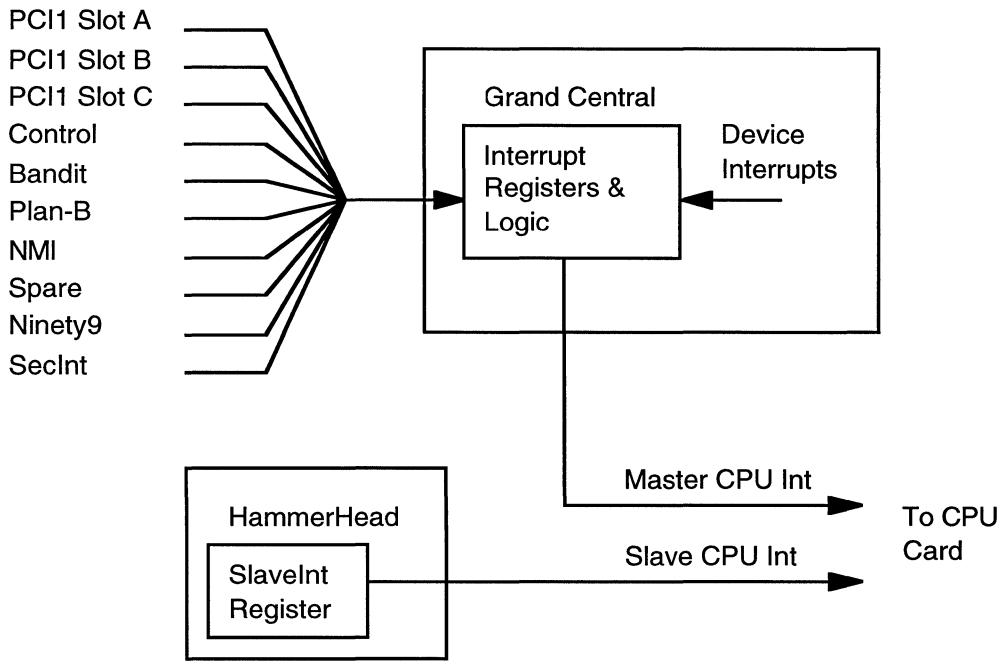


TABLE 20 PowerMacintosh 8500/7500 Interrupt Wiring

Grand Central External Interrupt	Source
External Int 0	Cuda NMI
External Int 1	Not Used
External Int 2	Bandit Error Interrupt
External Int 3	PCI Slot A
External Int 4	PCI Slot B
External Int 5	PCI Slot C
External Int 6	Control (Graphics Monitor Timing)
External Int 7	Ninety9 (Video Monitor Timing)
External Int 8	PlanB (Video Input Timing)
External Int 9	Not Used
External Int 10	MP Interrupt, connected to Ethernet ROM Chip Select

Rev. Level	ECN #	Approved	Date	Revision Description
A	34359	Steve Polzin	05-Jul-95	Initial Release

CHAPTER 3 Initialization & Configuration

3.1 PowerMacintosh 8500/7500 Initialization Overview

The PowerMacintosh 8500/7500 motherboard initialization process can be divided in three separate efforts. The first effort involves configuring the base memory system. This activity is centered on the HammerHead ASIC registers. HammerHead latches at reset, configuration bits that indicate the number of PCI bridges in the given system. These bits are then readable (and writable) by software. HammerHead also controls the L2 cache which resets to the OFF state. Software needs to explicitly enable the L2 cache and set a few modes. HammerHead contains the CPUID register that together with the Generic Board Register, enables the correct little picture to come up in the "About This Macintosh" picture. The HammerHead ERS contains detailed descriptions of the various registers.

Once the base memory interface has been configured, software needs to configure the PCI busses before accessing either the I/O system or frame buffer. Both PCIs are configured via a configuration header space in each PCI node. As the PCI bridges are discovered and PCI nodes behind them discovered, the configuration header space in each node can be filled in by software.

Finally, after getting the appropriate address spaces figured out for the given PCI configuration, software can then continue with the balance of the standard Mac initialization sequence (send the boot sound out of the speaker, initialize the frame buffer, SWATCH, RaDACal etc.) and starting reading the operating system off of the disk.

3.2 PowerMacintosh 8500/7500PCI Configuration

The bulk of the effort to configure a PowerMacintosh 8500/7500 system is to configure both PCI busses. PCI uses a special configuration command to access a defined configuration header in each PCI node/device. In these headers, software can determine the node/device type and assign it an appropriate address space that allows the operating system to generate either register-type accesses (4-byte PCI I/O reads and writes) or memory-like accesses (32-byte PCI Memory reads and writes).

3.2.1 PowerMacintosh 8500/7500 Specific PCI Configuration Space Definitions

Table 2 details the contents of the ID field in the PCI Configuration Header. for all PowerMacintosh 8500/7500 ASICs.

TABLE 2 PowerMacintosh 8500/7500 Specific PCI Configuration Space Definitions

Field	Bandit	Grand Central	Control/Kaos	PlanB
Vendor ID	0x106B	0x106B	0x106B	0x106B
Device ID	0x0001	0x0002	0x0003	0x0004

Notes:

1. Bandit uses additional configuration space for more registers
2. Control/Kaos use BaseAddr0 in a unique manner, see Control/Kaos ERS.
3. Grand Central BaseAddr0 value needs to be loaded by Software
4. PlanB BaseAddr0 value needs to be loaded by Software
5. Control/Kaos Base Addr1 and 2 need to be loaded by Software

3.3 PowerMacintosh 8500/7500 Initialization Flow

Following reset, initialization software needs to perform the following initialization:

1. Read CPUID register to determine that this is a PowerMacintosh 8500/7500 machine.
2. Read Motherboard ID register to determine how many PCI busses are present (and therefore PowerMacintosh 8500/7500 vs. PowerMacintosh 9500).
3. Based on system speed from CPUID, load memory timing register for best performance.
4. Size/Test memory and configure base registers for interleave etc.
5. Based on Motherboard ID start probing the PCIs with configuration commands to see what is out there.
6. As PCI nodes are discovered, configure them and allocate decode space in the appropriate Bandit/ Control/Kaos address mask registers.
7. When Grand Central is found, read the Generic Board Register to determine the box type, EMMO status and the amount of power each slot draws.
8. Configure Grand Central and AWACs to send out the boot sound.
9. Configure the appropriate framebuffer to put up the grey screen.
10. Configure Cuda and get the mouse live.
11. Based on the amount of power the PCI cards want, put up the appropriate sweaty or melted Mac on the screen.
12. When all PCI devices and drivers are configured/loaded, initialize and enable the L2 cache if present.

CHAPTER 4

PowerMacintosh 8500/7500 Expansion

4.1 Introduction

PowerMacintosh 8500/7500 uses the industry Standard PCI bus as its I/O Expansion bus. PCI is a 32-bit multiplexed address and data, non-split 16-33MHz bus. PowerMacintosh 8500/7500 implements two copies of the PCI: One for motherboard I/O and three expansion slots and the other for video in. This chapter defines the expansion PCI.

All Apple designed PCI chips or cards (PCI Devices) are fully PCI compliant. This implies the following:

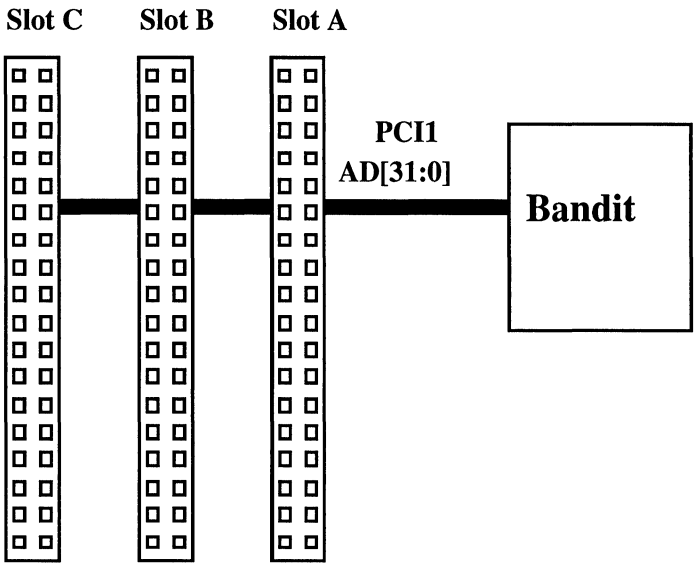
1. Designed and documented in Little Endian Mode
2. Configuration space implemented according to the PCI spec V2.0

4.2 PowerMacintosh 8500/7500 Logical PCI Characteristics

PowerMacintosh 8500/7500 implements the ARBus to PCI bridge in 32-bit mode. All three PCI connectors on the PowerMacintosh 8500/7500 Motherboard connect to the lower 32 bits of the PCI as shown in Figure 1.

Rev. Level	ECN #	Approved	Date	Revision Description
A	34359	Steve Polzin	05-Jul-95	Initial Release

FIGURE 1 PowerMacintosh 8500/7500 Expansion PCI Wiring



4.2.1 PowerMacintosh 8500/7500 PCI Signal Support

PowerMacintosh 8500/7500 supports the required and optional PCI signal pins as follows:

- AD[31:0]
- C/BE[3:0]
- PAR
- FRAME#
- TRDY#
- IDRY#
- STOP#
- DEVSEL#
- IDSEL#
- REQ#
- GNT#
- CLK
- RST#
- INTA#,INTB#, INTC#, INTD# (Wired together)

PowerMacintosh 8500/7500 Does not support the optional PCI wires as follows:

- PERR#
- SERR#
- LOCK#
- 64-Bit extension wires
- Cache Support wires
- JTAG wires

4.2.2 PCI Arbitration

PowerMacintosh 8500/7500 uses the Gated Clocks PCI arbiter which offers the following features:

- One high priority request/grant pair
- Five round robin request/grant pairs
- High priority parking via the ArbCritical wire
- Request parking detection and elimination
- Two high priority grants and Critical arbitration

4.2.2.1 High priority grants

The high priority master is given the grant when requested. Request parking is allowed for the high priority master, but it is discouraged. There are possible deadlocks with the PCI bridge that arise if the high priority master asserts and holds the request line. The proper way to hold the grant is through the use of the arbcritical wire.

4.2.2.2 Critical Arbitration

High priority masters use critical arbitration to keep the grant when it has more transactions to complete soon. Typically it is used for real time data streams that can't take long periods of interruption. When the high priority master wants to keep the grant, it asserts and hold the ArbCritical wire high. The arbiter will automatically detect retries (Stop asserted with no TRdy) and will deassert the grant to the high priority master for at least three clocks. This gives another master the chance to complete an operation that might be necessary to resolve a deadlock situation (such as reads passing on a PCI bridge chip).

4.2.2.3 General Round Robin Grants

There are 5 general round robin grants. The last state of the round robin grant is held whenever the high priority grant is asserted. After the high priority master completes the transaction and deasserts request, the round robin grants return to their previous state. The round robin grants can transition to a different state while the high priority grant is asserted.

Request parking is not allowed. A master is attempting request parking when it holds request to keep the grant. The master will be allowed to keep the grant until another master comes along and requests the bus. Masters will be guaranteed to complete one transaction. This insures that the other 4 masters in the round robin chain are allowed fair time on the bus.

If a master receives a grant, it has 16 clocks of idle time during which to take the bus. Idle time is defined as IRdy and Frame being deasserted (high) on the PCI bus. If the high priority master jumps in during those 16 clocks, the count is held until the round robin grant is reasserted. The count then continues to 16. When the terminal count of 16 is reached, an internal state bit is set to indicate the master is attempting to park. This bit allows the grant state machine to advance to the next requester without allowing a transaction from that master. This feature of the arbiter also allows for broken masters with request lines shorted to ground.

4.2.3 Endianess

The ARBus is defined to be big-endian byte ordering while the PCI is defined to be little endian byte ordering. Bandit performs the appropriate byte "swapping" and address swizzling. Byte addressing is similarly numbered in conflicting order on either side of the bridge. Bandit supports two modes of byte swapping; Big endian mode (default) that maps ARBus and PCI data bytes as shown in Figure 2 and little endian mode (selected via a mode bit) as shown in Figure 3. Table 2 and Table 3 detail the mapping of ARBus to PCI addresses in big endian mode. Table 4 and Table 5 detail the mapping of ARBus to PCI addresses in little endian mode.

FIGURE 2 ARBus <-> PCI Data Byte Swapping in Big Endian Mode

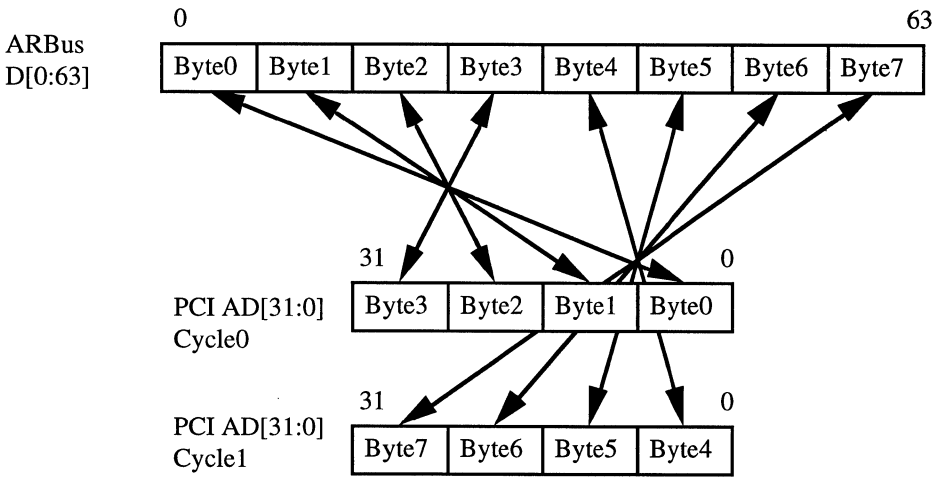


FIGURE 3 ARBus <-> PCI Data Byte Swapping in Little Endian Mode

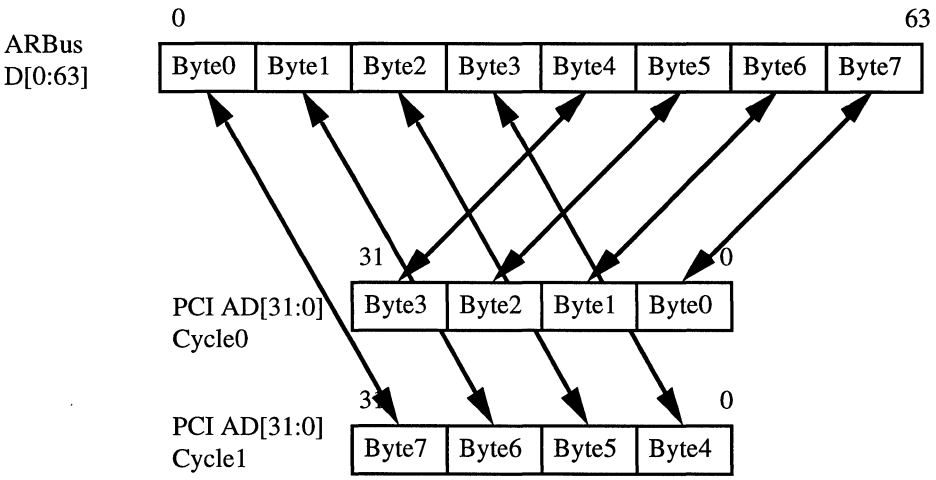


TABLE 2 ARBus to PCI address Mapping in Big Endian Mode for Memory Space

ARBus Address	PCI Address	Comment
[0]	[31]	
[1]	[30]	
..	..	
[28]	[3]	
[29]	[2]	
0b00	[1]	Wrap Mode
0b00	[0]	Wrap Mode

TABLE 3 ARBus to PCI address Mapping in Big Endian Mode for I/O Space

ARBus Address	PCI Address
[0]	[31]
[1]	[30]
..	..
[30]	[1]
[31]	[0]

TABLE 4 ARBus to PCI address Mapping in Little Endian Mode for Memory Space

ARBus Address	PCI Address	Comment
[0]	[31]	
[1]	[30]	
..	..	
[28]	[3]	
[29] XOR 0b1	[2]	Address Swizzle

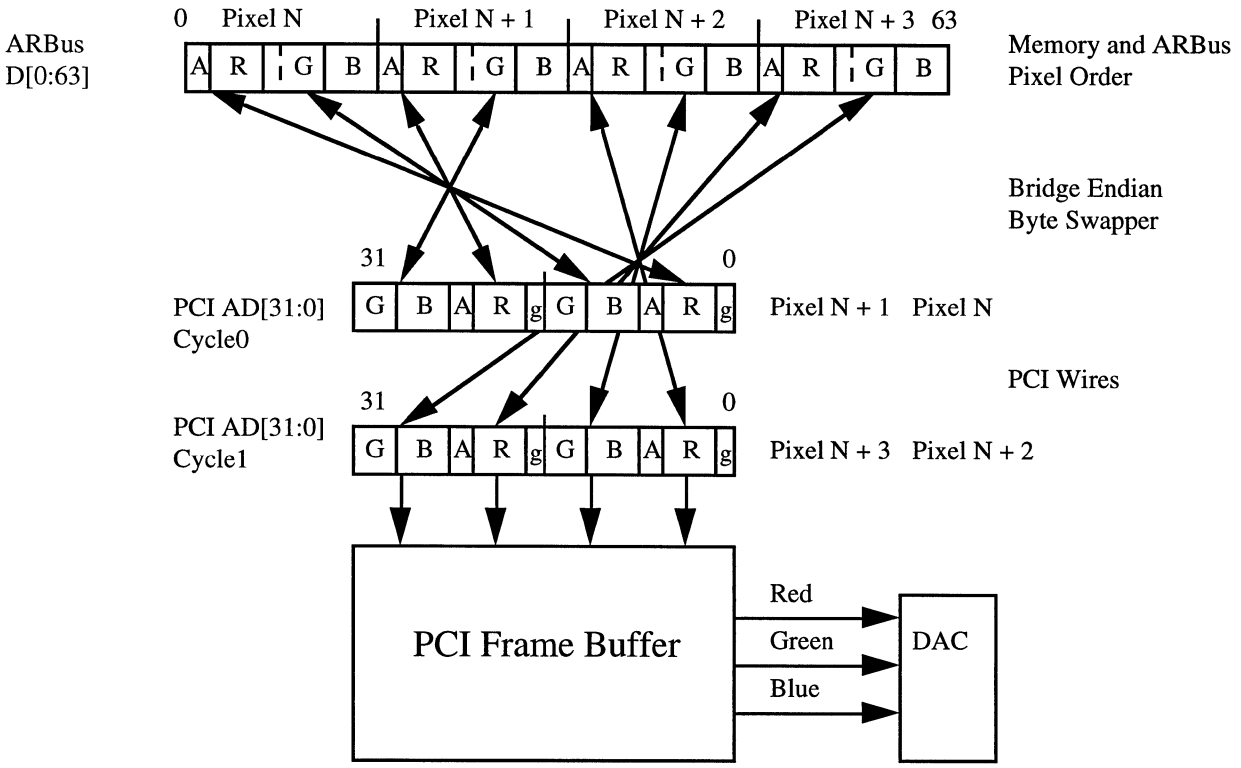
TABLE 5 ARBus to PCI address Mapping in Little Endian Mode for I/O Space

ARBus Address	PCI Address	Comment
[0]	[31]	
[1]	[30]	
..	..	
[28]	[3]	
[29:31]	[2:0]	Address Swizzle for Burst accesses
[29:31] XOR 0b100	[2:0]	Address Swizzle for 32-Bit accesses
[29:31] XOR 0b110	[2:0]	Address Swizzle for 16-Bit accesses
[29:31] XOR 0b111	[2:0]	Address Swizzle for 8-Bit accesses

4.2.3.1 PCI and Pixels

The Address swizzling and byte swapping defined above provides for the correct address invariant transfer of byte stream data between the big endian ARBus and the little endian PCI. For non-byte stream data-types, it is assumed that software provides the appropriate byte swapping facility. The one exception to this convention is video frame buffers. Due to the pervasive practice in Mac software of directly accessing the pixels in the frame buffer, MacRISC and PowerMacintosh 8500/7500 define specific big-endian formats for pixels on the PCI. These formats are shown in Figure 6 on page 48. To retain a consistent view of pixels by software irrespective of the location of the pix-map (main memory or a PCI frame buffer), the byte swapping hardware in Bandit/Control/Kaos is used as shown in Figure 4 and Figure 5.

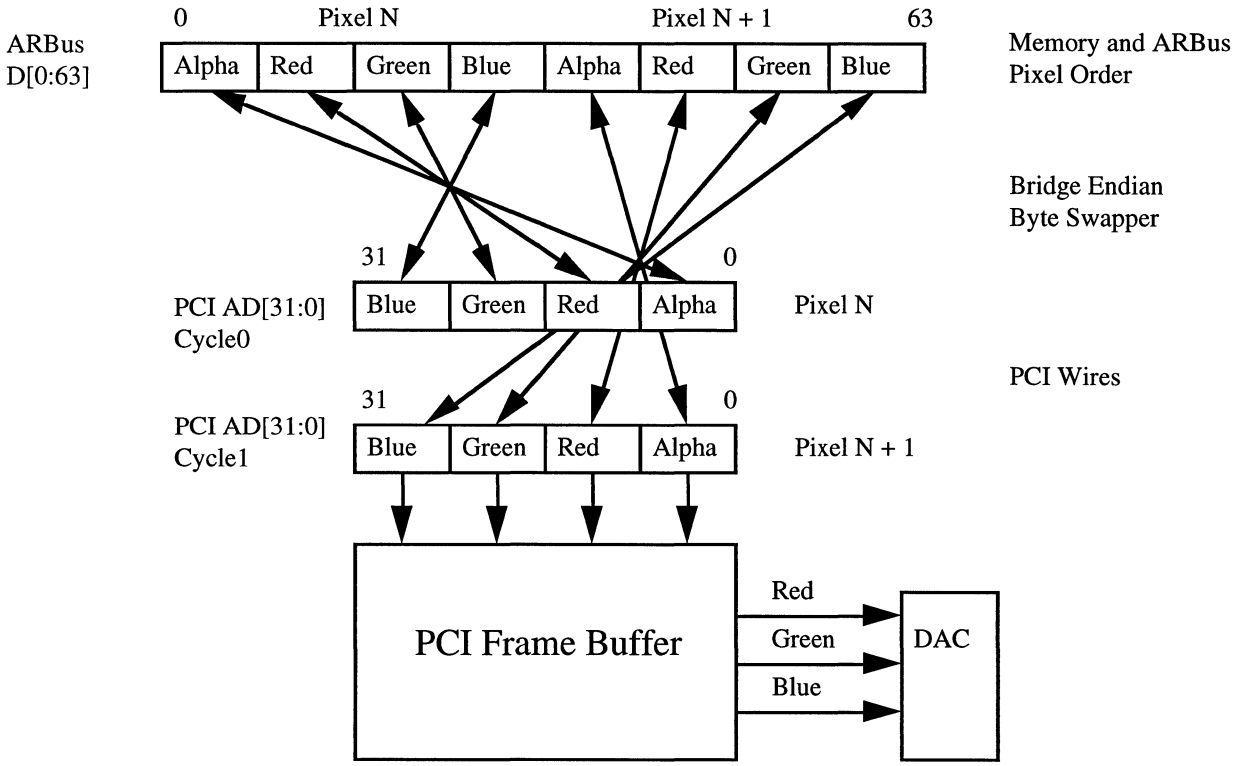
FIGURE 4 16BPP PCI to ARBus Endian Mapping



4.2.3.2 Pixel Formats

The MacRISC Architecture defines pixels to exist in memory in a big-endian format. All access by the processor to pixels in DRAM, on-board-VRAM or expansion card VRAM yield the same results. This forces pixels to be transferred

FIGURE 5 32BPP PCI to ARBus Endian Mapping

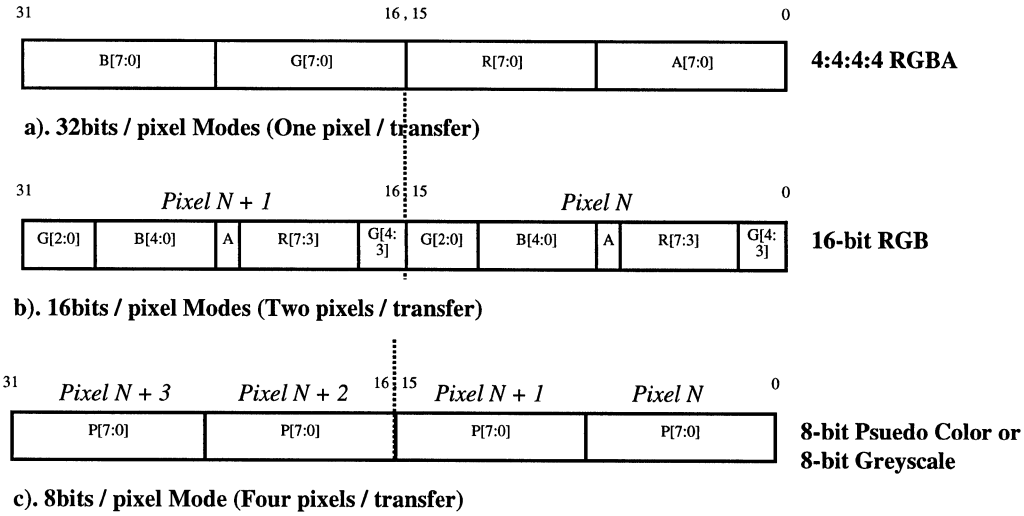


across the PCI wires in the MacRISC specific big-endian format as shown in Figure 6 (also see Section B.3 on page 79).

4.3 PowerMacintosh 8500/7500 Electrical PCI Characteristics

PowerMacintosh 8500/7500 requires all PCI cards and chips to use the 5V signaling standard specified by the PCI spec V2.0. The PowerMacintosh 8500/7500 Expansion PCI always run at 33MHz. The PowerMacintosh 8500/7500 motherboard and enclosures provide 50W of power for three PCI slots. This power may be divided in any ratio between slots and between 5V and 3.3V supply rails. PowerMacintosh 8500/7500 specific software inquires each PCI card at boot time (via the `PRSNT1#` and `PRSNT2#` signals) to verify that the maximum card power is not exceeded.

FIGURE 6 MacRISC PCI Pixel Formats



4.4 PCI Physical Card Definition

The PowerMacintosh 8500/7500 motherboard and enclosures accept standard PCI cards as defined by the PCI spec V2.0. Apple product designs that support PCI require cards to use the standard ISA fence.

CHAPTER 5

PowerMacintosh 8500/7500 Physical Characteristics

5.1 PowerMacintosh 8500/7500 Motherboard Overview

The PowerMacintosh 8500/7500 motherboard is designed to fit in both the OutRigger and Fridgidare product designs. The PowerMacintosh 8500/7500 Motherboard is a 6-Layer board using 6/6.5 routing rules.

5.1.1 Rear Panel Motherboard Connectors

PowerMacintosh 8500/7500 and PowerMacintosh 9500 are designed to have common rear-panel motherboard I/O connections. These are detailed in Table 25.

TABLE 25 PowerMacintosh 8500/7500 and PowerMacintosh 9500 Common I/O Connectors

Function	Connector Type
External SCSI	DB25
AAUI Ethernet	FreindlyNet
10Base-T Ethernet	RJ-45
ADB	4-pin Mini DIN (one)
Video Monitor	DB15 (Not used on PowerMacintosh 9500)

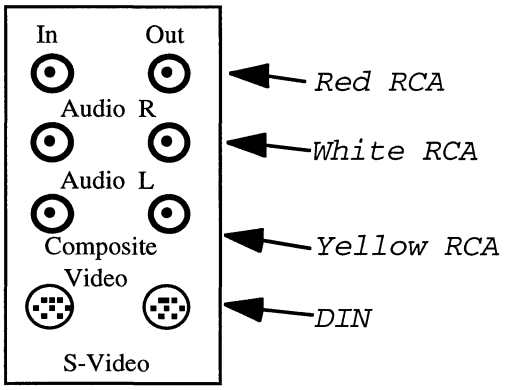
TABLE 25 PowerMacintosh 8500/7500 and PowerMacintosh 9500 Common I/O Connectors

Function	Connector Type
SCC	9-Pin Mini DIN, Stacked, two GeoPorts
Audio	1/8" Mini Stereo, stacked In/Out

5.1.2 PowerMacintosh 8500/7500 AV Connector Panel

PowerMacintosh 8500/7500 uses a seperate connector panel to provide access to the AV features of the machine. This panel is designed to appear very much like the back of a consumer VHS VCR to ease the hook-up of AV Equipment. Figure 19 shows a rough drawing of the AV Connector panel used in PowerMacintosh 8500/7500.

FIGURE 19 PowerMacintosh 8500/7500 AV Connector Panel



5.2 Clocking Strategy

The PowerMacintosh 8500/7500 motherboard has a number of clocking domains that are supplied asynchronous clocks from seperate sources as shown in Table 26. The most timing critical is the ARBus timing which runs up to 50MHz. The expansion PCI bus runs at 33MHz while the Video PCI runs synchronous to the ARBus.

TABLE 26 PowerMacintosh 8500/7500 Clocks

Clock Chip	Reference	Destinations
Beruit	33.333MHz Crystal	PCI 33MHz clocks to three slots, Grand Central, Gated Clocks Arbiter, and Bandit
Athens	31.334MHz Crystal	Dot Clock to RaDACal, 50MHz SCSi clock to MESH, 31.334MHz clock to Grand Central
Grand Central	22.579MHz Oscillator	Internal I/O clocks, RTC, Audio Clock, VIA clock, SCC PClk.

TABLE 26 PowerMacintosh 8500/7500 Clocks

Clock Chip	Reference	Destinations
Curio/LXT	20.00MHz Oscillator	Ethernet clocks
CUDA	32.768K Crystal	CUDA clocks
Beruit/Max	40 and 44MHZ Oscillator	System clocks for 604 120/40 and 132/44MHz systems. 51Ω source series terminated on the Max card
Moto 970/ Lorax	25MHz Crystal	System clocks for 602 100/50MHz systems. 51Ω source series terminatos on the Lorax card.

Rev. Level	ECN #	Approved	Date	Revision Description
A	34359	Steve Polzin	05-Jul-95	Initial Release

Appendix A Caches & MP

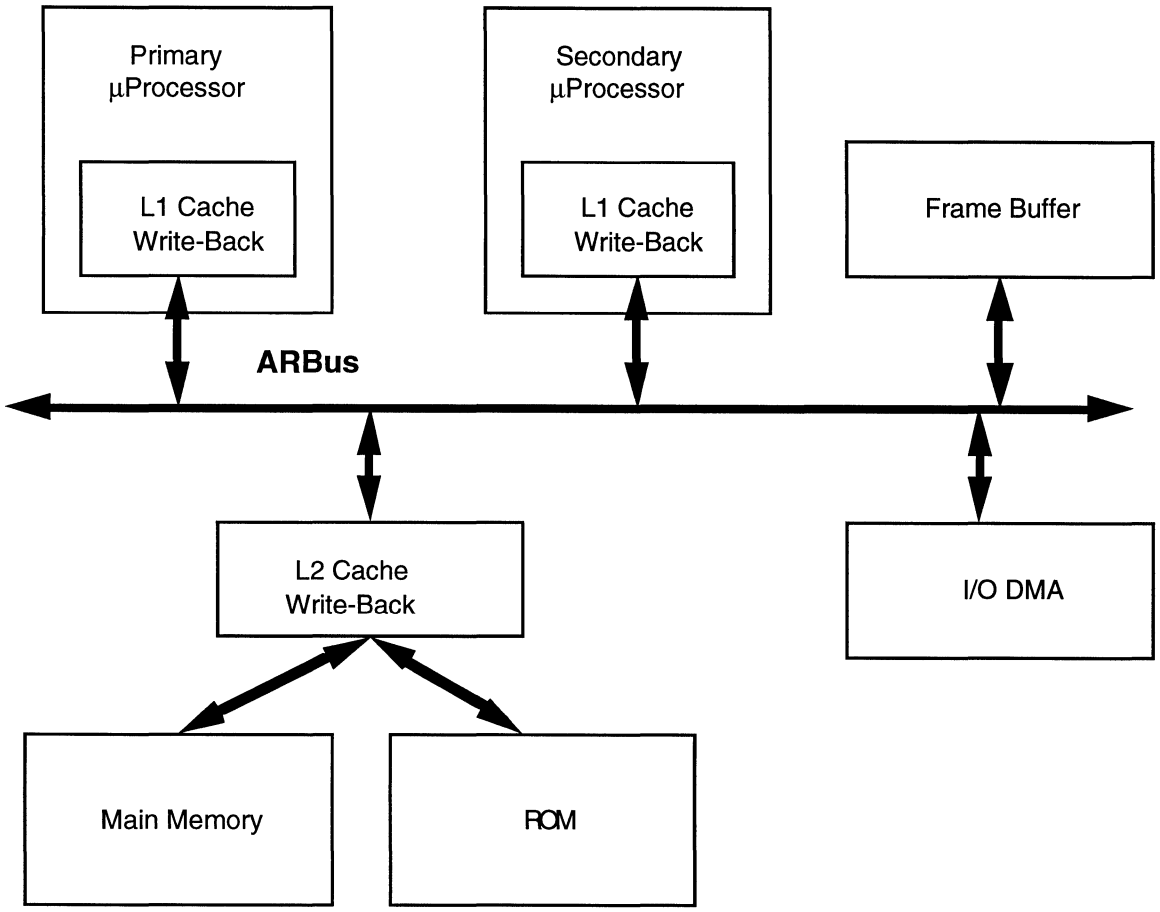
This Appendix discusses the PowerMacintosh 8500/7500 L2 cache and the expected multi-processor behavior.

A.1 Overview

The PowerMacintosh 8500/7500 system uses the 601, 603 and 604 series of PowerPC µprocessors. These chips all employ on-chip-write-back (copy-back) level 1 caches. These caches are kept coherent with the rest of the system using a MESI (Modified, Exclusive, Shared, invalid) protocol defined in the Apple RISC Bus (ARBus) specification. The ARBus defines a protocol for keeping multiple L1 caches coherent as well as providing for coherent DMA. The PowerMacintosh 8500/7500 system provides for an optional level 2 cache that 'fronts' the main memory system. This L2 cache is also a write-back cache. The PowerMacintosh 8500/7500 system provides for a secondary slave processor to be resident on the CPU daughter card. Figure 20 illustrates the Cache structures in PowerMacintosh 8500/7500.

This appendix gives a brief overview of how these level1 (L1) caches interact with the level 2 (L2) cache in the PowerMacintosh 8500/7500 system and how the secondary slave processor is allowed to share data with the main master processor and the rest of the system.

FIGURE 20 PowerMacintosh 8500/7500 Caches



A.1.1 L1 cache

The master processor in the PowerMacintosh 8500/7500 system can be any of the 601, 603 or 604 µprocessors. The L1 cache sizes are shown in Table 27.

TABLE 27 PowerMacintosh 8500/7500 Cache Sizes and Organization

Cache	Size	Sets	Comments
601 L1	32Kb	8 sets, each 4K	Unified I and D
603 L1	8K I, 8K D	2 sets, each 4K for each I&D	Split I and D
604 L1	16K I, 16K D	4 set, each 4K for each I&D	Split I and D

TABLE 27 PowerMacintosh 8500/7500 Cache Sizes and Organization

Cache	Size	Sets	Comments
L2-256	256K	1	I, D, I/O DMA, Video DMA selectable via HH Registers
L2-512	512K	1	I, D, I/O DMA, Video DMA selectable via HH Registers
L2-1M	1Meg	1	I, D, I/O DMA, Video DMA selectable via HH Registers
L2-4M	4Meg	1	I, D, I/O DMA, Video DMA selectable via HH Registers

The master processor L1 cache is always kept co-herent with main memory using the protocol defined in the ARBus specification.

A.1.2 L2 Cache

The level 2 cache in PowerMacintosh 8500/7500 is a simple "far" cache that simply fronts the main memory system. Addresses eligible to be cached include all of DRAM(0x00000000 to 0x7FFFFFFF) and all of ROM (0xFF000000 to 0xFFFFFFFF).

The L2 cache implements a write-back protocol to minimize both read and write traffic to/from DRAMs. The L2 cache obeys the sense of the ARBus GBL and CI wires. When asserted, the Global wire forces the master µprocessor to snoop the given address in the L1 TAG store. To maintain co-herency, if a dirty hit occurs, the L1 cache asserts ARBus ARTRY to abort the transaction and pushes the dirty cache-line to memory. Likewise, when an address with GBL asserted is presented on the ARBus, the L2 cache snoops the L2 tag. If a hit occurs, the data is serviced by the L2 cache and the DRAM or ROM access is aborted gracefully. When GBL is deasserted, both the L1 and L2 TAGs are not snooped and hence the transaction is not kept co-herent.

The ARBus CI (Cache Inhibit) wire is used to prevent a cachable address (i.e. an address that falls in the ranges specified above) from forcing an allocate if the address misses. The CI wire may be asserted by the µprocessor for special areas of memory marked to be non-cachable via the PowerPC BATs registers. CI may be asserted by some other ARBus node (Bandit, BlueFish) to prevent a given transaction from being cached in the L2 cache. CI is asserted or not asserted by Bandit and BlueFish as a function of a configuration bit local to each chip.

A.1.3 General ARBus MP Cache Coherence

The ARBus defines an MP cache protocol that allows for symmetric multi-processing. Each processor will always 'see' the latest copy of a given piece of data. This is accomplished with write-back L1 caches using the MESI protocol. When one processor wants to load a given piece of data from memory into a register (L1 lookup yields an Invalid state), if the data has been modified by another processor and the latest copy is not in main memory, the processor with the most recent copy (L1 lookup yields a Modified state), the processor with the most recent copy asserts ARBus ARTRY to the first processor and pushes the modified data to memory. When the original processor replays the original ARBus transaction to service the original load instruction, memory has the most recent copy.

Additional cache and page states are communicated between processors on the ARBus using the TT, TC and SHARED wires during Address-Only transactions. In general, system hardware does not need to take any action in response to these address only transactions. In particular, the L2 cache has no need to change any state as a function of these address only transactions.

A.2 Software Implications of ARBus Cache Coherence

The cache coherence protocol defined by the ARBus forces a rather high penalty for access to shared modified data. This penalty is typically two to three times the access time for an L1 cache miss or more than twenty times an L1 cache hit. This sort of cache coherence is most useful in a multi-tasking, multi-process environment where each process has very little if any shared data with any other process. This allows the operating system to schedule any process on any processor with the hardware naturally moving a process's written data from cache to cache as the process is re-scheduled across the available set of processors. Given that high-performance RISC systems are already stressing available system DRAM bandwidth, this protocol further limits MP system performance by using DRAM to "stage" the migration of written data between processors.

For a multi-tasking, multi-process environment that allows processes to be multi-threaded, typically large amounts of data are shared and the access to this data is via a set of memory based semaphores (spinlocks). While the number and size of such semaphores is typically small (sometimes as small as a single word), the access to these semaphores defines the scalability of the application as more processors are added to the set of available processors. In this environment, shared written data needs to be shared between processors as opposed to being fought over. Ideally, if no process to processor affinity facility is available, it would be desirable if the small number of semaphores are broadcast to all processor L1 caches so that access penalty is no more than an L1 cache hit.

Since discrimination of semaphore writes vs. general data writes is non-trivial, if all writes were broadcast to all processors, thread specific modified data (temps, loop counters etc.) would appear in all caches in the system. As a counter example, if there is high degree of process to processor affinity (i.e. if process Q was last scheduled on processor N, then the operating system scheduler makes a best effort attempt to schedule process Q back on processor N when it becomes eligible), the high-penalty of sharing written data incurred on the ARBus is not as significant.

For very effective sharing of written data, an intervention protocol (write-update) should be used. Intervention uses system bus. bandwidth to keep caches coherent NOT DRAM bandwidth (in PowerMacintosh 8500/7500, bus bandwidth is 2 to 3 times the available DRAM bandwidth). Given the schedule and resource constraints, PowerMacintosh 8500/7500 does not implement SMP using the ARBus protocol or a new intervention protocol. Rather, a simple asymmetric multi-processing model is implemented as shown in section A.3.

In all cases, MP software should strive to have the following characteristics:

- Lock semaphores should be located at least a cache block apart.
- Relevant data should be located with the semaphores. When ownership of the semaphore is granted, the relevant data is in the cache already.

A.3 PowerMacintosh 8500/7500Asymmetric Multi-Processing

PowerMacintosh 8500/7500 uses a "far" L2 cache with only a single logical "set" to minimize cost and complexity. This L2 cache configuration would be very in-effective for symmetric multi-processing as discussed above (each processors accesses would thrash against the other in the L2). To support multiprocessing, PowerMacintosh 8500/7500 uses an asymmetric configuration where a primary processor is the "Master" and the secondary processor is the "Slave". The primary creates specific tasks for the secondary, starts the secondary and is informed of the results. To prevent L2 thrashing, the L2 cache controller in the HammerHead ASIC does not allocate on misses from the secondary processor. Secondary processor accesses that hit in the L2 are serviced normally. The L1 caches of the primary and secondary processors remain co-herent as described in section A.1.3.

A.3.1 Secondary Processor Scheduling

The following illustrates the mechanism by which the primary processor schedules a job on the secondary processor:

1. Secondary Processor is in "wait for interrupt" loop.
2. Primary processor sets up secondary work block, loads address in "Secondary Start Vector" location and executes a "barrier" operation insuring coherence.
3. Primary writes the HH SecInt register that asserts the interrupt pin on the secondary processor.
4. Secondary processor fields this interrupt and fetches the address and attributes of the process it is to execute via the "Secondary Start Vector". If part of the Primary set-up creates new code for the secondary, the secondary needs to invalidate its I-Cache before proceeding.
5. Primary continues on.
6. When Secondary finishes the work block, results and status are loaded into the "Secondary Result Structure", executes a "barrier" operation insuring coherence, and jumps back into the "wait for interrupt" loop. Additionally, the Secondary may interrupt the primary by reading the Ethernet Address ROM in Grand Central space. This read posts an interrupt to the Primary (see Table 20 on page 36).
7. When appropriate, the Primary processor checks the "Secondary Result Structure" for indication the secondary has completed the work block (successfully, or unsuccessfully).

A.3.2 Secondary Processor Initialization

The following illustrates the mechanism by which the primary and secondary processor start-up and recognize which is which.

1. Following reset, both processors start at the PowerPC architectural reset vector in ROM (0xFF010000). Very early on, they both read the HH "LastGrant" register and therefore determine which is the primary and secondary.
2. The Primary processor continues on with system initialization.
3. The Secondary processor performs some rudimentary initialization and jumps in the "wait for interrupt" loop.

Note that in the above examples, more interesting software issues have been glossed over, such as:

- Virtual memory and page table synchronization

A.3.3 PowerMacintosh 8500/7500 MP limitations

The following are the known limitations of the PowerMacintosh 8500/7500 asymmetric MP implementation:

- Secondary does not allocate in the L2. This is a Secondary processor performance issue only. All caches in the system are still coherent.
- Secondary cannot field interrupts from I/O etc.
- Secondary and Primary page tables must be synchronized "by hand" if 603.

Appendix B MacRISC

This Appendix includes relevant MacRISC documents taken verbatum from Ron Hochsprung.

B.1 MacRISC and Endians

Introduction

This note concerns what MacRISC designs should do to support **Little-Endian** modes, even though the "native" mode of PowerPC is **Big-Endian**[1]. Such support might be done to enable ports of operating systems that, for some reason or other (stupidity?!), require that the underlying hardware "appear to be" Little-Endian (e.g., NT).

Note that even if we do not provide any support for Little-Endian, we will still be dealing with Endian issues since PCI (our local bus and expansion bus) is Little-Endian. Before going into the more "interesting" aspects of Little-Endian support, we will dwell on the basics of Endian-ness, and in particular, how it affects PCI implementations.

Another document, ***Endian-Aware Software Support***, describes several alternative mechanisms to allow "Endian-independent" software to be produced. Software for V0, where PCI drivers, etc., especially those written in C code for the 68K (emulated) should follow the guidelines suggested in that document.

Endian Basics

To give a concrete example around which to discuss the issues, consider coding for a system that contains a DBDMA[2] controller.

DBDMA defines a descriptor format whose C definition is could be defined as:

```
struct {
      byte  C;          // "command" byte
      byte  F;          // "flags"
```

---

1. The terms **Big-Endian** and **Little-Endian** come from Jonathon Switf's *Gulliver's Travels*. The subjects of the empire of *Blefuscu* were divided into two camps: those who ate their eggs starting with the "big" end and those which started with the "little" end.

2. As defined in *Descriptor Based DMA Architecture - A DMA Architecture Optimized for 32-bit Motherboard Designs.*

```
half  L;          // "length" (count)
word  A;          // "address"
dword X;          // "field64"
}     DBDMA_descriptor;
```

Note: the above assumes that byte, half, word, dword have been typedef'd appropriately to 8,16,32 and 64 bit scalars, respectively.

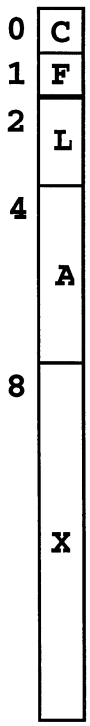
A compiler would assign offsets to the fields of the descriptor as follows:

```
C     0
F     1
L     2
A     4
X     8
```

For further discussion, suppose that the following code was used to initialize a descriptor:

```
DBDMA_Descr  aDescr;
aDescr.C = 0x10;
aDescr.F = 0x11;
aDescr.L = 0x2223;
aDescr.A = 0x44454647;
aDescr.X = 0x88898A8B8C8D8E8F;
```

Consider the following diagram, which presents the layout of the descriptor in an "Endian-neutral" format:

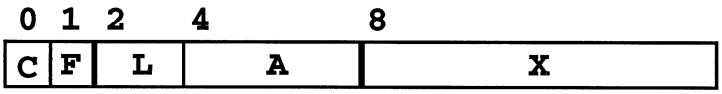


Notice, in particular, how the offsets are associated with the "beginning" of each field. As we shall see, the primary difference between Big- and Little-Endian has to do with what is considered to be the

"beginning" of a field.

Big-Endian

Suppose one rotates the diagram counter-clockwise:



This diagram shows how a Big-Endian processor and/or memory system would organize the structure. In a Big-Endian system, memory is organized with the address of each byte increasing from most significant to least significant.

> *Note: In all "horizontal" diagrams in this document, the organization of memory is assumed to have the more significant bytes to the left and the less significant bytes to the right. This is consistent with computer systems of interest to this discussion, and is the order with which most people are already familiar. Likewise, all bit(byte)-field designations are done by referencing the most-significant bit(byte) number of the field first.*

For single-byte values, there are no particular issues. However, for multi-byte quantities, a question immediate springs to mind: in what order should we interpret the bytes of a value?

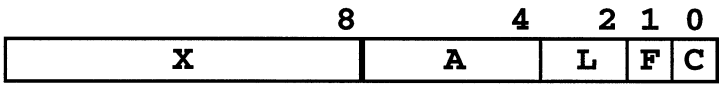
As noted above, we will assume that all multi-byte fields are interpreted with more significance to the left and less significance to the right. This means that the address of the most significant byte of the address field A is 4, while byte 7 corresponds to the least significant byte of A.

Bit numbering in a strictly Big-Endian architecture should naturally follow the ordering of the bytes; i.e., the most-significant bit should be bit-0. This is true of PowerPC, but not true of the 68K (except for the bit-field instructions)! In all bit-field designations, we will follow strict Big-Endian rules; thus, the first bit designated (the most-significant) will have the lowest bit-number.

> ***Note: The 68K family and PowerPC are naturally Big-Endian systems.***

Little-Endian

Suppose we take the original diagram and rotate it clockwise:



This diagram shows how a Little-Endian system would organize the descriptor. Notice to which bytes the "beginning" of each field refers. Instead of referring to the most-significant byte of a field, the offsets refer to the least-significant byte of each field.

Hence, in this example, byte-4 refers to the least significant byte of the A field, while byte-7 refers to the most significant byte.

Bit numbering within a Little-Endian architecture naturally follows that of the byte ordering. I.e., bit-0 represents the least-significant bit of a field. Thus, in bit-field designations, the first bit denoted (the most-significant) will have the highest bit-number.

> ***Note: The x86 family and VAXes are examples of naturally Little-Endian systems.***

Scalar Accesses

If all accesses to a data structure were done with reads and writes of a field's size, a program could not determine on what system it was executing. I.e., a Word access to A would always get the correct value.

The values chosen above are encoded; the first nybble gives the size of the element, while the second represents the byte offset of each byte, assuming Big-Endian ordering. All access to aDescr.L would be identical on either a Big-Endian or Little-Endian system. I.e., we can not tell by using "normal" code what the Endian-ness of our system is.

Trick Accesses

However, what if we defined a "trick" structure that let us look "inside" of a scalar. For example, consider:

```
union {
      half  H;
      byte  B[2];
      }     halfTrick;
halfTrick ht;
ht.H = aDescr.L;
if( ht.B[0] == 0x22 )
      printf( "I'm on a Big-Endian");
else
      printf( "I'm on a Little-Endian" );
```

This "trick" code can determine what the Endian-ness of my system is by detecting the ordering of significance relative to address[3].

Mixed-Endian

If all systems were entirely either Big-Endian or Little-Endian, life would be so much easier! Unfortunately, we are living in a world that is Mixed-Endian; some data we access at least crosses busses which are Little-Endian.

Herein lies the rub[4]!

It is very important to note that only data is involved with Mixed-Endian nonsense. Addresses are simply passed on directly across bridges[5], at least so far as byte-swapping is concerned. There may be transformations required when transporting an address across a bridge, usually involved with encoding

---

3. The Apple Pascal system used on Apple IIs, which was based upon UCSD Pascal, used such a trick. The "object code" (called, P-Code) was a byte-code for a "virtual" machine. However, there were many places where 2-byte integers were contained within the P-Code. The interpreter for P-Code had to know in what order to read the bytes to get the proper interpretation; hence, the trick. Since this was way before the era of Political Correctness, they chose to call the Endian-ness of a system its **sex**!; the 68K is considered male, while the 6502 was female.

4. I happened to look up the definition of **rub** in my *Webster's New Collegiate Dictionary*. The definition I had assumed was there ("1.b OBSTRUCTION,DIFFICULTY."), but there was an even more appropriate one: "1.d: something that mars serenity".

5. A bridge is any entity which interfaces two different busses. In this context, the NuBus interfaces of the Mac II family are bridges.

byte-lanes, transfer size, etc. Other that this, however, an address is never swapped!

Note: bridges which support Little-Endian mode must "swizzle and/or un-swizzle" the addresses; see discussion of PowerPC Little-Endian Mode below.

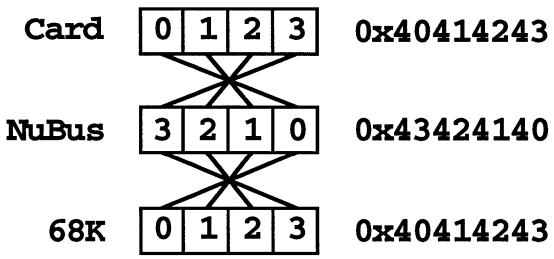
Example: NuBus

NuBus is defined as a Little-Endian bus. At the time of the design of the Mac II, we were faced with a decision about how to interface to NuBus from the 68K world. What we decided to do was follow a regime called "Address Invariance" (also referred to as "byte address consistency")[6].

Address Invariance is a simple rule that says that bytes get mapped across a bridge according to its address (or, byte-lane-number); i.e., the address of a byte is kept the same on both sides of the bridge. On the Mac II, byte-lane-0 of the 68K bus gets mapped to byte-lane-0 of NuBus, etc.

If one views a 68K 32-bit word going to NuBus, following this rule, it looks as if the bytes are swapped (in significance). Indeed, they are. However, since in the NuBus world we are living in a rather closed environment, the card designers undo this byte-swap on their side of NuBus. Thus, if one looks at memory on a card it looks exactly the same way it would on the Mac side.

The following diagram shows how data gets mapped from the Mac, across NuBus and, onto a card:



Note: Byte-swapping is like parity. Any even number of byte-swaps produces the original ordering.

Two Problems with Mixed-Endian

The support of Mixed-Endian systems must solve two different problems, in various combinations.

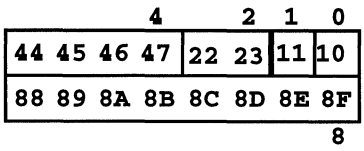
Data "Swapping"

Data swapping occurs when data in one Endian format is read in by the "other" Endian system. For example, suppose we take the DBDMA descriptor in our example from above which was generated by a Little-Endian system and saved to disk. The data is then read back from the disk on a Big-Endian system.

Assume that the data is written to disk in byte-address order. Thus, the Little-Endian data whose memory image looks like:

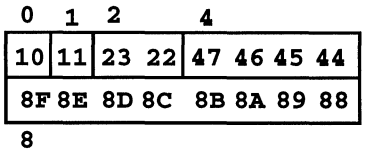
---

6. p. 4-7, *Designing Cards and Drivers for Macintosh II and Macintosh SE.*



Note: To make the diagrams correspond more to real memory systems, we have changed them to assume that memory is configured in an 8-byte wide configuration.

When read by the Big-Endian, the data would end up in memory (assuming, again, the read is done in byte-address order) as:

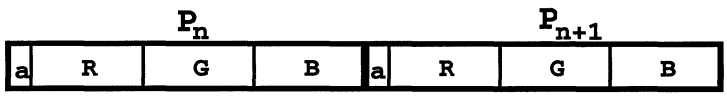


Notice that the byte-offsets of each field are still correct. However, the data within each field has been "swapped". I.e., if **aDescr.A** was read, using a normal Word load, the data in the register would be: **0x47464544**. But, the original data was written as **0x44454647**!

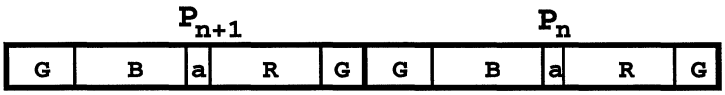
Data Swapping and Frame Buffers

Another example of data swapping is what happens to multi-byte pixels in a frame buffer. Mac software defines several multi-byte pixel formats. For this discussion, we will look at 16-bit pixels, since they are the most interesting with regard to the effects of byte swapping.

The Mac 16-bit format interprets a half-word as consisting of a 1-bit alpha followed by three 5-bit Red, Green and Blue color components. The following diagram shows how these pixels are packed into a word in Big-Endian memory.



When this data is moved across PCI, the data swapping will cause the data to appear as:



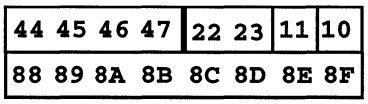
Two things should be noted about what happens during the "swap". First, the relative location of the pixels is correct for the Little-Endian PCI; this is a direct consequence of the Address Invariance being applied. Second, the data within the pixels has been "munged". In particular, notice how the Green component has been split into two pieces; this is because it spans a byte boundary.

Address "Swizzling"

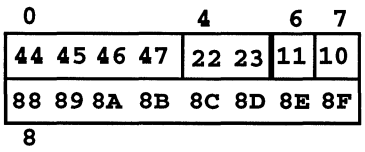
Suppose we wanted to make it "appear that" memory is maintained in Little-Endian format, even though we are running on a processor that is inherently Big-Endian (e.g., PowerPC). This would be the case, for example, if a port of NT were to run on the hardware; NT requires its world to "appear to be" Little-Endian.

> ***Note: For the moment, don't worry how we get data into memory in this format; that's part of what this document is supposed to describe.***

I.e., we want the **aDescr** data in memory to look like:



However, if we place Big-Endian byte-lane numbers on the diagram, we have:



If one looks at the data within the fields, the byte ordering is correct. But, now the addressing is wrong! The Addresses are "swizzle"; e.g., what should be on byte-lane 0 (**aDescr.C**) is really on byte-lane 7.

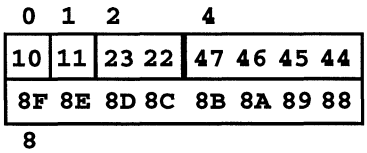
PowerPC Little-Endian Support

The PowerPC Architecture defines two separate mechanisms for dealing with Little-Endian (and, Mixed-Endian) systems: Byte-Reverse Loads/Stores and Little-Endian Mode.

Byte-Reverse Loads and Stores

Within the PowerPC Architecture, there exists a class of Loads and Stores that perform a "byte-swapping" based upon the size of the data transferred; e.g., a Load Word Byte-Reverse Indexed (**lwbrx**) swaps a 4-byte value. The primary purpose of these instructions is to allow efficient access to data that is stored in the "other" Endian-ness.

Using the example given above, where we have read the descriptor in "swapped" order from the Little-Endian file, we have:



If we use the **lwbrx** to access **aDescr.A**, we will get **0x44454647**, which is the correct data in the correct format!

The Byte-Reverse Loads and Stores address the problem of Data Swapping.

Caveat: The use of Byte-Reverse Loads and Stores is somewhat more expensive than normal ones. This is because they only exist in the Indexed form, and there are no Update forms. Hence, addresses of fields within data structures must be either explicitly calculated and/or their offsets loaded into a register. Also, there is currently no software support (e.g., in the C compiler) to generate these instructions. **(For suggestions on how these instructions could be supported see Endian-aware Software Support).**

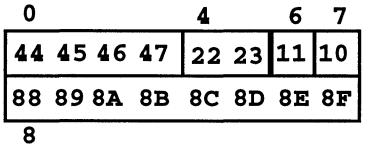
Little-Endian Mode

The PowerPC Architecture also includes a "Little-Endian Mode". The effect of this mode is that addresses are "swizzled" when they are used to access memory. The swizzle uses an exclusive-or of the low-order 3 bits of an address with a constant which depends upon the size of the Load or Store. Word Loads and Stores use a value of 0b100, Halves use 0b110 and Bytes use 0b111. The resulting value is used to make memory references to the Big-Endian memory system.

Note: The Effective Address itself is not modified; only its interpretation as used to access memory! For example, the "update" forms of Loads and Stores will alter the base register with the same value, regardless of the current Endian Mode. I.e., the "swizzle" is totally transparent to software.

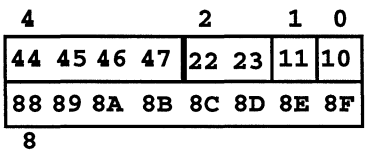
Notice that the swizzle involves only the low-order 3 bits. The size of the swizzle depends upon the maximum scalar data type that can be accessed by the system; it does not depend upon the width of the processor's data path. In the case of PowerPC, the longest scalar is a double-word; hence, the swizzle value is 3 bits.

In Big-Endian memory we have (using the example described in Address Swapping):



Now, if we run the processor in Little-Endian Mode and access the data using the standard offsets, a reference to **aDescr.A** will get **0x44454647**, which is correct! (We get to the correct data by XORing the offset (4 == 0b100) with 0b100, thus giving us 0b000 --> 0.)

The effect of Little-Endian Mode on the above example is that the data fields "appear to be" at the following offsets:



The PowerPC Little-Endian Mode addresses the problem of Address Swapping.

Caveat: Little-Endian Mode does not support mis-aligned data accesses. Access to

mis-aligned data must be done by code sequences and/or subroutines. As with the Byte Reversed Loads and Stores, there is currently no compiler support for the mis-aligned cases.

PCI and Endian-ness

PCI (Peripheral Component Interconnect) is being designed into the MacRISC Architecture as a standard for local interconnect and system expansion. For example, the Grand Central chip, which implements DBDMA for standard Mac peripherals, will be interfaced to the system via PCI.

The connection of the native RISC-chip's bus (ARBus) to PCI will require a bridge function. While it is quite probable that the bridging services will be contained within an ASIC that includes other functions within the system, for the purposes of this document, we will examine only the requisite bridge functions.

ARBus (Apple RISC Bus) Basics

ARBus is the name given to the subset of the 60x's bus which MacRISC implementations use. The ARBus consists of a non-multiplexed 32-bit Address bus, 64-bit (Big-Endian) Data bus and several control fields.

There are two types of data transactions supported by the ARBus:

• Single-beat transactions, from 1-8 Bytes (where the data is contained within a single 64-bit data "beat"). This is encoded by specifying the starting-address (the most significant byte of the transaction) as the Address and its length by the **TSIZ** lines. These accesses are typically done when the memory area to which they are made is marked Cache Inhibited.

• 32-Byte Cache-Sector Bursts, transferred in 4 64-bit data beats. Burst Reads start with the double-word containing the "critical word" first, with subsequent data "wrapping around"; Burst Writes are always Cache-Sector aligned.

For purposes of discussion later, the byte-lanes of ARBus will be called D0,...,D7, where D0 refers to the most-significant byte-lane of the processor (i.e., the one containing bits **D**[0:7]).

PCI Basics

PCI consists of a multiplexed Address/Data (**AD**) bus; the basic PCI incorporates a 32 bit wide path (**AD**[31:0]), while a 64-bit extension uses a 64 bit wide path. Since our systems will be using only the 32-bit version, we will not discuss the 64-bit extensions.

Devices contain registers which the device itself uses for determining if it is the target of a transaction; these registers are setup by the system during startup. Three address spaces are provided in the standard: Memory, I/O and Configuration.

The Memory Address Space supports a full 32-bit address space, where a 30-bit Address (i.e., to 32-bit Word resolution) is transferred during the Address Phase and 4 (8) Byte-Enable Lines (**BE**[3:0]/ **BE**[7:0]) indicate which byte lanes contain valid data during Data Phases. (The low-order 2 bits of the **AD** bus contain sequencing options during the Address Phase.)

I/O Address Space is (effectively) limited to the first 64K of possible addresses. A 16-bit address (down to byte resolution) is transferred during the Address Phase; the upper 16 bits of **AD** are forced to 0's for an I/O Cycle. This space is intended to support older models of I/O transfers, which can be

generated by x86 I/O instructions.

Configuration Space is designed to support system self-configuration. Each device must contain a 64 Word "Configuration Register" set, the first 16 of which have fixed functions. Included in this area are "base address registers", which designate addresses to which the device will respond during Memory and/or I/O transactions. Configuration Space accesses must generate a device-specific select line (**IDSEL**) to indicate which device is being accessed. During system initialization, software is required to configure the PCI devices so that they will respond appropriately under normal operation.

For purposes of discussion, the PCI's byte-lanes will be referred to as P3,...,P0, where P3 refers to the most-significant byte-lane (i.e., the one containing bits **AD**[31:24]).

In this document, we only consider the "gross" effects of Endian-ness of PCI. Details about how transactions are encoded, how transactions get mapped across an ARBus-PCI bridge, etc., should refer to ***MacRISC and PCI***.

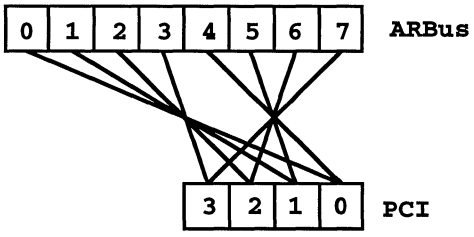
Basics of PCI-ARBus bridging

In this section, we describe the basic requirements of a ARBus-to-PCI Bridge with respect to Endianness; i.e., dealing with the fact that PowerPC is Big-Endian and PCI is Little-Endian. We do not discuss the additional support required for **Little-Endian Mode**; a later section describes these extensions.

Following the rule of Address Invariance, the PCI-bridge should perform a "byte-swap" across the bridge between ARBus and PCI for Memory transactions. This swap is done across a double-word (8 byte) sized unit. Thus, an ARBus access to byte-0 will be mapped to a PCI byte-0 access, transferring the data across PCI byte-lane-0.

> ***Note: In the diagrams to follow, two different representations are given. One shows "wiring" between the byte-lanes of the PowerPC ARBus Data bus and PCI's AD byte-lanes. The other representation shows the mapping in terms of the byte-lane designations.***

Thus, a diagram of the data path in the bridge would look like:




```
D0,D4 - P0
D1,D5 - P1
D2,D6 - P2
D3,D7 - P3
```

> ***Note: In all the ARBus-PCI diagrams, we show how byte-lanes get connected between the 8 ARBus byte-lanes and the 4 (32-bit) PCI byte-lanes. The bridge may also have to re-order multi-word accesses to or from PCI depending upon what mode the bridge is in; see PCI and PowerPC Little-Endian Mode, below.***

PCI and PowerPC Little-Endian Mode

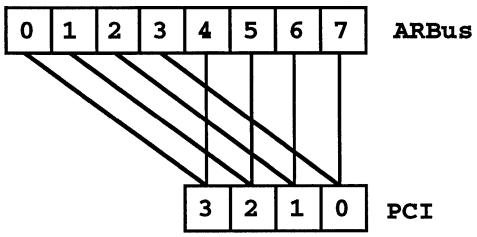
As mentioned above, PowerPC includes a Little-Endian Mode switch which enables an "address-swizzle". This Little-Endian Mode only works when the main memory is consistently maintained with the byte ordering of Little-Endian. This maintenance of Little-Endian memory could be done in software, but the most obvious support would be done by altering the default action of the bridge.

ARBus-to-PCI Bridge in Little-Endian Mode

The PCI bridge support is trivial; simply disable (by a bridge-specific Configuration flag) the default byte-swapping for Memory accesses and let data flow "straight through" from PCI. Note that all processor accesses will be swizzled when the processor is in Little-Endian Mode; these addresses must be un-swizzled when passed on to PCI, to access the appropriate data. By transparently mapping PCI accesses, the illusion of Little-Endian memory is provided.

The "un-swizzling" of addresses to (and, the swizzling of addresses from) PCI involves two mechanisms. The first is simply performing the XOR. The second is changing the order in which 32-bit data is multiplexed between the 64-bit ARBus **D**-bus and PCI's 32-bit **AD**-bus. This is because inverting address bit **A**[29] (**AD**[2]), which is involved for swizzles of all data types, implies which word of a double-word is transferred first.

The following diagram shows the net effect of having both the processor and ARBus-PCI bridge running in Little-Endian Mode:



D0,D4 - P3
D1,D5 - P2
D2,D6 - P1
D3,D7 - P0

Note: The byte lanes shown in the diagram represent the processors physical byte-lane numbering, which doesn't change. However, the processor's view of these lanes, because of the Little-Endian Mode, would appear reversed; i.e., the ordering would "appear to be" 7..0, not 0..7.

PCI-to-ARBus accesses in Little-Endian Mode

Transactions destined for the ARBus side of the world from PCI, when the bridge is running in Little-Endian Mode, must have the same address swizzling (as described above) performed by the bridge.

DBDMA and Endian-Ness

Apple chips should be designed with two somewhat conflicting goals:

- PCI Compliance. PCI Compliance means that an "Apple" PCI chip should be able to pass whatever "validation" suites are available for PCI. In particular, they must appear to be Little-Endian at RESET time.
- Ease of use in Apple systems. After being initialized by Apple startup code, the chips should "do the right thing". In particular, all accesses, data structures, etc., should work with the processor and bridges in either Big-Endian or Little-Endian Mode.

This requires an Endian-Mode switch, similar to that within the ARBus-PCI bridges. The reason is fairly simple. Remember the comment above about Endian-ness being like parity. In Big-Endian Mode, the bridge will perform a byte-swap. This byte-swap must be compensated for by a byte-swap within the chips. Likewise, when the system is running in Little-Endian mode, no swap will be done by the bridge; thus, no swap should be done by the chips. Hence, the swap/no-swap should be selectable with-

in our chips, both bridges and others.

Note that this swapping has to be done on field-sized units. In particular, for frame buffers, the "swap" depends upon the bit-depth in which the frame buffer is being accessed. A 1-byte grey-scale or indexed-color does not require any swapping!; address invariance will naturally place the data in the appropriate Little-Endian byte lane. However, 16-bit (1-5-5-5 format) and 32-bit (24-bit) true-color have to be swapped on a pixel-by-pixel basis. Remember that the location of the pixels are correct, as guaranteed by address invariance; only the data within a pixel has been swapped.

> ***Note: The above comments are particularly true for devices where existing software can not reasonably be changed; e.g., Frame Buffers. However, devices and/or cards for which new software must be written, anyway, can easily adopt the "Endian-aware" style in the first place. This would simplify the hardware and is how "generic" PCI cards would have to be supported, anyway.***

Software Support of Endian-ness

Dealing with Endian issues in software is complicated by the fact that compilers have not provided the necessary support. This section discusses what minimal compiler support would greatly simplify the process and describes how software folke can deal with Endian-ness explicitly, in place of such support.

First of all, one has to be aware that there is a problem. Unfortunately, many programmers are unaware of the problem. When one writes a program that is entirely self-contained, it doesn't make much difference. Except for the "tricks" noted above, a typical C programmer can happily ignore Endian-ness (or, not even have heard of it!).

When, however, a programmer must deal with data whose source and/or destination could be of a different Endian-ness, said programmer had better be very careful!

Please refer to the ***Endian-Aware Software Support*** document for further details and comments.

B.2 MacRISC and PCI

Introduction

PCI (Peripheral Component Interconnect) is being designed into the MacRISC Architecture as a standard for local interconnect and system expansion. For example, the Grand Central chip, which implements DBDMA for standard Mac peripherals, will be interfaced to the system via PCI.

The connection of the native RISC-chip's bus (ARBus) to PCI will require a "bridge" function. In some implementations, the PCI bridge function will be contained within an ASIC which includes other functions; for the purposes of this document, we will examine only the requisite PCI bridge functions of such ASICs.

Another document (***MacRISC and Little-Endian***) describes the general problem of "Mixed Endian-ness", with which we will be faced because of the possibility of Little-Endian chips and/or cards which need to be supported on PCI.

ARBus (Apple RISC Bus) Basics

ARBus is the name given to the subset of the 60x's bus which MacRISC implementations use; ARBus uses only Basic Protocol of the 60x. The ARBus consists of a non-multiplexed 32-bit Address bus

(**A**[0:31]), 64-bit (Big-Endian) Data bus (**D**[0:63]) and several control fields (e.g., **TBST***, **TSIZ**[0:2], **TT**[0:3]).

There are basically two types of data transactions supported by the ARBus:

- Single-beat transactions, from 1-8 Bytes (where the data is contained within a single 64-bit double-word). This is encoded by specifying the starting-address (the most significant byte of the transaction) as the Address (**A**[0:31]) and its length via the **TSIZ** lines, along with **TBST*** unasserted..
- 32-Byte Cache-Sector Bursts. These are cycles which are made when **TBST*** is asserted. Burst Reads start with the double-word containing the "critical word" first, with subsequent data "wrapping around"; Burst Writes are always Cache-Sector aligned.

ARBus provides explicit information at the beginning of a transaction as exactly how many bytes are being transported; it allows only a contiguous range of bytes to be moved (1-8 for Single-Beat or 32 for Bursts).

ARBus provides a single 32-bit (Memory) address space.

PCI Basics

PCI uses a multiplexed Address/Data (**AD**)bus; the basic PCI incorporates a 32 bit path, while a 64-bit extension uses a 64 bit wide path. For this section, we consider only the 32-bit version, which will be used in our initial MacRISC implementations.

The other PCI signals relevant to this paper are the Command/Byte Enable lines (**C/BE#[3:0]**). These lines encode the transaction type (e.g., Memory Read, I/O Write) during the Address Phase and indicate which byte-lanes contain valid Data during Data Phases. In all cases, these lines are driven by the transaction's initiator. Note that PCI allows non-contiguous transfers, which occur when a bus initiator does not drive contiguous **BE**s; compliant PCI targets must support these.

The low-order 2 bits of the Address bus are not used for basic memory transactions. Instead, they encode options on implicit address sequencing for "burst" sequences. (Unfortunately, none of the options include support for ARBus's "wrap around" cache protocol.) The combination of the upper 30 bits of the **AD** lines (presented during the Address Phase) and the **C/BE#** lines (available during the Data Phases) provide the necessary address and size information for a transfer.

Unlike ARBus, PCI has no indication presented during its Address Phase which indicates the intended length of a transfer; instead, the end of a transfer is indicated. This introduces several "corner cases" in the implementation of any Apple chips, especially a "bridge" chip.

Another unusual feature of PCI is that any target device can either "retry" a request (usually done before any Data is transferred) or, interrupt a Data transfer by generating a "disconnect" signal. The intention is that the current bus master will retry and/or continue the operation where it left off after relinquishing the bus for other potential masters. Again, this introduces more "corner cases" with which to be dealt; compliant PCI initiators must support these.

PCI provides three different "address spaces" (a 32-bit Memory space, 16-bit I/O space, 13-bit Configuration space), plus an address-less (or, "broadcast") signaling mechanism (Special Cycles). Furthermore, there is no direct address allocation provided by PCI; a "configuration" process is necessary to configure each device with the addresses to which the system will expect it to respond.

In MacRISC, a special mechanism (encoding the PCI access types via address bits) is required to be able to generate the appropriate transaction types on PCI from ARBus.

Bi-Endian-ness

In order for MacRISC systems to support multiple operating systems (e.g., MacOS, AIX, NT, OS/2), it is necessary that they be Bi-Endian. I.e., they must be able to run correctly with the processor running in either Big-Endian (e.g., for the MacOS and AIX) and Little-Endian (e.g., NT and OS/2) modes. As described below, this requires special hardware support in the ARBus-PCI bridge.

> ***Note: for details about how PowerPC handles Endian issues, see the PowerPC Architecture books.***

MacRISC Support of PCI Features

This section discusses how MacRISC deals with the non-data-transfer aspects of PCI.

64-bit Bus Extension Pins

Since the initial implementation of PCI on MacRISC will be using the standard 32-bit version, we only need to implement the 32-bit version of the connector. This is certainly true on single-slot systems, since there is absolutely nothing to be gained to a PCI card by adding the connector.

However, in multi-slot expansion systems, we should install the 64-bit version of the connector. This enables card-to-card transfers to use the extra bandwidth available. The only additional "logic" required are some pull-ups for the **REQ64#**, **ACK64#** and **AD[63:32]** lines.

Cache Support

MacRISC assumes that there is no coherency maintenance across a bridge. This means that the ARBus-PCI bridges do not need to generate **SDONE** or **SBO#**; the system should provide pull-ups.

Exclusive Access

The support discussed here allows completely general semaphore primitives to work within both system memory and PCI Memory space; within PCI Memory space, the semaphore locations must be marked Cache Inhibited.

Two separate hardware mechanisms exist:

- PCI "locked" sequences, using its **LOCK#** signal.
- **LWARX** and **STWCX.** in PowerPC.

Only support of the PCI **LOCK#** is provided in MacRISC; **LWARX** and **STWCX.** work only within the MacRISC coherency domain, which is essentially just processors and memory located on ARBus.

PCI LOCK# Support

PCI provides a mechanism for exclusive accesses from a master to a single target via the **LOCK#** protocol. This functions similar to the NuBus Resource Lock, except that locking of the entire bus is not required; other bus traffic is allowed to proceed, except that accesses to the locked target by other than the locking master are RETRY'd.

A simple way of implementing this may be to enforce a system-wide bus lock for the duration of a

locked sequence from PCI to any resource on the ARBus side, including other PCI's accessed via another ARBus-PCI bridge. This would ensure that the semantics of **LOCK#** are honored by accesses to any target in the system.

> ***Note: the ARBus requires a "LOCK" signal to indicate when it has logically been "locked" because of LOCK# forwarding and/or LWARX / STWCX. primitives from the processor.***

While this simple bus-lock would allow PCI accesses to semaphore locations controlled by **LWARX / STWCX.** primitives (within ARBus memory) to work, it does require that all arbitration (for ARBus and PCI(s)) be "centralized".

A more sophisticated, and higher performance, mechanism is to lock each segment only as needed. The response to an initial request with **LOCK#** from a PCI initiator has to be delayed until the entire path (segment by segment) from source to target has been locked. If any "lock" request through a bridge can not be performed (because of a Locked state on the "other" segment), the original request has to be "backed out" (i.e., RETRY'd) along each segment (in reverse order) back to the source.

Once the entire path's Locked state has been established, no further action is required until the "unlock", which simply propagates the "unlock" forward through the locked path, segment by segment. In effect, only the segments contained in the path from the initiator to its target gets "locked". This mechanism can be implemented in a "distributed" fashion, not requiring any "central" arbitration.

> ***Note: this mechanism works without deadlocks because both sides of a bridge (ARBus and/or PCI) support a RETRY mechanism. But!, we have to be <u>VERY</u> careful!***

LWARX, STWCX. to PCI Space (NOT!!!)

The PowerPC provides a simple, yet powerful, means of many types of semaphores to be implemented. This mechanism uses a "reservation" model with two instructions; one to set the reservation (**LWARX**), and one to conditionally store (**STWCX.**) based upon the state of the reservation. This reservation mechanism uses the cache coherency provided by PowerPC to guarantee exclusive access (defined by the semantics of **STWCX.**). See ***PowerPC User Instruction Set Architecture*** for details.

> ***Note: while the PowerPC mechanism is very general, not all processors can implement all of the types of semaphores which can be provided by PowerPC. The only "generic" semaphore which can be used across heterogeneous processors is a "Universal Lock/UnLock", using a set of "magic" values[7].***

Semaphores maintained in ARBus memory can be completely general, as long as the ARBus is "locked" using the rules described above. However, general semaphores within PCI Memory can not be supported[8]. <u>Hence, semaphores must be kept in "System Memory" (i.e., on the ARBus); no support for semaphores on PCI is provided. No support for **LWARX** and/or **STWCX.** to PCI is intended.</u>

---

7. The locked value is actually one of two values, either **0b00000000** or **0b1xxxxxxx**. The "magic" value is the one chosen for UnLocked, which is **0b01000000**. This value, when stored in the upper byte of a 32-bit word, where the low-order bytes are all 0's for the UnLocked case, has been shown to universal in the sense that all known processor "atomic" protocols have a sequence which can use this value. This value set comes from *Shared-Data Formats Optimized for Scalable Coherent Interface Processors*, IEEE P1596.5.

ARBus-PCI Bridging

Access Mapping

The primary function of a bridge is to pass transactions from one side to the other, performing any necessary buffering, byte-swapping, etc. To perform the minimum function, a bridge must be configured with the address to which it should map across.

The PCI model is that a "positive" decode is done against a range (or, ranges) of addresses from the "Standard Bus" (i.e., ARBus) to the PCI for Standard Bus accesses. It also performs an "anti-decode" for any PCI accesses.

A suggested way of doing this is for the bridge to support a number (say, 16) of register pairs, where one register is a "match" value and the second is a "mask" value. Every access (on both busses) is matched against all of these registers, in parallel. A summary OR of the individual matches determines whether the bridge passes an access to the other side. A **true** value of the OR for an access on the ARBus would indicate a transaction which must be passed to the PCI side; a **false** on the OR for a PCI access means that the transaction must be passed to the ARBus side.

16 such pairs would allow an average of 4 address spaces per chip/slot, for up to 4 slots and/or chips.; we need to determine if this is sufficient. Note that if a resolution down to 64 KB is sufficient, then both the match and mask values (16 bits each) could be packed into a single 32-bit register for each compare set, simplifying the software somewhat.

> ***Note: the MacRISC Address Map provides "fixed" addresses for up to 4 ARBus-PCI bridges.***

ARBus to PCI

Supporting ARBus accesses to PCI is relatively straightforward, since ARBus provides all necessary information (e.g., direction, size) during its Address cycle. Requests are simply mapped to the appropriate encodings during the Address phase and generate and/or interpret the **BE#** data during Data phases.

However, the bridge must still deal with retries and/or disconnects by continuing with the transfer until the entire transaction length has been transferred.

PCI non-Memory Cycles

The ARBus can only make "memory" cycles; the 60x chips do not have a natural way of generating the equivalent of x86 ***I/O Cycles***, which are directly supported by PCI. Since some devices (either, third-party chips or cards) may require access to x86-style I/O registers, we must provide a mechanism to make these cycles on the PCI side of a bridge.

Likewise, during system configuration, software must access the PCI Configuration Registers of devices, which are done by making ***Configuration Cycles*** on PCI. PCI Configuration Cycles are a "broad-

---

8. This is not quite true. The true statement is that they could be implemented with "cooperative" OS support. However, without depending on this support, we should assume that the mechanism described here is the only one which can be guaranteed to work.

cast" type of access. An "out of band" mechanism (i.e., driving the device's/slot's **IDSEL** line) must be provided.

PCI also defines a class of ***Special Cycles***, some of which directly support x86 equivalent cycles. Since some of these may be useful for us, due to PCI cards using them, they, also, must be able to be generated from the 60x.

In addition to these non-Memory cycles, a special range of "pass-through" Memory cycles are defined within MacRISC. This is so that Apple chips can be allocated their address space in a simpler fashion than the full-blown PCI allocation mechanism. Accesses to this Memory area generate the same type of Memory cycles indicated by the ARBus access as would normal PCI Memory accesses.

Pass-Through Memory Cycles

A Pass-Through Memory Cycle is made when a bridges address has **A**[7] == 0b1. In this case, the bridge passes the original ARBus address during the Address Phase. This effectively defines an 16 MB Pass-Through Memory region per ARBus-PCI bridge.

I/O Cycles

PCI I/O Cycles are made when a PCI bridge is accessed with an address with **A**[7:8] == 0b00. An access to this address will generate an I/O cycle on the PCI side of the bridge, forcing the upper 16 bits of the **AD** lines to 0 and passing the low-order 16 bits from the ARBus address. Software should guarantee that the upper bits (other than the bridge select bits) are zeros.

Note: I/O Cycles must pass all 16-bits onto the AD bus during the Address Phase. This is the only type of cycle which uses the original A[30:31] from ARBus.

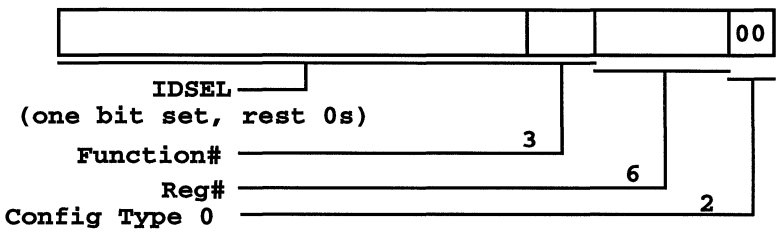
Configuration Cycles

Configuration Cycles are generated in an indirect manner, similar to the mechanism suggested in the PCI specification for x86 class processors. This mechanism uses two registers (Config Address and Config Data), accessed using normal ARBus Cache-Inhibited, single-beat Reads and Writes.

The first register is the Config Address register. This register contains the address to be used during the Address Phase of a subsequent PCI Config cycle. Its format depends upon the type of Config Cycle to be generated on the local PCI.

Type 0 Config Cycles

For Type 0 Config Cycles, where the device being accessed is on directly on the PCI bus side of the bridge, the format of the Config Address register is:

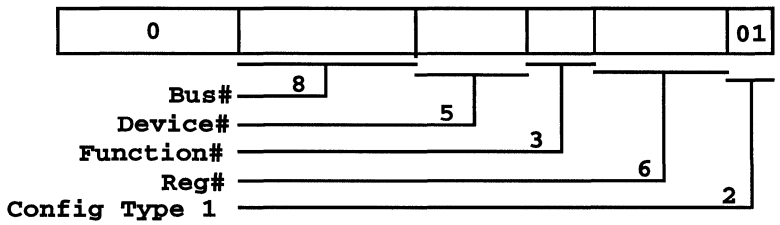


Note that the upper bits of the address must contain a single bit which represents the desired **IDSEL**.

Type 1 Config Cycles

Type 1 Config Cycles are used in hierarchical PCI configurations. In this case, the device to be accessed is not on the PCI bus side of the bridge, but, instead, is located across a PCI-PCI bridge which is on the local PCI bus.

The address for Type 1 cycles must be written to the Config Address register as:



Note that in either case, the contents of the Config Address register is driven to the AD bus without modification!

The second register involved with Config Cycles is the Config Data register. This is actually a "pseudo-register" in the sense that it does not physically exist within the bridge. An access to this "register" causes a Config (or, Special) Cycle to be made on the PCI side, using the address contained within the Config Address register.

The type of Config cycle (Config Read / Config Write) depends upon the cycle accessing the Config Data "register". The data for a PCI Config Write Cycle is specified by the data being written to the Config Data "register". Data read from the during a PCI Config Read Cycle is the value returned for a read of the Config Data "register".

The PCI method assumes that these registers are accessed by I/O accesses to hard-wired addresses. Since we can't generate I/O cycles to start with, MacRISC adopts a simplified decode. Namely, using A[7:10], as follows:

A[7:10]	Interpretation
00xx	- PCI I/O Cycle
0100	- Config Address Register
0110	- Config Data "register" -> PCI Config Cycle
0111	- Config Data "register" -> PCI Special Cycle

Special Cycles

Special Cycles are generated using a modification of the Config Cycles. As indicated above, Special Cycles are generated by ARBus writes to Config Data "register", using a different encoding on A[7:10]. Special Cycles are only valid for Writes.

This form of Special Cycle generation is used for devices on the directly connected PCI bus. To generate Special Cycles on PCI busses located beyond PCI-PCI bridges attached to the local PCI bus (or, beyond in hierarchy), a Config Cycle is generated using the above mechanism, but where the Config Address has the Device and Function fields encoded with ones (as described in the PCI specification). This Config Cycle is interpreted by subsequent PCI-PCI bridges as a clue to generate a Special Cycle on its (the bridge selected by the Bus# field) PCI bus.

ARBus-PCI Data-Swapping and Address-Swizzling

The PCI specification suggests that all Host-PCI bridges provide for Address Invariance. I.e., the byte lanes of the host map to the byte lanes of PCI according to their byte-lane number. This mapping is done relative to the processor's "view" of memory, not according to some arbitrary numbering of physical byte lanes. Thus, the actual operation required of a bridge depends upon the Endian Mode in which the bridge is operating.

The system memory which is directly attached to the ARBus does not have to do anything to support either Endian mode. (In fact, it has no way of knowing in what mode the processor is running.) However, the ARBus-PCI bridge needs to know, since it must perform the operations as described below, depending upon the processor's mode. Since it has no directly accessible information (e.g., a pin), the ARBus-PCI bridge must have a software accessible mode bit which tells the bridge in what mode to make accesses.

The PowerPC model for Endian-ness presumes that memory is mapped to "appear" as it should according to the Endian Mode on which the processor is running[9].

Big-Endian Mode

The "native" mode of PowerPC is Big-Endian. In this mode, the memory "appears" to have byte-lane 0 in the Most Significant byte of a double-word and byte-lane 7 on the Least Significant byte. In fact, the hardware bus interface directly maps this mode, even to the point of labeling the bus byte lanes.

When the Host-PCI bridge is running in Big-Endian Mode, no alteration of the ARBus address is necessary during the Address Phase. The address is simply placed directly onto the PCI side.

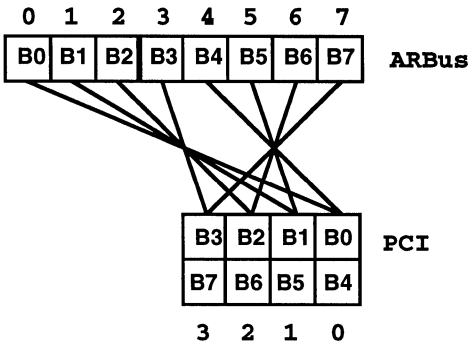
However, during the Data Phases, a byte-swapping is performed. I.e., **D**[0:7] (ARBus byte-lane 0) <-> **AD**[7:0] (PCI byte-lane 0), **D**[56:63] <-> **AD**[63:56] (for 64-bit PCI implementations) / **AD**[31:24] (for 32-bit versions).

When mapping between ARBus and PCI in Big-Endian Mode, the muxing of the words (required when going from 64-bit ARBus to 32-bit PCI) is done by placing the word on ARBus **D**[0:31] onto PCI first, followed by the word on **D**[32:63], thus preserving the correct address order.

The following diagram illustrates the byte-swapping and word ordering for Big-Endian Mode. The numbers above and/or below the boxes represent the byte-lane numbering of ARBus and PCI. The names within the boxes indicate how data gets swapped across the bridge. Note that the order of the words on PCI is Most Significant first, followed by Least Significant (i.e., Big-Endian order).

---

9. Note that the processor has two separate Endian Mode controls, one for Problem (User) and one for Privileged (System). When we refer to the "processor's Endian Mode", we mean the System mode. Special software support is necessary to deal with mixed-Endian tasks.



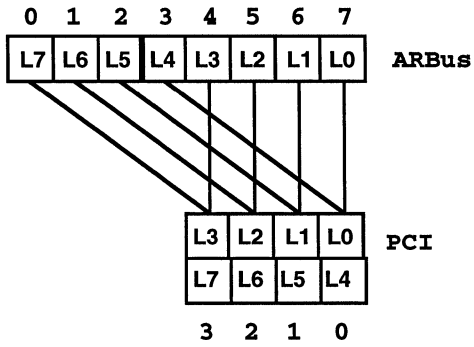
Little-Endian Mode

In Little-Endian mode, the processor makes the memory "appear" to be organized in Little-Endian order by "swizzling" the low-order address bits as accesses are made to the cache and/or memory. This re-numbering makes it look like byte 0 is in the Least Significant byte lane (i.e., physical byte lane 7) and byte 7 in the Most Significant byte lane (i.e., physical byte lane 0). Such swizzled address show up in their swizzled form on the ARBus addresses during a bus transaction.

For single-beat ARBus-to-PCI (PCI-to-ARBus) transactions, the address must "un-swizzled" (swizzled). The swizzle is performed by XORing the low-order 3 bits of the address with a constant which depends upon the data-size of the transaction. The following table shows the constant (in binary) with which to XOR for each of the basic transaction sizes. Note that when in Little-Endian Mode, the processor does not support mis-alignment; hence, the address of a given transfer size will be aligned to that size.

```
1-byte111
2-byte110
4-byte100
8-byte000
```

The following diagram shows how the byte-lanes match up between ARBus and PCI when the bridge is in Little-Endian Mode. Inside the boxes, the name indicates the Little-Endian byte-lane as it "appears" to software. Note that the order of the words on PCI is Least Significant first, followed by Most Significant (i.e., Little-Endian order).



As an example of address swizzling, a half-word access by PowerPC code running on the processor to

an effective address of 0x???????2 would appear as a 2-byte transaction on ARBus with an address of 0x???????4 (the address having been swizzled by XORing with 0x6); the data would be transferred on ARBus byte lanes 4-5 (**D**[32-47]), which matches the ARBus address. When transferred to PCI by the ARBus-PCI bridge, the address used on the PCI side has to be un-swizzled, yielding the original software address of 0x???????2, while the data is passed "straight through" to byte lanes 3-2.

Likewise, a 2-byte access on PCI at address 0x???????6 (appearing on byte lanes 3-2) would be made to ARBus address 0x???????0, with the data passed through to ARBus byte lanes 0-1.

In Summary:

When the bridge is in Big-Endian mode, it SWAPs the DATA, but leaves addresses unmodified.

When in Little-Endian mode, the bridge performs no data swapping, but SWIZZLES the ADDRESS.

PCI to ARBus

Mapping of PCI Transactions to ARBus

Most of the corner cases come up in PCI to ARBus transactions. All valid PCI "Memory" sequences must be mapped to the requisite number of ARBus transactions to implement the semantics. For example, non-contiguous **BE**s must generate a sequence of byte transactions to ARBus.

> ***Note: we do not provide any way of mapping I/O, Configuration or Special Cycles from PCI to ARBus.***

PCI provides more functionality than can be conveniently mapped to ARBus. The general rule for bridging should be to provide as direct a mapping as feasible, using the DisConnect mechanism to force "hard" cases to be continued. The semantics of DisConnect guarantees that the original transfer will occur (albeit, in a piecemeal fashion), just not quite as fast as the initiator may have expected.

PCI **Memory Read**s should map to Single-Beat ARBus reads, using the most direct mapping possible. I.e., a 1-byte read from PCI (single Data Phase, one **BE** set) should cause a 1-byte read on ARBus. Any request which can not map directly to one Single-Beat ARBus read is allowed to do a full 64-bit ARBus read. The bridge uses the **BE**s and **FRAME#** to decide what ARBus Single-Beat transaction should be done. Any PCI Memory Read which asks for more than what can be supplied by a Single-Beat ARBus read should use DisConnect to force subsequent data transactions from the initiator.

PCI **Memory Read Line**s should use ARBus Burst reads. Since the address used for the read may not be cache line aligned, the bridge should ask for the "critical word" first.

If an implementation can support the extra buffering, **Memory Read Multiple**s could be mapped into sequences of ARBus Burst reads, thus, implementing the intention of the PCI specification. However, this seems a little over-zealous, so the normal implementation would simply do a single cache sector read on ARBus and use DisConnect to subsequent Data Phases beyond the first cache sector boundary.

On PCI, **Memory Write**s require that the bridge acquire the data from the PCI device, analyze **BE**s, etc. before initiating requests on the ARBus side. To minimize buffering, etc., the bridge may issue a DisConnect to the initiator while performing the ARBus transactions.

> ***Note that a PCI request may need to be broken into several ARBus transactions, depending upon whether the BEs can be mapped into ARBus transactions which re-***

quire contiguous transfers.

For PCI **Memory Write and Invalidate**, we need only support the sequential addressing mode, with the first address aligned to a cache sector boundary (32 bytes). Other cases use DisConnect to enforce Single-Beat sequences on ARBus.

Data-Swapping and Address-Swizzling

See the discussion above for how ARBus-PCI bridges handle the basic mappings between PCI and ARBus for Addresses and Data.

Big-Endian Mode

When the bridge is running in Big-Endian mode, the data is byte-swapped as indicated in the ARBus to PCI section, above. The address used for the ARBus transaction is that of the Least-Significant byte lane being transferred from PCI (which becomes the Most-Significant due to byte-swapping).

Little-Endian Mode

The general case of a PCI transaction requires a somewhat more complex swizzling when running in Little-Endian mode. This is because the length and alignment of transaction originating on PCI does not necessarily follow the restrictions dictated by the PowerPC Architecture.

As described in the ARBus to PCI section, no byte-swapping is performed on the data. However, the address needs to be swizzled.

The rule for mapping from PCI to ARBus in Little-Endian mode is that the address used on ARBus is the swizzled version of the Most-Significant byte of the PCI data using a value of 0x7. For example, a 3-byte transfer in byte lanes 2-0 on PCI would be made to an address of (ARBus) byte lane 5 (with the data passed "straight through").

B.3 PCI Apertures

3.3. Dealing with Different Pixel Representations

This document describes a mechanism to enable "arbitrary" transformations on pixel-data being transported across PCI between cooperating devices. The mechanism relies upon the concept of "apertures" to support multiple, simultaneous views of the a shared pixel "buffer".

A simple example for which apertures could be used is to support a processor (which is rendering graphics data) and a motion-video source (which is sourcing digitized video) accessing the same frame buffer. A problem exists if the pixel formats of these two sources are not identical. For example, if the processor is accessing pixels in 1-5-5-5 RGB format and the video source is supplying 4-2-2 YUV data, how can simultaneous access be afforded?

One strategy might be to have a mode switch within the frame buffer's controller which controls in which pixel format accesses are expected. However, with this simplistic approach, software would have to ensure that the video was not sourcing when the processor was accessing the frame buffer and vice versa. While possible, this would introduce significant complication into a multi-media system.

3.3.1. Apertures

An alternative strategy is to define multiple "apertures" (or, views) into the single instance of a pixel buffer. An **aperture** is a "logical" view of the pixel buffer which can be different (e.g., in Color space, Endian-ness, etc.) than other apertures and/or the actual "physical" storage representation. The advantage of the aperture mechanism is that it can provide simultaneous access to a single, shared physical pixel buffer via an arbitrary number of logical views (the apertures), each of which can represent different pixel formats, view sizes, etc.

Two concrete examples follow which illustrate the utility of the aperture mechanism.

3.3.2. Color Space Conversion

This example is a solution to the problem introduced in Section 3.3, where we have two sources desiring simultaneous access to a single pixel buffer.

Figure 9 shows two apertures, one for the processor and a second for the video source; both are mapped to a common physical pixel buffer.

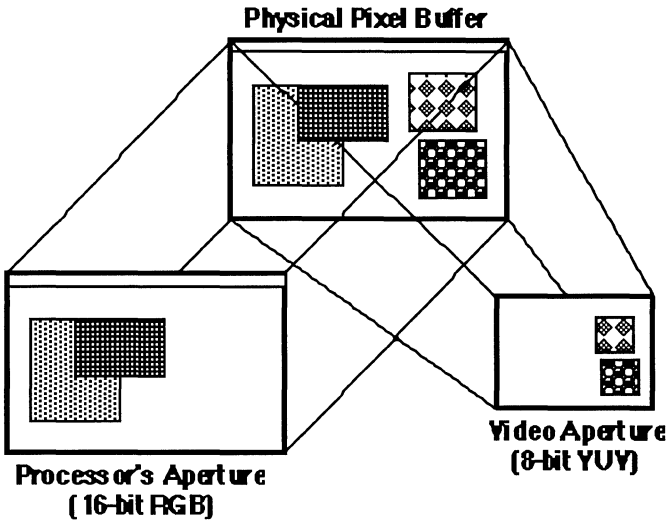


Figure 9. Aperture Example

In this figure, the aperture sizes (i.e., the number of bytes consumed within the address space) appear different even though the number of pixels (e.g., 1024x768) are the same. This is because the number of bytes required for a view is dependent upon the byte-size of a pixel within that view.

In this example, both the processor and the video source can access the frame buffer simultaneously. If we assume that the frame buffer is physically storing pixels in the RGB format assumed by the processor, then no transformation is required for the processor's accesses. However, the video source's data would have to be converted into RGB from YUV. The frame buffer controller "knows" that it has to do this conversion by observing into which aperture the video is writing pixels.

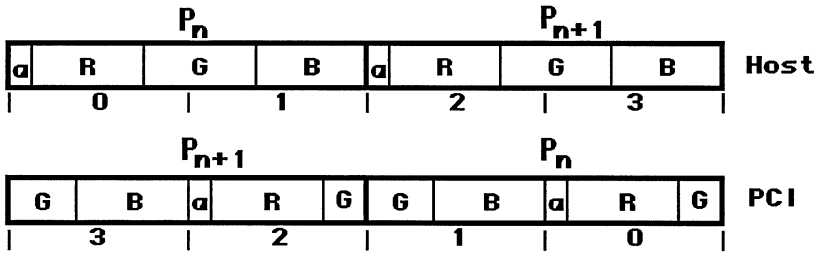
Note that not all transforms are "easy". In the this example, the video source's accesses are "write-only" to the frame buffer; reading pixels in YUV format would lose data. Likewise, reading pixels which are physically stored in 24-bit RGB can not provide loss-less read access through a 1-5-5-5 aperture; reading such pixels through an 8-bit CLUT aperture may not be possible at all.

An assumption made in this example is that each aperture represents the entire frame buffer. However, the height and width need not be the same between apertures or the physical buffer. Thus, for example, apertures could be used to present "sub-views" of the physical buffer.

3.3.3. Endian-ness Conversion

Another case where apertures are an appropriate mechanism is dealing with mixed "Endian" views of pixels; for example, a Big-Endian processor which is accessing a PCI Little-Endian frame buffer. The Host-PCI bridge would perform a "byte-swap" of data in order to guarantee the address invariance required by the PCI specification. While this guarantees that pixels are placed into their proper location (i.e., address and byte-lanes), it does cause the bytes within a pixel to be swapped.

If, for example, the pixel format was 1-5-5-5 RGB, a pixel written by the processor would appear on the PCI bus with its bytes swapped, albeit in the appropriate byte lanes. The following diagram shows two such pixels as viewed on the Host bus and on the PCI bus.



As with the Color Space example, a simplistic way of handling this would be to put the frame buffer controller into a "Big-Endian Mode". In this mode, the frame buffer controller would perform a "byte-unswap" for each pixel as it read from or wrote to the physical frame buffer. However, this has the same problem as with the Color Space example; it prevents convenient inter-operability with other PCI devices, most of which will be Little-Endian.

Instead, the frame buffer controller would implement (at least) two apertures. One aperture would present a Big-Endian view of the frame buffer while another presents a Little-Endian view. Note that the basic Big-Endian "transform" is a simple byte-swap based upon pixel size (e.g., 16-bit). Thus, it is relatively simple to implement in existing controller designs which support multiple pixel formats, since the basic multiplexing data-paths are already present.

3.3.4. Aperture Implementation

The "logical" view of a pixel buffer afforded by an aperture is specified by its starting address (i.e., the address of the top-left-most pixel), width, height and pixel format. While the width

and height parameters are usually thought of in terms of pixels, there is a direct mapping in terms of bytes as implied by the pixel format. (In addition, there is often another specification required -- "row-bytes" -- which is the address offset between successive rows.)

Because the aperture mechanism uses the "logical" view's pixel address to determine to which aperture a particular access is being made (conceptually, by matching base-address/ size pairs), the address-ranges of apertures must be different, regardless of any similarity between their other parameters. Once the aperture has been determined, attributes associated with that aperture (e.g., pixel format, Endian-ness) define what transformations are necessary to map between the aperture's logical view and the physical presentation.

A simple implementation of the apertures could use upper "alias" bits of the pixel buffer to determine to which aperture accesses belong. For example, if a frame buffer implemented 2 apertures, it could simply use the upper bit of an access's address to determine the aperture.

While this aliasing mechanism is probably the simplest method, others are possible. For example, multiple configuration base registers could be used, one for each aperture. Or, the size of the apertures could be non-power-of-two; this would require somewhat more hardware to detect to which aperture an access belongs, but consumes less address space.

3.3.5. Aperture Attributes

Apertures have associated attributes which specify the actions of the controller when accesses to the aperture are made. Among these attributes are:

- **Logical Pixel Format.** I.e., Color Space, Endian-ness. See section 6.2.
- **Logical size and organization**. I.e., base-address, width, height, row-bytes.
- **Physical mapping**. I.e., location of the Logical view within the Physical buffer.

3.3.6. Aperture Configuration

The Aperture mechanism assumes that the Device Driver for controllers support API calls which can query for the aperture capabilities of the controller and initialize an aperture on behalf of applications programs.

Apertures, in general, can be re-used; they can be treated as a "logical device" in that they can be "opened", accessed for some period of time, and then "closed", thus allowing re-use. I.e., their function need not be fixed at system startup time. Note, however, that a simple frame buffer with no transform or conversion capability still fits within the aperture model; it simply implements a single, "hard-bound" aperture.

The number of apertures, how aperture attributes are setup, etc. are not defined by this specification. The Device Driver model provides the abstraction layer which "hides" these low-level details from applications programs.

B.4 MacRISC Address Map

Introduction

This note describes the MacRISC Address Map. The map will be implemented in all MacRISC systems, with the possible exception of PDM. PDM is a "bridge" machine, and, hence, looks more like a traditional Mac II.

General Features

Since the MacRISC expansion strategy for both logic board and expansion slots is based upon PCI, the Address Map is designed with very few "locked down" address assignments. Instead, most of the chip and/or slot addresses are determined dynamically at "configuration" time, as part of the system startup code.

The most important fixed locations are those of the **System Controller** and the **ARBus-PCI Bridges**. The System Controller contains the information about the board type (BoardID), which informs software of the gross characteristics of the system (e.g., how many PCI bridges, general topology of the board, etc.) Using this BoardID, the startup code can then start locating and initializing the various chips and/or cards of the system.

Several general guidelines have been followed in the allocation of address space. Among these are:

Huge Main Memory (DRAM) Space

The Address Map allocates 2 GB of space for "DRAM" memory. This should cover the largest configuration of RAM for MacRISC systems.

Space for 4 ARBus-PCI Bridges

Any MacRISC system will contain at least one ARBus-PCI bridge. Higher end systems may contain multiple bridges, which are required because of the physical limitations of PCI and/or performance criteria. (The BoardID could indicate the actual number of bridges present.)

The address space allocated for PCI bridges includes sub-addresses that are used to generate PCI I/O Cycles. Also within this area are PCI Memory ranges for Apple-specific chips which need not participate in the generic PCI address allocation scheme (e.g., Grand Central, Blue Fish).

In order to perform PCI Config and Special cycles, a set of bridge "registers" is used. Only one location corresponds to a physical register (the Config Address Register); the other locations are used to trigger appropriate PCI cycles. To execute a Config cycle, the appropriate Config address (including **AD**[1:0], which controls PCI-PCI bridge pass-through of Config cycles) is stored into the bridge's Config Address Register. A subsequent access to its Config Data address will perform a PCI Config Read or Write cycle (depending upon whether a Load or Store access was being made). A similar mechanism is used for Special Cycles, using a different bridge Special Data address access.

All PCI I/O, Config, Special and Memory accesses are affected by the current "byte-swapping" mode of the PCI bridge chip (e.g., Bandit). Software can use the Byte Reversed forms of Loads and Stores to perform Configuration, etc.

Large ROM Space

The Address Map allocates 16 MB of space for "ROM"; the address space allocated allows the ROM to have a single decode (i.e., no aliasing is required). The ROM space is at the very top of address space, which directly maps to the processor's IP=1 space, which is the default address space for its ROM. Physical ROMs smaller than 16 MB occupy the top end of the space allocated for ROM.

Large Frame Buffer Space

Most MacRISC system's will contain a built-in frame buffer. A default starting address for this frame buffer has been pre-allocated within an area normally used for PCI allocation. The BoardID will indicate whether the frame buffer is accessed via PCI, or not.

In either case, the starting address of the frame buffer can be fixed within the Address Map. For a PCI-accessed frame buffer, the frame buffer's address and size are, in a sense, pre-allocated within the PCI space.

PCI Address Allocation

Following the philosophy of PCI, no fixed allocation for PCI devices (in either Memory or I/O space) is included in the Address Map. There is, however, a fixed allocation for addressing the ARBus-PCI bridge chips (e.g., Bandit), so that there is some "rooted" mechanism for walking the PCI(s) within a system, determining each device's address requirements and allocating address space appropriately.

The Address Map contains a large contiguous range (**0x80000000..0xEFFFFFFF**) and a smaller range (**0xF9000000..0xFEFFFFFF**) out of which to allocate PCI devices. Note that most Apple chips (e.g., Bandit, Grand Central, Blue Fish) can be allocated space within areas reserved by the ARBus-PCI bridges, thus, not taking away any of the "official" PCI space.

DBDMA Channel Register Space

DBDMA is the standard DMA Architecture defined for the MacRISC Architecture. DBDMA defines its own address allocation strategy, including addresses for channel controls and interrupts.

Since DBDMA devices are PCI devices, they do not have any fixed addressing. However, since every MacRISC will include at least one DBDMA (i.e., Grand Central), a special pre-allocation can be made within the PCI bridge spaces. There is sufficient address space within the bridge space that several DBDMA devices (e.g., Grand Central and ChefCat) can be allocated without "stealing" any other PCI address space.

DBDMA Device Register Space

While DBDMA explicitly defines the range for its Control and Status Registers, it does not directly address (no pun intended) the issue of direct access to the devices whose data is being transferred by DBDMA. I.e., the processor must directly access the Control and Status Registers of devices to initiate and control transfers.

A sub-range of the DBDMA spaces is allocated for such Device Registers. Since the early implementations of MacRISC (e.g., TNT) will be required to run existing Mac software (both Mac OS itself, and all those "ill-behaved" applications), the device register map allows the use of the same offsets for the VIA and SCC.

Control/Status/Configuration Register Space

Various Control and Status Registers may need to be implemented within any particular MacRISC implementation.

One important register has been given a "fixed" allocation in MacRISC. This is the so called "System ID" register, which will be at a fixed spot for software folke. Note that due to the nature of how "Box-ID" is determined (e.g., by reading VIA bits?), only the primary determination of "BoardID" is fixed by the MacRISC, which is allocated within the "System Control" space. Thus, there is no single register that contains all the "System ID"; mungy start-up code must "walk" the device tree, acquiring the necessary data as it goes.

NuBus Support

As part of a strategy of providing transition support for current users and developers, the Address Map allows a PCI-to-NuBus Bridge implemented as a PCI card that is cabled to an external NuBus chassis. All "NuBus" accesses are directed through this Bridge card to the NuBus chassis. Likewise, parent-board accesses from NuBus cards are made on their behalf by the Bridge card.

The mechanism for allocating PCI Memory address space allows a PCI-NuBus bridge to obtain addressing for up to 7 SuperSlots, although this would consume most of the available PCI space. A more realistic allocation would allow a full "rack" of NuBus Slots, with 3 SuperSlots (C,D,E) available, while still leaving a large address space free for normal PCI slots and devices.

> **Note: the ROM space exists in what would normally be NuBus Slot F. However, the Mac II model has always assumed that the maximum slot number is E; hence, there is no compatibility problem.**

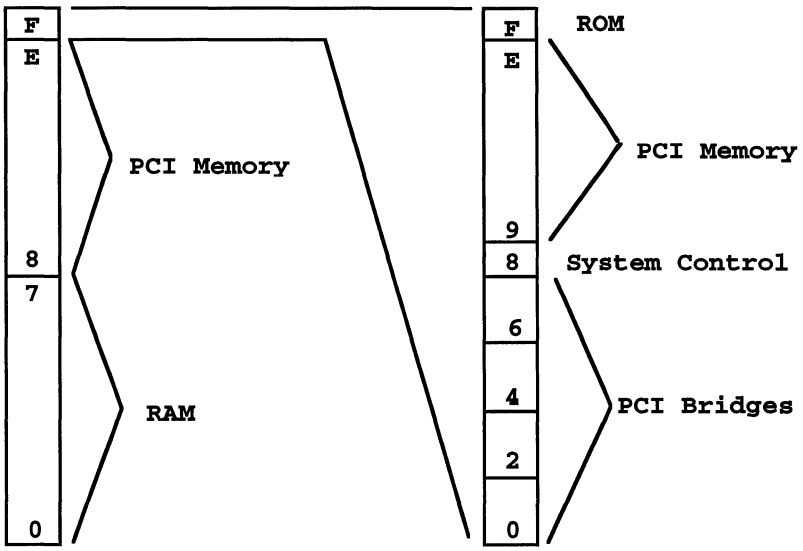
The Address Map

The following table gives the main address allocations. (See the following section for more details of the ARBus-PCI spaces.)

0x00000000-0x7FFFFFFF - Main Memory (DRAM) [2 GB]
0x80000000-0xF0FFFFFF - PCI "Memory"
0xF0000000-0xF8FFFFFF - PCI Bridges/System Control
0xF9000000-0xFEFFFFFF - PCI "Memory"
0xFF000000-0xFFFFFFFF - ROM

The following diagram shows the Address Map. The first bar represents the first nybble of an address while the second bar corresponds to the second nybble. (The following section shows sub-addressing

for subsequent address bits.)



ARBus-PCI Bridge Sub-Addressing

The above table and diagram show the major allocation of space for 4 ARBus-PCI bridges. This section shows how these ranges are further sub-allocated to allow the production of all possible PCI cycles from the processor. The upper 8 bits of an address are determined by the PCI Bridge number, ranging from **0x80..0x83**.

The next bits are used to encode the desired PCI cycle type as follows:

```
PCI Cycle
      I/O  1111  0bb0  0aaa aaaa aaaa aaaa aaaa aaaa  23-bit I/O Address
           1111  0bb0  10xx xxxx xxxx xxxx xxxx xxxx  Config Address Reg
   Config  1111  0bb0  110x xxxx xxxx xxxx xxxx xxxx   Config Data "Reg"
  Special/ 1111  0bb0  111x xxxx xxxx xxxx xxxx xxxx
    IntAck
   Memory  1111  0bb1  aaaa aaaa aaaa aaaa aaaa aaaa  24-bit Memory Address
                 bb = Bridge #
```

For more information about how ARBus-PCI bridges are supposed to function, see the ***MacRISC and PCI*** document.

Rev. Level	ECN #	Approved	Date	Revision Description
A	34359	Steve Polzin	05-Jul-95	Initial Release